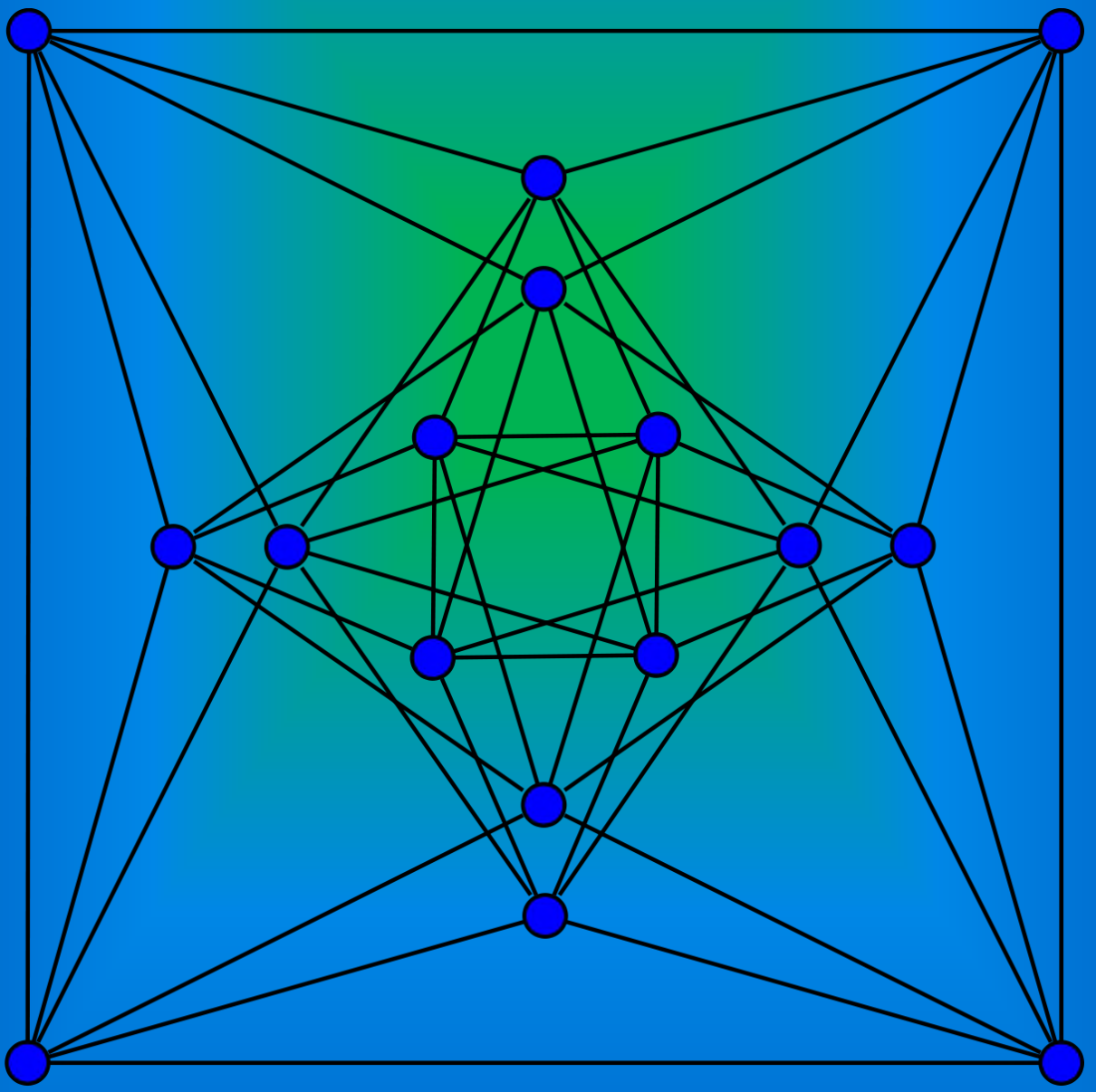
Алексеев В.В.

Теория алгоритмов

Учебно-методическое пособие



СарФТИ НИЯУ МИФИ

**Саровский физико-технический институт-
филиал Национального исследовательского
ядерного университета «МИФИ»**

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ЭЛЕКТРОНИКИ
Кафедра вычислительной и информационной техники

Алексеев В.В.

Теория алгоритмов

Учебно-методическое пособие

Саров
2021

УДК 621.
ББК 22.176
А47

Алексеев В.В. – Теория алгоритмов.
Учебно-методическое пособие. СарФТИ НИЯУ МИФИ, 2021 г.

Данное пособие представляет собой самостоятельный модуль и предназначено для студентов, изучающих теорию алгоритмов по информационным направлениям подготовки.

Пособие составлено на основании лекций, читающихся студентам факультета информационных технологий и электроники в СарФТИ НИЯУ МИФИ.

В пособии изложены основные положения теории алгоритмов, приведены основные его формализации, алгоритмические проблемы, рассмотрены классы сложности алгоритмов.

Теоретические сведения для наглядности иллюстрируются рисунками, таблицами. Даны примеры построения типовых алгоритмов.

Учебно-методическое пособие может быть полезно студентам и аспирантам других специальностей, использующих теорию алгоритмов и современные методы дискретного анализа в направлении своей подготовки.

Содержание

Основы теории алгоритмов

1. Определение алгоритма

В основе многообразных процессов обработки информации лежит понятие «алгоритм», который в принципе и определяет возможность автоматизации любых процессов деятельности человека, и, конечно, вычислительных. Поэтому понятие алгоритма относится к основным, базисным понятиям математики. Примерами алгоритма могут служить известные из школы методы умножения «столбиком», деления «уголком», методы решения систем линейных уравнений, правила дифференцирования сложных функций, построение геометрических фигур по заданным параметрам и др. Все многообразие вычислений основывается на использовании ограниченного количества операций алгебры, тригонометрии и анализа. Именно поэтому изначально понятие метода вычисления считалось прозрачным и не нуждалось в специальных исследованиях.

На протяжении долгого времени понятие алгоритма, которое подменялось термином «метод», было интуитивным и его можно было выразить примерно так: алгоритм - это строгая система правил, которая определяет последовательность действий над некоторыми объектами и после конечного числа шагов приводит к достижению поставленной цели. Здесь следует отметить, что система правил является алгоритмом, если любые исполнители, не знакомые с существом задачи, строго следуя данной системе правил, будут действовать одинаково и достигнут одного и того же результата.

Словесное описание алгоритмов математики использовали долгое время. Множество вычислительных алгоритмов формулировалось именно в такой форме (например, алгоритмы поиска корней квадратных и кубических уравнений и даже алгебраических уравнений любых степеней). В XVII веке Г. Лейбниц пытался найти общий алгоритм решения любых математических задач. Но только в XX веке, когда алгоритм стал объектом математического изучения, выдвинутая Г. Лейбницем идея, приобрела более конкретную форму. Уже в начале XX века Э. Борель (1912 г.), Г. Вейль (1921 г) пришли к понятию алгоритма как эффективной вычислительной процедуре, определив ее термином *вычислимой функции*.

Частичная формализация понятия алгоритма началась с попыток решения проблемы разрешения, которую сформулировал Давид Гильберт в 1928 году. Следующие этапы формализации были необходимы для

определения эффективных вычислений или «эффективного метода». Среди таких формализаций — рекурсивные функции Геделя — Эрбрана — Клини (1930 – 1935 г.г.), λ-исчисление Алонзо Чёрча (1936 г.). Понятие вычислимой функции было уточнено математиками А. Черчем, К. Геделем (1936 г.). Ими был определен класс частично рекурсивных функций, имеющих строгое математическое определение.

Одним из следующих формальных определений алгоритма является определение английского математика А.Тьюринга, который в 1936 году описал схему гипотетической (абстрактной) машины, имитирующей алгоритмические процессы, которые он и назвал алгоритмом в случае их успешной реализации, т.е. алгоритм – это то, что умеет делать такая машина. А если что-то не может быть сделано машиной Тьюринга, то это уже не является алгоритмом. Таким образом, А.Тьюринг формализовал правила выполнения действий при помощи описания работы некоторой конструкции.

Следует отметить, что вычислительная машина - это устройство для реализации алгоритмов, тогда как машина Тьюринга - это математическая модель, которая является абстракцией и никогда не была реализована, да и вообще не может быть реализована. Польза машины Тьюринга в том, на ее базе доказывается возможность реализации сложных алгоритмических процессов на основе простых операций, которые могут быть легко смоделированы на простых устройствах и тем самым доказать существование или не существование алгоритма решения задачи.

Описывая различные алгоритмы для своих машин и утверждая реализуемость всевозможных композиций алгоритмов, Тьюринг убедительно показал разнообразие возможностей предложенной им конструкции и высказал тезис: «Всякий алгоритм может быть реализован соответствующей машиной Тьюринга». Этот тезис является формальным определением алгоритма.

Примерно в одно время с А. Тьюрингом английский математик Э. Пост разработал похожую, но более простую алгоритмическую схему и реализующую ее машину. Позже было предложено еще несколько общих определений понятия алгоритма, и каждый раз удавалось доказать, что, хотя новые алгоритмические схемы и выглядят иначе, они в действительности эквивалентны машинам Тьюринга: все, что реализуемо в одной из этих конструкций, можно сделать и в других.

В 1954 году советский математик А.А. Марков предложил свою алгоритмическую схему преобразования слов и назвал ее нормальным

алгоритмом. Он ввел также понятие нормализации как перехода от разных способов описания алгоритмов к эквивалентным нормальным алгоритмам. Основная гипотеза теории алгоритмов в форме Маркова звучит так: «Всякий алгоритм нормализуем». Алгоритмическая схема Маркова, как и машина Тьюринга, в общем случае не может быть физически реализована, так как она, например, допускает неограниченно большую длину слов. А вот формулировка алгоритма по Маркову гласит: «Алгоритм - это точное предписание, которое задает вычислительный процесс, начинающийся с произвольного (но выбранного из фиксированной для данного алгоритма совокупности) исходного данного и направленный на получение полностью определяемого этим исходным данным результата».

Наиболее общий подход к уточнению понятия «алгоритм» было предложено советским ученым Колмогоровым А.Н., которым было дано еще и его «наглядное» представление: «Алгоритм, примененный ко всякому «условию» («начальному состоянию») из некоторого множества («области применимости» алгоритма), дает «решение» («заключительное состояние»). Алгоритмический процесс расчленяется на отдельные шаги заранее ограниченной сложности; каждый шаг состоит в «непосредственной переработке» (одного) состояния в (другое). Процесс переработки продолжается до тех пор, пока либо не произойдет безрезультатная остановка, либо не появится сигнал о получении «решения». При этом не исключается возможность неограниченного продолжения процесса…»

Поскольку точного и однозначного определения алгоритма не существует, приведем несколько наиболее распространенных определений его понятия.

1. **Алгоритм** (*algorithm*) - это набор инструкций, описывающих последовательность действий для достижения результата решения задачи за конечное число действий.
2. **Алгоритм** – это формально описанная вычислительная процедура, получающая исходные данные, называемые также входом алгоритма или его аргументом, и выдающая результат вычисления на выход.
3. **Алгоритм** – это набор правил, указывающих определенные действия, в результате которых входные данные преобразуются в выходные. Последовательность действий в алгоритме называется алгоритмическим процессом, а каждое отдельное действие – шагом алгоритма
4. **Алгоритм** – это общий, единообразный, точно установленный способ решения любой задачи из данной массовой проблемы.

Как видно из приведенных определений понятия алгоритма, его можно интуитивно определить как конечную последовательность точно заданных правил решения задач заданного класса.

Отметим, что можно выделить **вычислительные** алгоритмы, и **управляющие**. Вычислительные алгоритмы, как следует из их определения, преобразуют некоторые начальные данные в выходные, реализуя вычисление некоторой функции. Алгоритмический процесс управляющих алгоритмов, как правило, существенно отличается от вычислительных и сводиться к выдаче необходимых управляющих воздействий либо в заданные моменты времени, либо в качестве реакции на внешние события.

Алгоритмы, в которых решения поставленных задач сводятся к арифметическим действия, называют численными алгоритмами.

Также отметим, что с точки зрения современной практики часто под алгоритмом понимают программу вычислительного (управляющего) процесса, автоматически осуществляемого средствами вычислительной техники. В этом случае критерием алгоритмичности процесса является возможность его программирования. Именно благодаря этой современной реальности, заключающейся в конструктивном подходе к математическим методам вычисления и управления, понятие алгоритма сегодня стало очень популярным не только в математике, но и в различных сферах инженерной и научной деятельности.

Из приведенных определений понятий алгоритма следуют его особенности, которые состоят в следующем:

1. *Конечность*. Алгоритм всегда должен заканчиваться после выполнения конечного числа шагов;
2. *Определенность*. Каждый шаг алгоритма должен быть определен;
3. *Ввод*. Алгоритм имеет некоторое число входных данных, задаваемых до начала его работы;
4. *Вывод*. Алгоритм имеет одно или несколько выходных данных;
5. *Эффективность*. Алгоритм считается эффективным, если все его операции можно точно выполнить в течение конечного промежутка времени.

Естественно, что для алгоритма характерны три вида данных:

- входные (исходные) данные;
- промежуточные данные;
- выходные данные.

В общем случае, используемые данные алгоритмом и их взаимодействие можно представить в виде условной функциональной схемы, представленной на рис. 1.1. Важно отметить, что одной из главных особенностей алгоритма является то, что на одинаковый набор исходных данных всегда выдается одинаковый набор выходных данных.

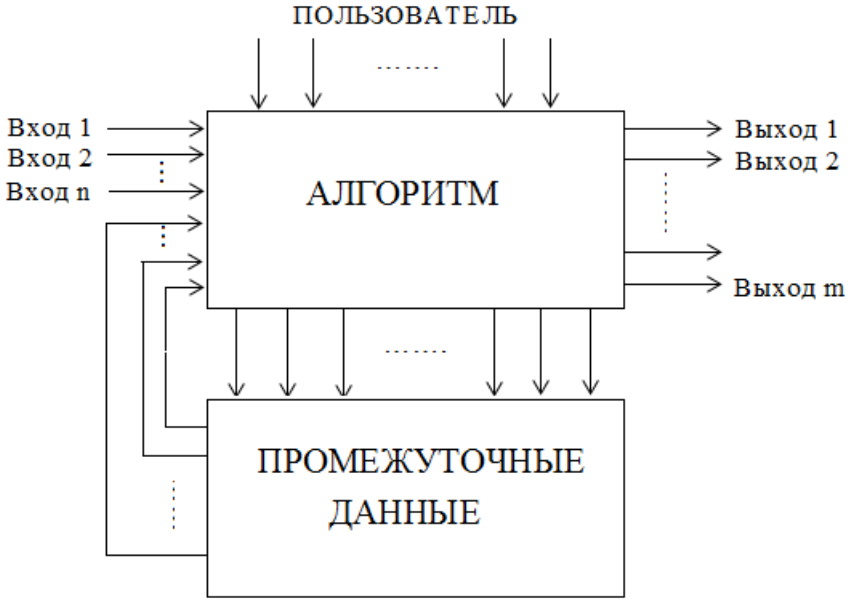


Рис. 1.1

В соответствии с вышеизложенным вполне стали определимы характерные черты алгоритма, четко вырисовывающиеся из его интуитивного определения и признающихся характерными для его понятия:

1. Алгоритм – это процесс последовательного построения величин, идущий в дискретном времени таким образом, что в начальный момент задается исходная конечная система величин (***правило начала***), а в каждый следующий момент получается новая система по определенному закону (программе) из системы величин, имевшихся в предыдущий момент времени (***дискретность алгоритма***);
2. Система величин, получаемых в какой-то (не начальный) момент времени, однозначно определяется системой величин, полученных в предшествующие моменты времени (***детерминированность алгоритма***);
3. Закон получения последующей системы величин из предшествующей должен быть простым и локальным (***элементарность шагов алгоритма***). (Типичный пример множества элементарных действий – система команд ЭВМ);

4. Если способ последующей величины из какой-нибудь заданной величины не дает результата, то должно быть указано, что надо считать результатом алгоритма (***результативность алгоритма***);
5. Начальная система величин может выбираться из некоторого потенциально бесконечного множества (***массовость алгоритма***)

Вполне очевидно, что понятие алгоритма, определяемое этими требованиями, конечно, не строгое, так как в формулировках встречаются слова «способ», «величина», «простой», «локальный» и др., точный смысл которых не установлен, и поэтому это нестрогое понятие алгоритма называют ***непосредственным***, или ***интуитивным понятием алгоритма***.

В рамках уточнения понятия алгоритма рассмотрим часто встречающуюся при изучении языков программирования задачу: для заданной конечной последовательности чисел A построить последовательность B, в которой числа последовательности A будут расположены в порядке возрастания.

Одним из простых и очевидных способов решения этой задачи является способ, заключающийся в следующем:

- просматриваем последовательность A и находим в ней наименьшее число;
- выписываем это число в качестве первого числа последовательности B и вычеркиваем его из последовательности A;
- снова обращаемся к последовательности A и находим в ней наименьшее число;
- приписываем его справа к последовательности B и вычеркиваем из последовательности A;
- продолжаем этот процесс до тех пор, пока не будут вычеркнуты все числа из последовательности A.

Для ясности перепишем этот способ решения в более четкой форме, разбив его на шаги и укажем переходы между шагами.

Шаг 1. Находим в A нименьшее число.

Шаг 2. Найденное число приписываем справа к B. (В начальный момент последовательность B пуста) и вычеркиваем его из A.

Шаг 3. Если в A нет чисел, то переходим у шагу 4, в противном случае переходим к шагу 1.

Шаг 4. Конец. (Получена последовательность B, являющаяся результатом данного решения).

На первый взгляд приведенной описание задачи и ее решения кажется вполне ясным для получения нужного результата. Но это впечатление ясности опирается на некоторые неявные предположения, к правильности которых мы привыкли, но которые, оказывается, легко и нарушить. Например, что значит “заданная последовательность чисел"? Является ли таковой последовательность $\sqrt[7]{3}, \sqrt[5]{2}, (1.2)^{\pi}, \sqrt[8]{lg\,\pi}$? Очевидно, да, но в нашем описании ничего не сказано, как найти наименьшее среди таких чисел, т.к. предполагается, что речь, очевидно, идет о числах, представленных в виде десятичных дробей, и что известно, как их сравнивать.

Таким образом, необходимо уточнить формы представления данных, и нельзя просто заявлять, что допустимо любое представление чисел, так как для каждого представления существует свой алфавит, и свой способ сравнения чисел (например, переводом в десятичную дробь).

Из приведенного примера описания алгоритма видна его сходимость и детерминированность, которые и создают впечатление ясности. Заметим, что здесь используется общепринятое соглашение: если шаг не содержит указаний о дальнейшем переходе, то выполняется следующий за ним шаг в описании.

Связь между шагами можно изобразить в виде графа, как показано на Рис. 1.2.

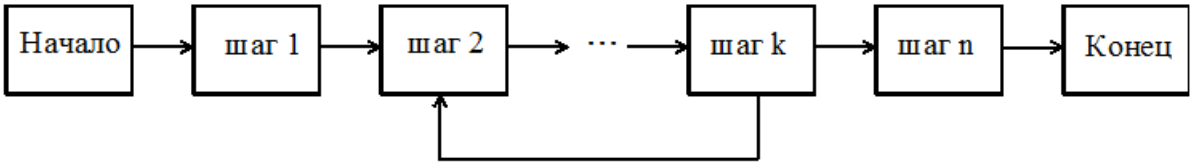


Рис. 1.2

Такой граф, в котором вершинам соответствуют шаги, а ребрам – переходы между шагами, называется блок-схемой алгоритма. Его вершины могут быть двух видов: вершины, из которых выходит одно ребро, называемые *операторами*, и вершины, из которых выходят два ребра, называемые *логическими условиями* или *предикатами*. Имеется единственный оператор начала и единственный оператор конца, из которого не выходит ни одного ребра (что соответствует правилу начала и правилу окончания соответственно определения алгоритма).

Заметим так же, что важной особенностью блок-схемы является то, что описываемые ей связи не зависят того, являются шаги элементарными или представляют собой самостоятельные алгоритмы. Это позволяет алгоритм разбивать на блоки, так что каждый блок будет представлять

самостоятельную алгоритмическую “единицу”, что и используется при программировании больших алгоритмов. Для отдельного блока неважно структура других блоков. Для программирования блока достаточно знать исходную для него информацию, форму ее представления и преобразования и место размещения результата.

Также с помощью блок-схем можно несколько алгоритмов, рассматриваемых как блоки, связать в один большой алгоритм. Например, если алгоритм A_1 вычисляющий функцию $f_1(x)$соединен с алгоритмом A_2, вычисляющий функцию $f_2(x)$, для которого результат алгоритма A_1 является исходными данными, то полученная блок-схема (Рис. 1.3) задает алгоритм, вычисляющий функцию $f_2(f_1(x))$, т.е. композицию функций f_1 и f_2. Такое соединений алгоритмов называется *композицией* алгоритмов.

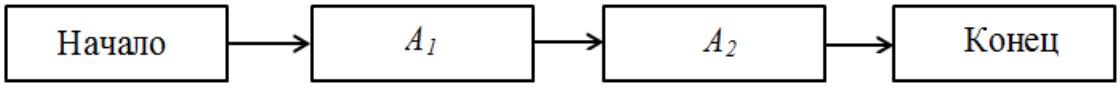


Рис. 1.3

Следует заметить, что блок-схема отражает разницу между описанием алгоритма и процессом его реализации. Описание – это граф, а процесс реализации – это путь в графе. Т.е. по существу, блок-схемы –это не язык, а просто очень удобное средство для описания детерминированности алгоритма.

2.Формализация понятия алгоритма. Универсальные модели алгоритмов

Интуитивное понятие алгоритма обладает целым рядом недостатков. Очевидно, что такие понятия, использованные при описании общих свойств алгоритмов, как ясность, элементарность шагов и т.п., сами нуждаются в уточнении. Очевидно, что их словесные определения будут содержать новые понятия, которые снова потребуют уточнения и т.д. Как уже отмечалось, начиная с 30-х годов, было предложено несколько уточнений понятия алгоритма. Считается, что все они достаточно полно отражают основные черты интуитивного понятия алгоритма. Действительно, все формальные определения алгоритма в некотором смысле эквивалентны друг другу. Поэтому в теории алгоритмов применяется другой подход: выбирается конечный набор исходных объектов, которые объявляются элементарными и конечный набор способов построения их них новых объектов. Этот метод был уже использован в теории множеств и получил название *конструктивного подхода*.

Алгоритмические модели, которые претендуют на право считаться формализацией понятия «алгоритм», должны быть универсальными, т.е. допускать описание любых алгоритмов.

Можно выделить три основных типа универсальных алгоритмических моделей, различающихся исходными эвристическими соображениями относительно того, что такое алгоритм. Первый тип связывает понятие алгоритма с наиболее традиционными понятиями математики – вычислениями и числовыми функциями. Наиболее развитая и изученная модель этого типа – ***рекурсивные функции*** – является исторически первой формализацией понятия алгоритма.

Второй тип модели связан с развитием вычислительной техники и основан на представлении об алгоритме как о некотором детерминированном устройстве, способном выполнять в каждый отдельный дискретный момент времени весьма примитивные операции. Такое представление не оставляет сомнений в однозначности алгоритма и элементарности его шагов. Кроме того, эвристика этой модели близка к ЭВМ и, следовательно, к инженерной интуиции. Основной теоретической моделью этого типа является, как уже отмечалось ранее, созданная в 30-х годах концепция машины Тьюринга. Именно ***машина Тьюринга*** явилась моделью современной ЭВМ и способствовала развитию современной вычислительной техники.

Третий тип алгоритмических моделей – это преобразование слов в произвольных алфавитах, в которых элементарными операциями являются подстановки, т.е. замены части слова (подслова) другим словом. Преимущества этого типа моделей заключаются в максимальной абстрактности и возможности применить понятие алгоритма к объектам произвольной, не обязательно числовой природы. Примерами моделей этого типа являются канонические системы Поста и нормальные ***алгоритмы Маркова***. При этом общность формализации в конкретной модели не теряется и доказывается сводимость одних моделей к другим, т.е. показывается, что всякий алгоритм, описанный средствами одной модели, может быть описан средствами другой.

Благодаря взаимной сводимости моделей в общей теории алгоритмов удалось выработать инвариантную по отношению к моделям систему понятий, позволяющую говорить о свойствах алгоритмов независимо от того, какая формализация алгоритма выбрана. Эта система понятий основана на понятии вычислимой функции, т.е. функции, для вычисления которой существует алгоритм.

2.1 Рекурсивные функции

2.1.1Примитивно-рекурсивные функции

Всякий алгоритм однозначно ставит в соответствие исходным данным (в случае если определен на них) определенный результат. Поэтому с каждым алгоритмом однозначно связана функция, которую он вычисляет. Исследование этих вопросов, как уже отмечалось, привело к созданию в 30-х годах прошлого века ***теории рекурсивных функций***. В этой теории, как и вообще в теории алгоритмов принят конструктивный (финитный) подход, основной чертой которого является то, что все множество исследуемых объектов (в данном случае функций) строится из конечного числа исходных объектов – базиса – с помощью простых операций, эффективная вычислимость которых достаточна очевидна. Операции над функциями называют ***операторами.***

Для простоты будем рассматривать только числовые функции, т.е. функции, аргументы и значения которых принадлежат множеству натуральных чисел N (в теории рекурсивных функций полагают N=0, 1, 2, …). Иначе говоря, числовой n-местной функцией $f(x_1, x_2, \cdots x_n)$ называется функция, определенная на некотором подмножестве $N \subseteq N^n$ с натуральными значениями. Как известно, если область определения $f : N^n \to N$ совпадает с множеством N^n,то говорят, что функция f всюду определенна, в противном случае – частично определенна.

Например, двуместная функция $f(x,y) = x + y$ является всюду определенной, а функция $f(x,y) = x - y$ - частично определена.

Рекурсивным определением функции принято называть такое определение, при котором значения функции для данных аргументов определяются значениями функции для более простых аргументов (уже вычисленных) или значениями более простых функций.

Одним из простых примеров рекурсивного определения являются, например, числа Фибоначчи, удовлетворяющие условию $f(n+2) = f(n+1) + f(n)$.

Все известные примеры алгоритмов можно свести к вопросу вычисления значений подходящей функции. Естественно назвать функцию, значения которой могут находиться с помощью некоторого алгоритма, ***вычислимой функцией***. Таким образом, ***вычислимая функция*** – это такая функция, для которой существует алгоритм, вычисляющий ее значения. Отметим, что вычисление функции происходит последовательно по определенным, заранее заданным правилам и инструкциям.

Так как в этом определении алгоритм понимается в интуитивном смысле, то и понятие вычислимой функции является ***интуитивным***.

Частичными числовыми функциями $f(x_1, x_2, \cdots x_n)$, где $x_i \in N \quad (1 \le i \le n)$, называют функции, определенные не на всех наборах $(x_1, x_2, \ldots, x_n) \in N^n$.

Простейшими и всюду определенными числовыми функциями являются:

- ***нуль функция*** $0(x)=0$;
- ***функция следования*** $S(x)=x+1$;
- ***функция проекции*** $I_m^n(x_1, x_2, \cdots, x_n) = x_m \quad (m \le n)$.

Далее определим операторы. Сразу следует отметить, что они обладают тем свойством, что, применяя их к функциям, вычислимым в интуитивном смысле, получаем функции, также вычислимые в интуитивном смысле.

2.1.2 <u>Оператор суперпозиции.</u> Суперпозиция является мощным средством получения новых функций из уже имеющихся. Напомним, что суперпозицией называется любая подстановка функций в функции.

Оператором суперпозиции F_m^n называется подстановка в функцию от m переменных m функций каждая из которых зависит от n одних и тех же переменных. Суперпозиция дает новую функцию уже от n переменных. Например, для функций $h(x_1,x_2,\ldots,x_m)$, $g_1(x_1,x_2,\ldots,x_n)$, $g_2(x_1,x_2,\ldots,x_n)$, ... $g_m(x_1,x_2,\ldots,x_n)$ их суперпозиция дает новую функцию $f(x_1, x_2, \ldots, x_n)$:

$$F_m^n(h, g_1, g_2, \ldots, g_m) = h(g_1(x_1, x_2, \ldots, x_n), g_2(x_1, x_2, \ldots, x_n), \cdots g_m(x_1, x_2, \ldots, x_n)) = f(x_1, x_2, \ldots, x_n). \qquad (2.1)$$

В этом случае говорят, что n-местная функция $f(x_1, x_2, \ldots, x_n)$ получена с помощью оператора суперпозиции из m-местной функции $h(x_1,x_2,\ldots,x_m)$ и n-местных функций $g_1(x_1, x_2, \ldots, x_n)$, $g_2(x_1, x_2, \ldots, x_n), \cdots g_m(x_1, x_2, \ldots, x_n)$, если

$$f(x_1, x_2, \ldots, x_n) = h(g_1(x_1, x_2, \ldots, x_n), g_2(x_1, x_2, \ldots, x_n), \cdots g_m(x_1, x_2, \ldots, x_n)).$$

<u>2.1.3 Оператор примитивной рекурсии</u>. Оператор примитивной рекурсии R_n определяет $(n+1)$-местную функцию f через n-местную функцию g и $(n+2)$ - местную функцию h следующим образом:

$$\begin{cases} f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n); \\ f(x_1, x_2, \ldots, x_n, y+1) = h(x_1, x_2, \ldots x_n, y, f(x_1, x_2, \ldots x_n, y)). \end{cases} \qquad (2.2)$$

Пара равенств (2.2) называется схемой примитивной рекурсии.

Тот факт, что функция f определена схемой (2.2) выражается равенством $f(x_1, x_2, \ldots x_n, y) = R_n(g, h)$. Эта схема определяет f

рекурсивно не только через другие функции, но и через значения f в предшествующих точках: значение f в точке $y+1$ зависит от значения f в точке y. Для вычисления $f(x_1,x_2,\ldots,x_n,k)$ понадобится $k+1$ вычислений по схеме (2.2) для $y=0, 1, \ldots, k$. Существенным в операторе примитивной рекурсии является то, что независимо от числа переменных в f рекурсия ведется только по одной переменной y, а остальные n переменных $x_1, x_2, \ldots, x_n$ на момент применения схемы (2.2) зафиксированы и играют роль параметров.

В случае, когда n=0, т.е. определяемая функция f является одноместной, схема (2.2) принимает более простой вид:

$$\begin{cases} f(0)=C; \\ f(y+1)=h(y,f(y)) \end{cases}, \qquad (2.3)$$

где C – константа.

Функция называется ***примитивно-рекурсивной***, если она может быть получена из нуль-функции $0(x)$, функции следования $S(x)$ и функции проекции I_m^n с помощью конечного числа применений операторов суперпозиции и примитивной рекурсии.

Этому определению можно придать более формальный индуктивный вид.

1. Функции $0(x)$, $S(x)$ и I_m^n для всех натуральных n, m, где $m \le n$, являются примитивно рекурсивными.
2. Если $g_1(x_1, x_2, \ldots, x_n), \ldots, g_m(x_1, x_2, \ldots, x_n), h(x_1, x_2, \ldots, x_n)$ примитивно рекурсивные функции, то $F_m^n(h,g_1,g_2,\ldots,g_m)$ – примитивно-рекурсивные функции для любых натуральных n, m.
3. Если $g(x_1, x_2, \ldots, x_n)$ и $h(x_1, x_2, \ldots, x_n, y, z)$ – примитивно рекурсивные функции, то $R_n(g,h)$ – примитивно-рекурсивная функция.
4. Других примитивно-рекурсивных функций нет.

Из такого индуктивного описания нетрудно извлечь процедуру, порождающую все примитивно-рекурсивные функции.

Пример 2.1 Доказать, что сложение, умножение, возведение в степень примитивно-рекурсивно:

1. Сложение $f_+(x,y)=x+y$ примитивно-рекурсивно.

   Решение.

   $f_+(x,0)=x=I_1^1(x);$

   $f_+(x,y+1)=f_+(x,y)+1=S(f_+(x,y)).$

Таким образом, $f_+(x,y) = h\big(x, y-1, f(x,y-1)\big) = R_1\big(I_1^1(x), h(x,y,z)\big)$,

где $h(x,y,z) = z+1$.

Действительно, имеем:

$f_+(x,0) = x;$

$f_+(x,1) = h(x,0,f_+(x,0)) = x+1;$

$f_+(x,2) = h(x,1,f_+(x,1)) = x+1+1 = x+2;$

$f_+(x,3) = h(x,2,f_+(x,2)) = x+2+1 = x+3;$

$\vdots \qquad \vdots \qquad \vdots \qquad \vdots$

$f_+(x,y-1) = h(x,y-2,f_+(x,y-2)) = x+(y-2)+1 = x+y-1;$

$f_+(x,y) = h(x,y-1,f_+(x,y-1)) = x+(y-1)+1 = x+y.$

2. Умножение $f_\times(x,y) = x \cdot y$ примитивно-рекурсивно.

Решение.

$f_\times(x,0) = 0;$

$f_\times(x,y+1) = x\cdot(y+1) = x\cdot y + x = f_\times(x,y) + x = f_+(x, f_\times(x,y));$

Или $f_\times(x,y+1) = h(x,y,f_\times(x,y)) = x+z$, где $z=f_\times(x,y)$.

Выполняя действия по полученной схеме, получаем:

$f_\times(x,0) = 0;$

$f_\times(x,1) = x+0 = x;$

$f_\times(x,2) = x+x = 2x;$

$f_\times(x,3) = x+2x = 3x;$

$\vdots \qquad \vdots \qquad \vdots$

$f_\times(x,y-1) = x+(y-2)x = x\cdot y - x;$

$f_\times(x,y) = x + x\cdot y - x = x\cdot y.$

3. Возведение в степень $f_{exp}(x,y) = x^y$ примитивно-рекурсивно.

Решение.

$f_{exp}(x,0) = x^0 = 1;$

$f_{exp}(x,y+1) = x^{y+1} = x^y \cdot x = x\cdot f_{exp}(x,y) = f_\times(x, f_{exp}(x,y)).$

$f_{exp}(x,y+1) = h(x,y,f_{exp}(x,y)) = x\cdot z),$ где $z = f_{exp}(x,y).$

Действительно, выполняя вычисления по полученной схеме, получаем;

$f_{exp}(x,0) = x^0 = 1;$

$f_{exp}(x,1) = x\cdot 1 = x;$

$$f_{exp}(x,2) = x \cdot x = x^2;$$
$$f_{exp}(x,3) = x \cdot x^2 = x^3;$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$f_{exp}(x,y-1) = x \cdot x^{y-2} = x^{y-1};$$
$$f_{exp}(x,y) = x \cdot x^{y-1} = x^y.$$

2.1.4 Оператор минимизации

Пусть задана некоторая функция $f(x,y)$. Зафиксируем значение x и выясним, при каком y функция $f(x,y)=0$.

Более сложной задачей является отыскание для данной функции $f(x,y)$ и фиксированного x наименьшего из тех значений y, при которых функция $f(x,y)=0$. Так как результат решения задачи зависит от x, то наименьшее значение y, при котором функция $f(x,y)=0$ есть функция x. Принято обозначение

$$\varphi(x) = \mu y[f(x,y) = 0], \qquad (2.4)$$

которое читается как: «*наименьшее y такое, что f(x,y)=0*».

Аналогично определяется функция многих переменных:

$$\varphi(x_1,x_2,\ldots,x_n) = \mu y[f(x_1,x_2,\ldots,x_n,y) = 0]. \qquad (2.5)$$

Переход от функции $f(x_1,x_2,\ldots,x_n,y)$ к функции $\varphi(x_1,x_2,\ldots,x_n)$ принято называть применением ***μ-оператора***.

Для вычисления функции φ можно использовать следующий алгоритм:

1.Вычислим $f(x_1,x_2,\ldots,x_n,0)$. Если это значение $f(x_1,x_2,\ldots,x_n,0)$ равно нулю, то пролагаем $\varphi(x_1,x_2,\ldots,x_n)=0$. Если $f(x_1,x_2,\ldots,x_n,0)\neq 0$, то переходим к следующему шагу.

2. Вычислим $f(x_1,x_2,\ldots,x_n,1)$. Если $f(x_1,x_2,\ldots,x_n,1)=0$, то полагаем $\varphi(x_1,x_2,\ldots,x_n)=1$. Если же $f(x_1,x_2,\ldots,x_n,1)\neq 0$, то переходим к следующему шагу. И т.д.

Если окажется, что для всех y функция $f(x_1,x_2,\ldots,x_n,y)\neq 0$, то функцию $\varphi(x_1,x_2,\ldots,x_n)$ в этом случае считают неопределенной. Но возможно, что существует такое y_0, что $f(x_1,x_2,\ldots,x_n,y_0)=0$ и, значит, есть и наименьшее y, при котором $f(x_1,x_2,\ldots,x_n,y)=0$, и в то же время может случиться, что при некотором z ($0<z<y_0$) значение функции $f(x_1,x_2,\ldots,x_n,z)$ не определено. Очевидно, что в этом случае процесс

вычисления наименьшего y, при котором $f(x_1,x_2,\ldots,x_n,y)=0$ не дойдет до y_0. И здесь функцию $\varphi(x_1,x_2,\ldots,x_n)$ считают неопределенной.

Пример 2.2

С помощью оператора минимизации реализовать функцию $f(x,y)=x-y$.

Данная функция может быть построена с помощью оператора минимизации следующим образом:

$$f(x,y)=\mu z[I_3^2(x,y,z)+I_3^3(x,y,z)+I_3^1(x,y,z)].$$

Вычислим, например, $f(8,3)$, т.е. значение функции при y=3, x=8. Для этого положим y=3 и будем придавать x последовательно значения:

z=0, 3+0=3≠8;
z=1, 3+1=4≠8;
z=2, 3+2=5≠8;
z=3, 3+3=6≠8;
z=4, 3+4=7≠8;
z=5, 3+5=8=8;

Таким образом, $f(8,3)$=5.

Итак, из простейших функций: константы нуля (0(x)=0), функции следования ($S(x)=x+1$) и функции проекции ($I_m^n(x_1,x_2,\cdots,x_n)=x_m$) с помощью операторов суперпозиции и примитивной рекурсии может быть получено огромное разнообразие функций. Тем самым выяснено, что эти функции имеют примитивно-рекурсивное описание, которое однозначно определяет процедуру их вычисления. Следовательно, они естественно относятся к классу вычислимых функций.

Также следует отметить, что, во-первых, все примитивно-рекурсивные функции всюду определены. Это следует из того, что простейшие функции всюду определены и операторы F_m^n и R_n это свойство сохраняют. Во-вторых, строго говоря, мы имеем дело не с функциями, а с их примитивно-рекурсивными описаниями. Эти описания также можно разбить на классы эквивалентности, отнеся в один класс все описания, задающие одну и ту же функцию. Однако задача распознавания эквивалентности примитивно-рекурсивных описаний, как будет далее показано, алгоритмически неразрешима.

2.2 Частично-рекурсивные функции. Тезис Чёрча

Функции, которые могут быть получены из простейших функций $0(x)$, $S(x)$ и $I_m^n(x_1,x_2,\cdots,x_n)$ помощью конечного числа применений операций суперпозиции, примитивной рекурсии и μ-оператора называются ***частично рекурсивными***.

Частично-рекурсивная функция называется ***общерекурсивной***, если она всюду определена.

Вполне очевидно, что множество общерекурсивных функций включает в себя множество примитивно рекурсивных функций, а частично рекурсивные функции включают в себя общерекурсивные функции. Известно, что не всякая общерекурсивная функция является примитивно рекурсивной. В то же время существуют частично рекурсивные функции, которые не могут быть продолжены до общерекурсивных. Заметим, что частично рекурсивные функции иногда называют просто рекурсивными функциями. Для любой частично рекурсивной функции можно указать алгоритм вычисления ее значений, т. е. все частично рекурсивные функции суть вычислимые функции.

Рассмотрим, как выполняются основные требования к алгоритмам в построенной алгоритмической модели – частично-рекурсивные функции.

Детерминированность определяется полной определенностью в вычислении простейших функций $0(x)$, $S(x)$, $I_m^n(x_1,x_2,\cdots,x_n)$, а также полной заданностью в действиях операторов суперпозиции, примитивной рекурсии и μ-оператора. Все соответствующие шаги подробно и однозначно описаны при их определении. Там же и показана ***элементарность*** каждого шага и ***дискретность*** вычислений.

Результативность определяется значением частично-рекурсивной функции, которое вычисляется в процессе реализации описанных операторов. Если функция не определена, то процесс вычислений не останавливается.

Массовость частично-рекурсивных функций определяется возможностью выбора в качестве аргумента любого натурального числа.

Таким образом, понятие частично-рекурсивной функции является исчерпывающей формализацией понятия вычислимой функции. Это обстоятельство выражено в виде **тезиса Чёрча:** ***всякая функция, вычислимая некоторым алгоритмом, частично-рекурсивна***.

Таким образом, согласно этому принципу класс функций, вычислимых с помощью алгоритмов в широком интуитивном смысле, совпадает с классом частично рекурсивных функций. Тезис Чёрча не

может быть строго доказан, т.к. он связывает нестрогое математическое понятие интуитивно вычислимой функции со строгим математическим понятием частично-рекурсивной функции.

Следует заметить, что этот тезис может быть опровергнут, если построить пример вычислимой функции, но не являющейся частично-рекурсивной.

3. Машина Тьюринга

3.1 Реализация математической модели Тьюринга

Другой формальной математической моделью понятия «алгоритм» является, как отмечалось, машина Тьюринга. Рассмотрим этот вопрос более подробно.

Итак, если для решения некоторой массовой проблемы известен алгоритм, то для его реализации необходимо лишь четкое выполнение предписаний этого алгоритма. Процесс, необходимый для реализации такого алгоритма естественно приводит к мысли о передаче функции человека, реализующего алгоритм, некоторому устройству–машине. Идею такой машины предложили в 1937 году американский математик Э. Пост и английский математик А. Тьюринг.

Машина Тьюринга включает:

1) Управляющее устройство, которое может находиться в одном из состояний, образующих конечное множество $Q=\{q_1, q_2, \dots, q_n\}$;
2) Бесконечную ленту, разбитую на ячейки, в каждой из которых может быть задан один из символов конечного алфавита $A=\{a_1, a_2, \dots, a_m\}$;
3) Устройство обращения к ленте, так называемую считывающую и пишущую (управляющую) головку, которая в каждый момент времени обозревает одну ячейку ленты и в зависимости от символа в этой ячейке и состояния управляющего устройства:
    a) записывает в ячейку символ (быть может, совпадающий с прежним или пустой);
    b) сдвигается на ячейку влево или вправо или остается на месте;
    c) переходит в новое состояние (или остается в прежнем).

Отметим, что конечный алфавит $A=\{a_1, a_2, \dots, a_m\}$ можно трактовать как ***внешний алфавит***. В этом алфавите в виде слова кодируется та информация, которая подается в машину. Машина перерабатывает информацию, поданную в виде слова, в новое слово. Множества $Q=\{q_1, q_2, \dots, q_n\}$ и $\{R, L, E\}$ рассматриваются как ***внутренний алфавит***. Символы $q_1, q_2, \dots, q_n$ выражают конечное число состояний машины. Для любой

машины число состояний фиксировано. Два состояния имеют особое назначение: q_1 – начальное состояние машины, q_z – заключительное состояние (стоп-состояние). В начальном состоянии машина находиться перед началом работы, а попав в заключительное состояние q_z, останавливается. Символы *R, L, E* – это символы сдвига (вправо, влево, на месте соответственно. Конечно, что все эти обозначения условно формальны и довольно часто используют и другие символы для обозначения и элементов внешнего алфавита, и операций сдвига (как например, при П, Л, Н, что соответствует операциям вправо (*R*), влево (*L*), на месте (*E*) соответственно, а состояние «останова», т.е. заключительное состояние, обозначают, например как q_s или q_0).

Структурная схема машины Тьюринга приведена на рис. 3.1

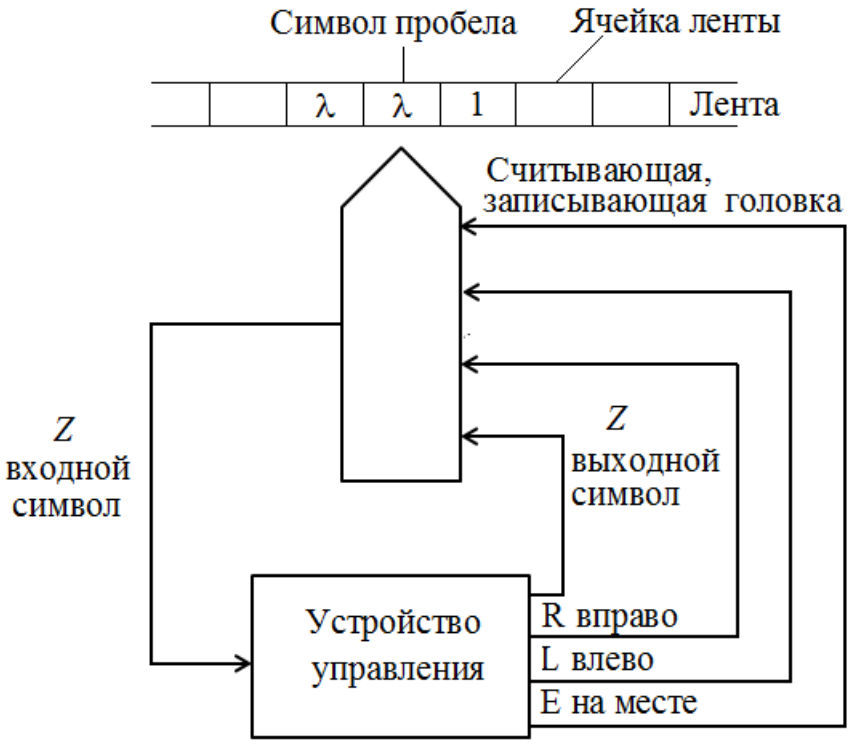


Рис. 3.1

Рассмотрим теперь, как выполняются основные требования к алгоритмам к алгоритмической модели – машине Тьюринга.

Данные машины Тьюринга – это слова в алфавите ленты.

Память машины Тьюринга – конечное множество состояний (внутренняя память) и лента (внешняя память). Лента бесконечна в обе стороны, однако в начальный момент времени только конечное число ячеек ленты заполнено непустыми символами, остальные ячейки пусты, т.е. содержат пустой символ, например, λ (пробел). Из характера работы

машины следует, что в любой последующий момент времени лишь конечный отрезок ленты будет заполнен символами.

Детерминированность машины определяется тем, что для любого внутреннего состояния q_i и символа a_j однозначно заданы:

a) следующее состояние q_i';

b) символ a_i', который нужно записать вместо a_j в ту же ячейку (стирание символа понимается как запись пустого символа λ);

c) направление сдвига головки d_k, обозначаемое одним из трех символов: *R* (вправо), *L*(влево), *E* (на месте).

Из вышесказанного следует, что любую машину Тьюринга можно описать либо системой команд (правил), имеющих, например, вид:

$$q_i a_j \to q_i' a_j' d_k, \qquad (3.1)$$

либо таблицей, строкам которой соответствуют состояния, столбцам – входные символы, а на пересечении строки q_i и столбца a_j записана тройка символов $q_i' a_j' d_k$, либо блок схемой, которую называют диаграммой переходов.

В диаграмме переходов состояниям соответствуют вершины, а правилу вида (3.1) - ребро, ведущее из q_i в q_i', на котором написано $a_j \to a_j' d_k$. Условие однозначности требует, чтобы для любого j и любого $i \neq z$ в системе команд имелась только одна команда, аналогичная (3.1), с левой частью $q_i a_j$. Состояние q_z (состояние «останова») в левых частях команд не встречается. На диаграмме переходов это выражается условием, что из каждой вершины, кроме q_z, выходит ровно m ребер, где m – число символов в алфавите ленты, и причем на всех ребрах, выходящих из одной вершины, левые части различны, а в вершине q_z нет выходящих ребер. Символы q_i' и a_j' можно опускать, если $q_i' = q_i$, $a_j' = a_j$.

Приведенное описание показывает также и ***дискретность*** машины Тьюринга.

Элементарные шаги машины – это считывание и запись символов, сдвиг на ячейку вправо или влево, а также переход управляющего устройства в следующее состояние.

Результатом работы машины Тьюринга является слово на ленте после остановки машины. Случай, когда машина не останавливается требует отдельного рассмотрения.

Массовость алгоритма (машины Тьюринга) обеспечивается возможностью выбора в качестве начальной системы любого слова в алфавите ленты.

Таким образом, приведенная здесь машина Тьюринга является конкретной алгоритмической моделью, претендующей на право формализации понятия «алгоритм».

Полное состояние машины Тьюринга, по которому однозначно можно определить ее дальнейшее поведение, определяется ее внутренним состоянием, состоянием ленты (т.е. словом, записанном на ленте) и положением головки на ленте. Полное состояние называют ***конфигурацией***, или ***машинным словом***, которое можно обозначить тройкой символов $\alpha_1 q_i \alpha_2$, где q_i – текущее внутреннее состояние, α_1 – слово слева от считывающей головки, а α_2 – слово, образованное символом, обозреваемым головкой, и словом справа от него, причем слева от α_1 и справа от α_2 нет пустых символов.

Пример 3.1. Конфигурация с внутренним состоянием q_i, в которой на ленте записано *abcde*, а головка обозревает ячейку с символом *b*, запишется как aq_ibcde (рис. 3.2),

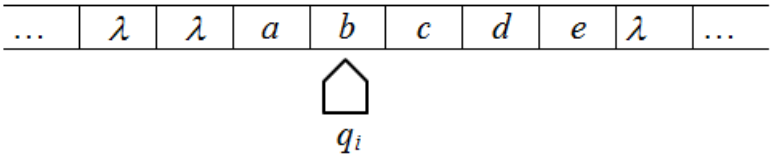


Рис. 3.2

Стандартной начальной конфигурацией называют конфигурацию вида $q_1\alpha$, т.е. конфигурацию, содержащую начальное состояние, в которой головка обозревает крайний левый символ слова, записанного на ленте.

Стандартной заключительной конфигурацией называют конфигурацию вида $q_z\alpha$.

Ко всякой не заключительной конфигурации K машины T применима ровно одна команда вида (3.1), которая переводит конфигурацию K в K'. Это обозначается как $K \to K'$. Если для K_1 и K_n существует последовательность K_1, K_2, … , K_n такая, что $K_1 \to K_2 \to \ldots \to K_n$, то это обозначают как $K_1 \Rightarrow K_n$.

Для простоты и удобства изображения различных конфигураций машины Тьюринга часто записывают информацию в виде слова, не изображая ленты и ее разбивки на клетки, а вместо изображения

управляющей головки и состояния машины записывать только состояние машины.

Рассмотрим пример реализации программы машины Тьюринга функциональной схемой, представленной в виде таблицы 3.1.

Таблица 3.1

	λ	a_1	a_2
q_1	$a_2 L\ q_3$	$a_1\ R\ q_2$	$a_2\ L\ q_1$
q_2	$\lambda\ E\ q_2$	$a_2\ E\ q_1$	$a_1\ E\ q_2$
q_3	$\lambda R\ q_z$	$a_1\ R\ q_4$	$a_2\ E\ q_1$
q_4	$a_1\ E\ q_3$	$\lambda\ R\ q_4$	$a_2\ R\ q_4$

Пусть начальная конфигурация имеет вид:

$$\lambda\ a_2\ a_2\ \lambda$$
$$q_1$$

Т.к. управляющей головкой обозревается символ a_2, а машина находится в состоянии q_1, то машина вырабатывает команду $a_2\ L\ q_1$, и в результате получаем вторую конфигурацию

$$\lambda\ a_2\ a_2\ \lambda.$$
$$q_1$$

Очевидно, следующие конфигурации будут иметь вид:

$\lambda\ a_2\ a_2\ \lambda$ - третья конфигурация,
q_1

$\lambda\ a_2\ a_2\ a_2\ \lambda$ - четвертая конфигурация,
q_3

$\lambda\ a_2\ a_2\ a_2\ \lambda$ - пятая конфигурация.
q_z

Так как после пятой конфигурации машина находится в состоянии q_z, то слово $a_2\ a_2\ a_2$ и является результатом работы машины Тьюринга по заданной программе. Таким образом, машина Тьюринга пришла к результату, выполнив команды:

$a_2\ q_1 a_2 \rightarrow q_1\ a_2\ a_2 \rightarrow q_1\ \lambda\ a_2\ a_2 \rightarrow q_3\ \lambda\ a_2\ a_2\ a_2 \rightarrow q_z\ a_2\ a_2\ a_2$.

Пример 3.2. Пусть для функциональной схемы, заданной таблицей 3.1 начальная конфигурация имеет вид:

$$\lambda\ a_1\ a_1\ a_2\ a_2\ \lambda$$
$$q_1$$

В соответствии с функциональной схемой алгоритма следующие конфигурации будут иметь вид:

$\lambda\, a_1\, a_1\, a_2\, a_2\, \lambda$ - вторая конфигурация,
q_1

$\lambda\, a_1\, a_1\, a_2\, a_2\, \lambda$ - третья конфигурация,
q_1

$\lambda\, a_1\, a_1\, a_2\, a_2\, \lambda$ - четвертая конфигурация,
q_2

$\lambda\, a_1\, a_1\, a_1\, a_2\, \lambda$ - пятая конфигурация,
q_2

$\lambda\, a_1\, a_1\, a_2\, a_2\, \lambda$ - шестая конфигурация.
q_1

Или работу алгоритма можно представить следующей схемой:
$a_1\, a_1\, a_2\, q_1\, a_2 \rightarrow a_1\, a_1\, q_1\, a_2\, a_2 \rightarrow a_1\, q_1 a_1\, a_2\, a_2 \rightarrow a_1\, a_1\, q_2\, a_2\, a_2 \rightarrow a_1\, a_1\, q_2 a_1\, a_2 \rightarrow$
$\rightarrow a_1\, a_1\, q_1\, a_2\, a_2 \rightarrow \ldots$.

Как видно из второй и шестой конфигураций, процесс работы машины начал повторяться, и, следовательно, машина будет работать бесконечно и результата не будет. (Как говорят программисты – машина «зациклилась»).

Если, например, $a_1 q_1 a_2 \Rightarrow b_1 q_z b_2$, то говорят, что машина T перерабатывает слово $a_1 a_2$ в слово $b_1 b_2$, и обозначают это как $T(a_1 a_2) = T(b_1 b_2)$. Запись $T(a)$ используется для обозначения машины T с исходными значениями a.

Еще раз подчеркнем, что исходные, промежуточные и результат записываются на ленте в некотором алфавите A. Кроме того, могут быть выделены специальные символы для представления формата данных. Исходными данными машины Тьюринга считаются записанные на ленте слова в алфавите исходных данных $A_{исх}$ ($A_{исх} \subseteq A$) и векторы из таких слов (словарные векторы над $A_{исх}$). Это значит, что для каждой машины будут рассматриваться только те начальные конфигурации, в которых на ленте записаны словарные векторы над $A_{исх}$. Запись на ленте словарного вектора ($a_1, a_2, \ldots, a_k$) называют правильной, если она имеет вид ($a_1 * a_2 * \ldots * a_k$), где * -специальный символ разделитель (маркер), не входящий в $A_{исх}$. Для любого вектора $V_{исх}$ над $A_{исх}$ машина T либо работает бесконечно, либо перерабатывает его в словарный вектор в алфавите, который называют алфавитом результатов и который обозначим $A_{рез}$. Заметим, что $A_{исх}$ и $A_{рез}$ могут пересекаться и даже совпадать. В ходе работы на ленте могут появляться символы, не входящие в $A_{исх}$ и $A_{рез}$ и образующие

промежуточный алфавит $A_{пр}$ (содержащий, в частности, разделитель). Таким образом, алфавит ленты $A = A_{исх} \cup A_{рез} \cup A_{пр}$. В простейшем случае $A_{исх} = A_{рез}$ и $A = A_{исх} \cup \{\lambda\}$.

Пусть f – функция, отображающая множество векторов (слов) используемого алфавита $A_{исх}$ в множество векторов $A_{рез}$. Машина Тьюринга T правильно вычисляет функцию f, если:

1) для любых векторов V и W, таких, что $f(V)=W$

$$q_1V^* \Rightarrow q_zW^*,$$

где V^* и W^* - правильные записи V и W соответственно;

2) для любого вектора V такого, что $f(V)$ не определено, машина T, запущенная в стандартной начальной конфигурации q_1V^*, работает бесконечно.

Если для функции f существует машина T, которая ее правильно вычисляет, то f называется правильно ***вычислимой по Тьюрингу***.

С другой стороны, всякой правильно вычисляющей машине Тьюринга можно поставить в соответствие вычислимую ею функцию.

Две машины Тьюринга с одинаковым алфавитом $A_{исх}$ называют эквивалентными, если они вычисляют одну и ту же функцию

Определения, связанные с вычислением функций, заданных на словарных векторах, имеют в виду переработку нечисловых объектов. Конечно, понадобятся числовые функции, а точнее, функции отображающие N в N. Для этого можно представлять числа в единичном (унарном) коде, т.е. для всех числовых функций $A_{исх} = \{1\}$, либо $A_{исх} = \{1, *\}$ и число x единиц будем представлять словом $1\ldots1 = 1^x$. Таким образом, числовая функция $f(x_1, x_2, \ldots, x_n)$ правильно вычислима по Тьюрингу, если существует машина T такая, что $q_1 1^{x_1} * 1^{x_2} * \ldots * 1^{x_n} \underset{T}{\Longrightarrow} q_z 1^y$, когда $f(x_1,x_2,\ldots,x_n)=y$, и T работает бесконечно, начиная с $q_1 1^{x_1} * 1^{x_2} * \ldots * 1^{x_n}$, когда $f(x_1, x_2, \ldots, x_n)$ не определена.

Следует отметить, что возможны и другие определения вычисления на машине Тьюринга.

Рассмотрим примеры вычисления некоторых функций по Тьюрингу.

Пример. 3.3. Сложение. С учетом введенного ранее представления чисел в унитарном коде сложение чисел a и b означает, что слово $1^a * 1^b$ переработать в слово 1^{a+b}, т.е. удалить разделитель * и сдвинуть одно из слагаемых, например, первое к другому. Это можно осуществить машиной T_+ с четырьмя состояниями и следующей системой команд:

$q_1 * \to q_z \lambda R$;
$q_1 1 \to q_2 \lambda R$;
$q_2 1 \to q_2 1R$;
$q_2 * \to q_3 1L$;
$q_3 1 \to q_3 1L$;
$q_3 \lambda \to q_z \lambda R$.

Первая команда введена для случая, когда a=0 и исходное слово имеет вид $*1^b$. Кроме этого, в этой системе перечислены не все сочетания состояний машины и символов ленты: опущены те из них, которые при стандартной начальной конфигурации никогда не встретятся. В таблицах, опущенные команды отмечаются прочерком. Например, алгоритм функциональной схемы сложения на T_+ приведен в таблице 3.2

Таблица 3.2

	*	λ	1
q_1	$\lambda R\ q_z$	-	$\lambda R\ q_2$
q_2	$1L\ q_3$	-	$1R\ q_2$
q_3	-	$\lambda R\ q_z$	$1L\ q_3$

Диаграмма переходов T_+ приведена на рис 3.3

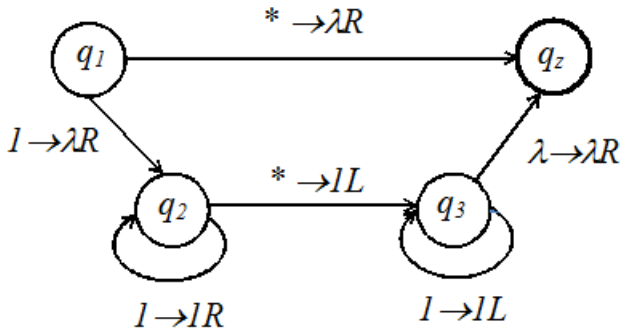


Рис. 3.3

Рассмотрим работу алгоритма на конкретном примере с начальной конфигурацией конфигурация имеет вид: λ 1 1 1 * 1 1 λ

q_1

λ 1 1 1 * 1 1 λ → λ λ 1 1 * 1 1 λ → λ 1 1 * 1 1 λ → λ 1 1 * 1 1 λ →
q_1 q_2 q_2 q_3

→ λ 1 1 1 1 1 λ → λ 1 1 1 1 1 λ → λ 1 1 1 1 1 λ.
q_3 q_3 q_z

Пример 3.4 Копирование слова, т.е. переработка слов a в $a*a$. Эту задачу решает машина $T_{коп}$ в соответствии с функциональной схемой алгоритма, представленного в таблице 3.3

Таблица 3.2

	1	λ	*	0
q_1	$0R\ q_2$	$\lambda R\ q_z$	$*L\ q_1$	$1L\ q_1$
q_2	$1R\ q_2$	$*R\ q_3$	$*R\ q_3$	
q_3	$1R\ q_3$	$1L\ q_4$		
q_4	$1L\ q_4$		$*L\ q_4$	$0Rq_1$

Диаграмма переходов $T_{коп}$ приведена на рис. 3.4. На этой диаграмме, а также последующих, приняты сокращения:

1) если из q_i в q_j ведут два ребра с одинаковой правой частью, то они объединяются в одно ребро, на котором левые части записаны через запятую;
2) если символ на ленте не изменяется, то он в правой части команды не пишется.

Отметим, что на петле q_4 использованы одновременно оба сокращения.

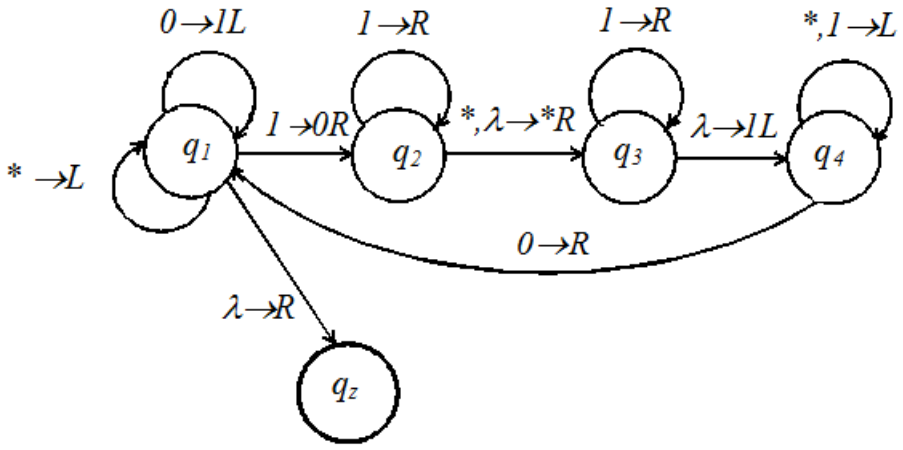


Рис.3.4

Машина $T_{коп}$ при каждом проходе исходного числа 1^a заменяет левую из его единиц нулем и пишет (в состоянии q_3) одну единицу справа от 1^a в ближайшую пустую клетку. При первом проходе, кроме того, в состоянии q_2 ставится маркер. Таким образом, копия 1^a строится за a проходов. После записи очередной единицы машина переходит в состояние q_4, которое передвигает головку влево от ближайшего нуля, после чего машина переходит в состояние q_1 и цикл повторяется. Он прерывается, когда q_1 обнаруживает на ленте не единицу, а маркер. Это значит, что все единицы слова 1^a исчерпаны, т.е. сделано a проходов. Тогда головка возвращается

влево в свое исходное положение, заменяя в процессе прохода все нули единицами.

3.2 Некоторые операции над машинами Тьюринга

Работа машины Тьюринга полностью определяется исходными данными и системой команд. С целью понимания того, как конкретная машина решает данную задачу, как правило, возникает потребность в содержательных пояснениях, которые, например, приводились для $T_{кол}$. Для того, чтобы сделать эти пояснения более формальными и точными используются блок-схемы и некоторые операции над машинами Тьюринга.

Теорема 3.1. Если функции $f_1(x)$ и $f_2(y)$ вычислимы по Тьюрингу, то их композиция $g(x)=f_2(f_1(x))$ также вычислима по Тьюрингу.

Доказательство. Пусть T_1 – машина, вычисляющая f_1, а T_2 – машина, вычисляющая f_2 и множества их состояний соответственно $Q_1 = \{ q_{11},\ldots q_{1n_1} \}$ и $Q_2 = \{ q_{21},\ldots q_{2n_2} \}$. Построим диаграмму переходов T из диаграмм T_1 и T_2 следующим образом: отождествим начальную вершину q_{21} диаграммы машины T_2 с конечной вершиной q_{1z} диаграммы машины T_1 (для систем команд это равносильно тому, что систему команд T_2 приписываем к системе команд T_1 и при этом q_{1z} в командах T_1 заменяем на q_{21}). Получим диаграмму с n_1+n_2-1 состояниями. Начальным состоянием T_1 объявим q_{11}, а заключительным - q_{2z}. Для простоты обозначений будем считать f_1 и f_2 числовыми функциями одной переменной.

Пусть $f_2(f_1(x))$ определена. Тогда

$$T_1(1^x) = 1^{f_1(x)} \text{ и } q_{11}1^x \underset{T_1}{\Longrightarrow} q_{1z}1^{f_1(x)}.$$

Машина T пройдет ту же последовательность конфигураций с той разницей, что вместо $q_{1z}1^{f_1(x)}$ она перейдет в $q_{21}1^{f_1(x)}$. Эта конфигурация является стандартной начальной конфигурацией для машины T_2, поэтому

$$q_{21}1^x \underset{T_2}{\Longrightarrow} q_{2z}1^{f_2(f_1(x))}.$$

Но так как все команды T_2 содержатся в T, то

$$q_{11}1^x \underset{T}{\Rightarrow} q_{21}1^{f_1(x)} \Rightarrow q_{2z}1^{f_2(f_1(x))}$$

и, следовательно, $T(1^*) = 1^{f_2(f_1(x))}$. Если же $f_2(f_1(x))$ не определена, то T_1 или T_2 не остановится и, следовательно, машина T не остановится. Итак, машина T вычисляет $f_2(f_1(x))$.

Построенную таким образом машину T называют композицией машин T_1 и T_2 и обозначают $T_2(T_1)$, а также изображают блок-схемой, показанной на рис. 3.5.

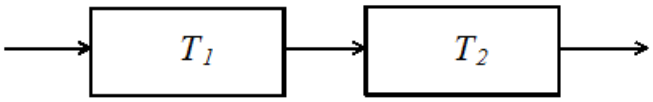


Рис. 3.5

Эта блок-схема всегда предполагает, что исходными данными машины T_2 являются результаты T_1.

Отметим, что определение композиции и вышеприведенная теорема остаются в силе, если T_1 и T_2 вычисляют функции от нескольких переменных. Важно лишь, чтобы данные для T_2 были в обусловленном виде подготовлены машиной T_1. Это видно на следующем примере.

Пример 3.5 Машина, диаграмма переходов которой приведена на рис. 3.6, это машина $T_+(T_{кол})$. Она вычисляет функцию $f(x)=2x$ для $x\neq 0$. При этом машина $T_{кол}$ строит двухкомпонентный вектор, а T_+ вычисляет функцию от двух переменных.

Для удобства последующих построений установим следующий важный факт. Оказывается, что для вычисления на машине Тьюринга достаточно, чтобы лента была бесконечной только в одну сторону, например, вправо. Такая лента называется правой полулентой.

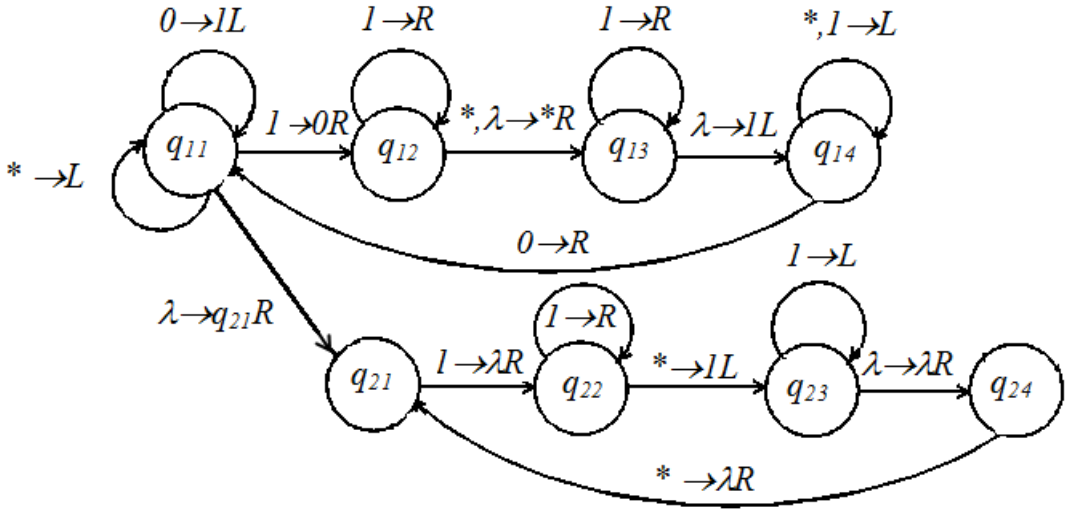


Рис.3.6

Теорема 3.2 Любая функция, вычислимая по Тьюрингу, вычислима на машине Тьюринга с правой полулентой. Иначе говоря, для любой машины Тьюринга T существует эквивалентная ей машина T_R с правой полулентой. Аналогичная теорема формулируется для левой полуленты.

Рассмотрим теперь вычисление предикатов на машинах Тьюринга. Машина T вычисляет предикат $P(\alpha)$ (α - слово в), если $T(\alpha)=\omega$, где $\omega = И$ *(истина),* когда $P(\alpha)$ истинно, и $\omega=Л$ *(ложь)*, когда $P(\alpha)$ ложно. Если же $P(\alpha)$ не определен, то машина T, как и при вычислении функций, не останавливается. При обычном вычислении предиката уничтожается α, что может оказаться неудобным, если после T должна работать другая машина. Поэтому вводится понятие вычисления с восстановлением: машина T вычисляет $P(\alpha)$ с восстановлением, если $T(\alpha)=\omega\alpha$. Если существует машина T, вычисляющая $P(\alpha)$, то существует и $\tilde{T}$, вычисляющая $P(\alpha)$ с восстановлением. Действительно, вычисление с восстановлением можно представить следующей последовательностью конфигураций:

$q_1\alpha \Rightarrow q_{n_1}\alpha * \alpha \Rightarrow \alpha * q_{n_2}\alpha \Rightarrow \alpha q_{n_3}\omega \Rightarrow q_{n_4}\omega\alpha$. Первая часть этой последовательности реализуется машиной $T_{коп}$, вторая – простым сдвигом головки до маркера *, третья – машиной T_R, вычисляющей $P(\alpha)$ на правой полуленте (одного копирования мало для восстановления, т.к. копию может испортить основная машина T, нужна машина, не заходящая влево от α), четвертая – переносом ω в крайнее левое положение. Машина T является композицией указанных четырех машин. Однако в конкретных случаях возможны и более простые способы восстановления α.

Пример 3.6. Машина с диаграммой рис.3.7 вычисляет предикат «α-*четное число»:* головка достигает конца числа в состоянии q_2, если число единиц четно, и в состоянии q_3, если число единиц нечетно, после чего она перемещается в состояние q_4 либо q_5 и печатает $И$ или $Л$ соответственно. Для того чтобы этот предикат вычислялся с восстановлением, достаточно в петлях q_4 и q_5 не стирать, а сохранять единицы, т.е. заменить команды $1\to\lambda L$ на команды $1\to L$.

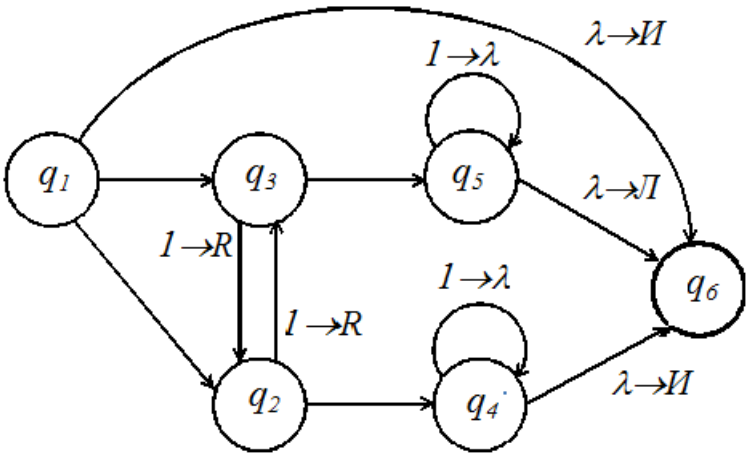


Рис. 3.7.

Так как машина Тьюринга с алфавитом $A^*=\{И,Л\}$ и командами $q_1И \to q_zЛЕ$ и $q_1Л \to q_zИЕ$ вычисляет отрицание логической переменной, то из вычислимости всюду определенного $P(\alpha)$ следует вычислимость $\overline{P(\alpha)}$.

Пусть функция $f(\alpha)$ задана описанием: «если $P(\alpha)$ истинно, то $f(\alpha)=g_1(\alpha)$, иначе $f(\alpha)=g_2(\alpha)$» (под «иначе» имеется в виду «если $P(\alpha)$ ложно»; если же $P(\alpha)$ не определен, то $f(\alpha)$ также не определена. Функция $f(\alpha)$ называется ***разветвлением*** (развилкой) *или* ***условным переходом*** к $g_1(\alpha)$ и $g_2(\alpha)$ по условию $P(\alpha)$.

Теорема 3.3 Если функции $g_1(\alpha)$, $g_2(\alpha)$ и предикат $P(\alpha)$ вычислимы по Тьюрингу, то развилка $g_1(\alpha)$ и $g_2(\alpha)$ по условию $P(\alpha)$ также вычислима.

Доказательство. Пусть T_1 – машина с состояниями $q_{11}, q_{12} \dots q_{1n_1}$ и системой команд Σ_1, вычисляющая g_1; T_2 – машина с состояниями $q_{21}, q_{22} \dots q_{2n_2}$ и системой команд Σ_2, вычисляющая g_2; T_P вычисляет с восстановлением $P(\alpha)$. Тогда машина T, вычисляющая разветвления g_1 и g_2 по P, есть композиция T_P и T_3, система команд Σ_3 которой имеет следующий вид:

$$\Sigma_3=\Sigma_1 \cup \Sigma_2 \cup \{q_{31}И \to q_{11}\lambda R,\ q_{31}Л \to q_{21}\lambda R,\ q_{1z} \to q_{2z}E\ \}.$$

Первые две из новых команд передают управление системам команд Σ_1 или Σ_2 в зависимости от значения предиката $P(\alpha)$. Третья команда введена для того, чтобы T_3 имела одно заключительное состояние q_{2z}. Отсутствие символа на ленте в этой команде означает, что она выполняется при любом символе.

Построенную по теореме 3.3 машину называют развилкой машин T_1 и T_2 по условию T_P и обозначают как $T_{раз}(T_1, T_2, T_P)$, а также изображают в виде блок-схемы, представленной на рис. 3.8.

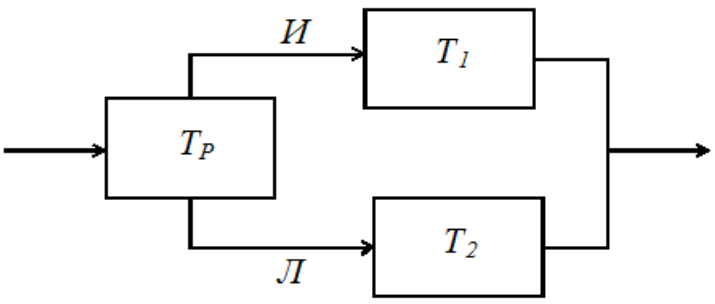


Рис. 3.8 Блок-схема развилки машин $T_{раз}(T_1, T_2, T_P)$

Пусть функция $f(\alpha)$ задана описанием: «пока истинно $P(\alpha)$, вычислять $g_1(\alpha)$, иначе $f(\alpha)=g_2(\alpha)$». Функция $f(\alpha)$ называется ***повторением*** или ***циклом*** от $g_1(\alpha)$ к $g_2(\alpha)$ по условию $P(\alpha)$.

Теорема 3.4. Если функции $g_1(\alpha)$, $g_2(\alpha)$ и предикат $P(\alpha)$ вычислимы по Тьюрингу, то цикл $g_1(\alpha)$ и $g_2(\alpha)$ по условию $P(\alpha)$ также вычислим.

Доказательство. Сохраняя обозначения из доказательства предыдущей теоремы, введем машину T, вычисляющую цикл g_1 и g_2 по P, - композиция $T_{раз}$ и машины T_3, система команд Σ_3 которой имеет следующий вид:

$\Sigma_3=\Sigma_1\cup\Sigma_2\cup\{q_{31}И\to q_{11}\lambda R,\ q_{31}Л\to q_{21}\lambda R,\ q_{1z}\to q_{p1}E\ \}$.

Первые две из новых команд передают управление системам команд Σ_1 или Σ_2 в зависимости от значений предиката $P(\alpha)$. Третья команда отождествляет начальную вершину q_{p1} диаграммы машины $T_{раз}$ с конечной вершиной q_{1z} диаграммы машины T_1 (т.е. после вычисления $g_1(\alpha)$ мы переходим к новой проверке условия $P(\alpha)$). Конечное состояние машины есть состояние q_{2z}.

Построенную по теореме 3.4 машину T называют циклом машин T_1 и T_2 по условию T_P и обозначают как $T_{цикл}(T_1, T_2, T_P)$, а также изображают в виде блок-схемы, представленной на рис. 3.9.

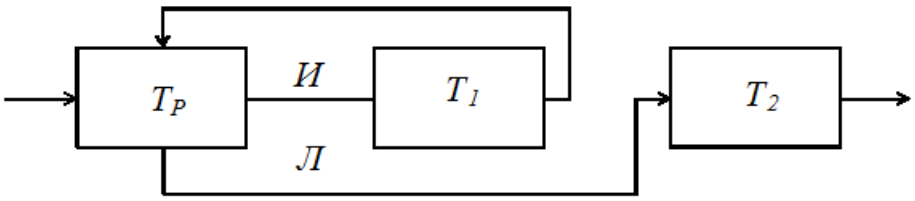


Рис. 3.9 Блок-схема цикла машин $T_{цикл}(T_1, T_2, T_P)$

Пример 3.7. В примере 3.4 было описано построение машины T_+ для сложения двух чисел в унитарном коде. На рис. 3.10 приведена диаграмма машины Тьюринга T_{++} для сложения n чисел (n=1, 2,…). Цикл из состояний q_1, q_2, q_3 – это немного модифицированная и «зацикленная» машина T_+, в которой заключительное состояние совмещено с начальным. Сумма, полученная в очередной итерации цикла, является первым слагаемым следующего цикла. Состояние q_4 реализует развилку. В нем проверяется условие – есть ли второе слагаемое. Если да (о чем говорит наличие маркера *), то происходит переход к следующему циклу; если нет (о чем говорит λ после единиц), то машина выходит из цикла.

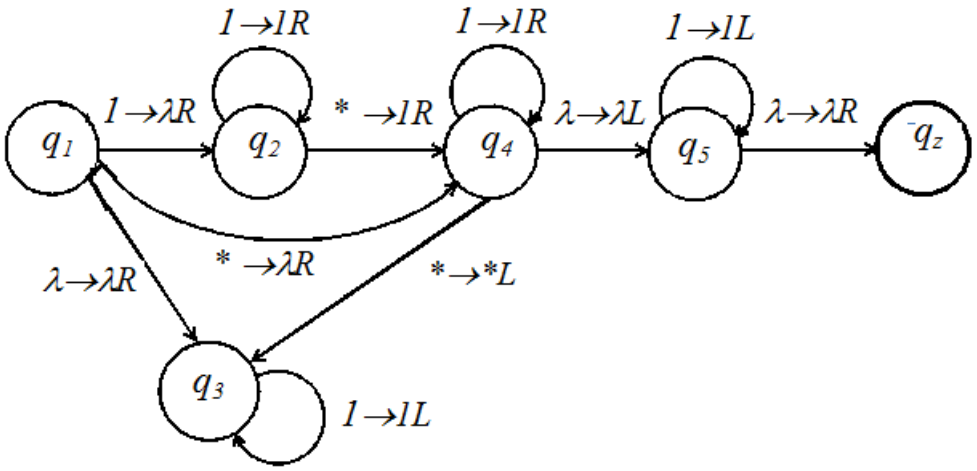


Рис. 3.10 Диаграмма переходов машины T_{++}

Благодаря вычислимости композиции, развилки и цикла, словесные описания и язык блок-схем можно сделать достаточно точным языком описания работы машин Тьюринга. Каждый блок – это множество состояний, в котором выделены начальное и заключительное состояния и система команд. Переход к блоку – это обязательно переход в его начальное состояние. Машина Тьюринга, описываемая блок-схемой, - это объединение состояний и команд, содержащихся во всех блоках. В частности, блоком может быть и одно состояние. Блоки, вычисляющие предикаты, обозначим буквой P.

На рис. 3.11 приведена блок-схема машины Тьюринга T_x, осуществляющей умножение двух чисел: $T_x(1^a*1^b)=1^{ab}$. Ее заключительной состояние – q_{03}.

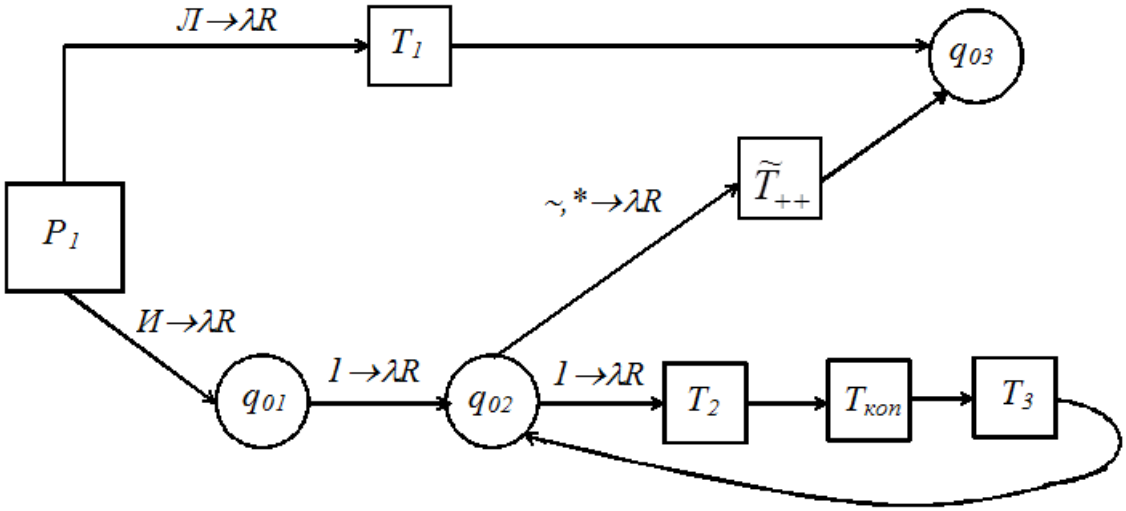


Рис. 3.11 Блок-схема машины T_x

Блоки реализуют следующие указания и вычисления:

P_1 – вычислить с восстановлением предикат «оба слагаемых больше нуля»;

T_1 – стереть все непустые символы справа;

T_2 – установить головку у ячейки, следующей (вправо) за маркером *; маркер * заменить на ~;

$T_{коп}$ – см. пример 3.4, в котором команду $q_1\lambda \to q_z R$ надо заменить на команду $q_1 \sim \to q_z R$;

$T_{коп}$ (a-1) копирует 1^b; после i-того цикла она останавливается в конфигурации $1^{a-i-1}(\sim 1^b)^{i-1} \sim q1^b * 1^b$;

T_3 – вернуть головку к крайнему слева непустому символу. После (a-1)-го цикла этим символом окажется ~ (или *, если a=1), и происходит выход из цикла и переход к $\tilde{T}_{++}$

$\tilde{T}_{++}$ работает как T_{++} (см. пример 3.7) с той разницей, что числа, которые она суммирует, разделены двумя видами маркеров: ~ и *; для этой цели в нее к командам с маркером * в левой части добавлены такие же команды для маркера ~.

Аналогично тому, как из машины T_+ была построена машина T_{++}, можно из T_x построить машину T_{xx}, которая перемножает несколько чисел. Другие важные примеры построения машин Тьюринга приведены, например, в [2,3].

3.3 Универсальная машина Тьюринга

Систему команд машины Тьюринга можно интерпретировать и как описание работы конкретного механизма, и как программу, т.е. совокупность предписаний, однозначно приводящих к результату.

При разборе примеров мы, как правило, использовали вторую интерпретацию, выступая в роли механизма, который способен воспроизвести работу любой машины Тьюринга. Эта способность связана с существованием алгоритма воспроизведения работы машины Тьюринга по заданной программе, то есть системе команд. Словесное описание такого алгоритма дать нетрудно, и его основное циклически повторяющееся действие состоит в следующем.

Для текущей конфигурации $\alpha_1 a_k q_i a_j \alpha_2$ найти в системе команд команду с левой частью $q_i a_j$. Если правая часть этой команды имеет вид $q_i' a_j' R$, то заменить в текущей конфигурации $a_k q_i a_j$ на $a_k a_j' q_j'$, если правая

часть имеет вид $q_i'a_j'L$, то заменить $a_k q_i a_j$ на $q_i'a_k a_j'$, если же правая часть имеет вид $q_i'a_j'E$, то заменить $a_k q_i a_j$ на $a_k q_i'a_j'$.

Как уже говорилось ранее, словесное описание алгоритма может быть неточным и нуждаться в формализации. Так как в качестве такой формализации понятия алгоритма сейчас обсуждается машина Тьюринга, то естественно поставить задачу построения машины Тьюринга, реализующей описанный алгоритм воспроизведения.

Для машин Тьюринга, вычисляющих функции от одной переменной, формулировка этой задачи такова: построить машину Тьюринга *U,* вычисляющую функцию от двух переменных и такую, что для любой машины T с системой команд Σ_T $U(\Sigma_T = \alpha) = T(\alpha)$, если $T(\alpha)$ определена (или останавливается), и $U(\Sigma_T, \alpha)$ не останавливается, если $T(\alpha)$ не останавливается.

Любую машину *U,* обладающую указанным свойством, называют ***универсальной машиной Тьюринга.***

Нетрудно обобщить эту формулировку для любого числа переменных.

Теорема 3.5. Универсальная машина Тьюринга существует.

Доказательство этой теоремы мы приводить не будем, но отметим, что имеется ее конструктивное доказательство, в котором описывается построение искомой универсальной машины Тьюринг и которое можно найти, например, в [3].

Следует отметить следующий важный момент. При построении машин Тьюринга в примерах с целью наглядности построений мы не жалели ни символов ленты, ни состояний. Нетрудно показать, а программист, привыкший к двоичному кодированию, легко в это поверит, что можно построить машину Тьюринга *U* всего с двумя символами на ленте. К. Шеннон построил универсальную машину всего с двумя состояниями, что совсем не очевидно. В то же время показано (Б. Боброу и М. Минский), что универсальная машина с двумя состояниями и двумя символами невозможна. Вообще в определенных пределах уменьшение числа символов *U* ведет к увеличению числа состояний, и наоборот.

Существование универсальной машины Тьюринга означает, что систему команд Σ_T любой машины *T* можно интерпретировать двояко, а именно как:

- описание работы конкретного устройства машины *T;*
- программу для универсальной машины *U.*

Для современного информатика это обстоятельство вполне естественно: любой алгоритм может быть реализован либо аппаратно (построением соответствующей схемы), либо программно (написание программы для универсального компьютера). Однако важно сознавать, что сама идея универсального алгоритмического устройства совершенно не связана с развитием современных технических средств его реализации (лампы, полупроводники, БИС, СБИС и т. д.) и является не техническим, а математическим фактом, который описывается в абстрактных математических терминах, не зависящих от технических средств, и к тому же опирающимся на крайне малое количество весьма элементарных исходных понятий. Характерно, что основополагающие работы по теории алгоритмов (в частности, работы Тьюринга) появились в 30-х годах XX века, до создания первых ЭВМ.

Указанная двоякая интерпретация сохраняет на абстрактном уровне основные плюсы и минусы двух вариантов инженерной реализации. Конкретная машина ***T*** работает гораздо быстрее; кроме того, управляющее устройство машины ***U*** довольно громоздко, т. е. велико число состояний и команд. Однако его сложность постоянна, и, будучи раз построено, оно годится для реализации сколь угодно больших алгоритмов, понадобится лишь большой объем ленты, которую естественно считать более дешевой и более просто устроенной, чем управляющее устройство. Кроме того, при смене алгоритма не понадобится строить новых устройств, нужно лишь написать новую программу.

3.4 Тезис Тьюринга

До сих пор нам удавалось для всех процедур, претендующих на алгоритмичность, то есть конструктивных процедур, строить реализующие их машины Тьюринга. Будет ли это удаваться всегда? Утвердительный ответ на этот вопрос содержится в тезисе Тьюринга, который формулируется так: ***всякий алгоритм может быть реализован машиной Тьюринга.***

Доказать тезис Тьюринга нельзя, поскольку само понятие алгоритма (или эффективной процедуры) является неточным. Это не теорема и не постулат математической теории, а утверждение, которое связывает теорию с теми объектами, для описания которых она создана. По своему характеру тезис Тьюринга напоминает гипотезы физики об адекватности математических моделей физическим явлениям и процессам. Подтверждением тезиса Тьюринга является, во-первых, математическая

практика, а во-вторых, то обстоятельство, что описание алгоритма в терминах любой другой известной алгоритмической модели может быть сведено к его описанию в виде машины Тьюринга.

Тезис Тьюринга позволяет, с одной стороны, заменить неточные утверждения о существовании эффективных процедур (алгоритмов) точными утверждениями о существовании машин Тьюринга, а с другой сторон утверждения о несуществовании машин Тьюринга истолковывать как утверждения о несуществовании алгоритмов вообще. Однако не следует понимать тезис Тьюринга в том смысле, что вся теория алгоритмов может быть сведена к теории машин Тьюринга.

3.5 Проблема остановки

В числе общих требований, предъявляемых к алгоритмам, есть и требование результативности. Его можно сформулировать так: нужно построить алгоритм *B,* такой, что для любого алгоритма *A B(A,* α) = *И*, если *A(*α*)* дает результат, и *B(A,* а) = *Л*, если *A(*α*)* не дает результат.

Эта задача называется **проблемой остановки.**

Имеет место следующая теорема, которую мы примем без доказательства (доказательство можно найти в [2])

Теорема 3.6. Не существует машины Тьюринга T_o, решающей проблему остановки для произвольной машины Тьюринга *T.*

Т.е. не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий, т.е. невозможностью предсказать, что для любых исходных данных решение будет получено за конечное количество шагов. Тем не менее, можно попытаться сформулировать причины, ведущие к алгоритмической неразрешимости, эти причины достаточно условны, так как все они сводимы к проблеме останова, однако такой подход позволяет более глубоко понять природу алгоритмической неразрешимости:

а) отсутствие общего метода решения задачи;

б) информационная неопределенность задачи;

в) логическая неразрешимость (в смысле теоремы Гёделя о неполноте).

Рассмотрим несколько примеров на очевидность проблемы остановки.

Пример 3.8. Вычисление совершенных чисел;

Совершенные числа – это числа, которые равны сумме своих делителей, например: 28 = 1+2+4+7+14.

Определим функцию *S(n)* = *n*-ое по счёту совершенное число и поставим задачу вычисления *S(n)* по произвольно заданному *n*. Общего метода вычисления совершенных чисел нет. Также, неизвестно, что множество совершенных чисел конечно или счетно. Поэтому наш алгоритм должен перебирать все числа подряд, проверяя их на совершенность. Отсутствие общего метода решения не позволяет ответить на вопрос об остановке алгоритма. Если мы проверили *K* чисел при поиске *n*-ого совершенного числа – означает ли это, что его вообще не существует?

Пример 3.9 Найти распределение девяток в записи числа π.

Определим функцию *f(n) = i*, где *n* – количество девяток подряд в десятичной записи числа π, а *i* – номер самой левой девятки из n девяток подряд: π=3,14159265… *f(1)* = 5.

Задача состоит в вычислении функции *f(n)* для произвольно заданного *n*. Поскольку число π является иррациональным и трансцендентным, то мы не знаем никакой информации о распределении девяток (равно как и любых других цифр) в десятичной записи числа π. Вычисление *f(n)* связано с вычислением последующих цифр в разложении π, до тех пор, пока мы не обнаружим *n* девяток подряд, однако у нас нет общего метода вычисления *f(n)*, поэтому для некоторых *n* вычисления могут продолжаться бесконечно – мы даже не знаем в принципе (по природе числа π) существует ли решение для всех *n*.

Неразрешимость проблемы остановки можно интерпретировать как несуществование общего алгоритма для отладки программ, точнее, алгоритма, который по тексту любой программы и ее входным данным определял бы, зациклится ли программа на этих данных или нет. Если учесть сделанное ранее замечание, такая интерпретация не противоречит тому эмпирическому факту, что большинство программ в конце концов удается отладить, т.е. установить причину зацикливания, найти и устранить его причину. При этом решающую роль играют искусство и опыт программиста.

3.6 Другие модели абстрактных машин.

В дальнейшем появились другие модели абстрактных вычислительных машин. Цель этих моделей — приблизиться к возможностям обычных компьютеров путем расширения возможностей доступа к памяти. Кратко опишем две типичные модели таких машин.

3.6.1 Многоленточные машины Тьюринга

Для некоторых задач составление алгоритмов работы обычной машины Тьюринга представляет значительные трудности, связанные с необходимостью где-то хранить результаты промежуточных вычислений, или производить посимвольное сравнение нескольких групп элементов. В некоторых случаях наличие дополнительной (рабочей) ленты или размещение входных слов на нескольких лентах одновременно позволяет получить более лаконичное решение. Это можно реализовать на многоленточной машине Тьюринга, схематическое изображение которой приведено на рис. 3.12.

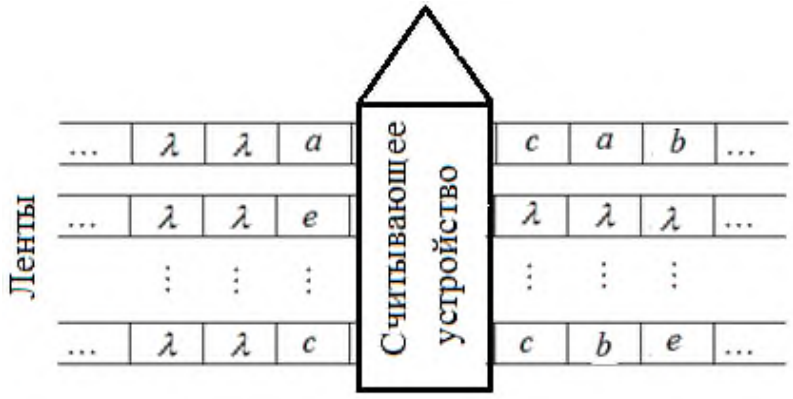


Рис. 3.12

Многоленточная машина для каждой ленты может иметь свой алфавит. Ленты в машине движутся независимо друг от друга. Однако состояние для всех лент машины единое, по сути это состояние управляющего механизма. Собственно ранее, при рассмотрении одноленточных машин, было принято считать что лента неподвижна, а головка перемещается в заданном направлении. Но для рассмотрения многоленточных машин это не вполне удобно, потому что ленты являются независимыми, а наглядно представить перемещение единого управляющего механизма по разным направлениям весьма проблематично. Итак, считаем что управляющий механизм неподвижен, а ленты могут свободно перемещаться вправо, влево или оставаться на месте независимо друг от друга. Типичная команда такой машины имеет вид

$$q_i a_{j1} \cdots a_{jn} \to q_i' a'_{j1} \cdots a'_{jn} d_{j1} \ldots d_{jn}.$$

Многие вычисления на такой машине приобретают более привычный вид, чем на одноленточной машине Тьюринга. Например, на четырехленточной машине легко организовать обычный алгоритм сложения столбиком для чисел в позиционной системе счисления: на первых двух лентах записываются слагаемые, на третьей ленте — перенос, на четвертой — результат. Команды реализуют строки таблицы сложения для всех пар цифр системы счисления, например, для десятичной системы 9 + 8= 17 (1 на третью ленту, 7 в результат) и т. д.

Однако как это ни покажется парадоксальным, вычислительная способность многоленточных машин совершенно не превосходит их одноленточные аналоги. Иными словами, задачи, которые можно решить на многоленточной машине с произвольным количеством лент, всегда решаются и при помощи одноленточной машины.

3.6.2 Недетерминированные машины Тьюринга.

По причинам, которые вскоре будут понятны, ***недетерминированные машины*** Тьюринга являются ключевым понятием в теории *NP*-полных задач. Недетерминированная машина Тьюринга отличается от обычной (детерминированной) машины Тьюринга тем, что может иметь более одного перехода от текущей конфигурации к следующей, или можно определить это так: недетерминированная машина Тьюринга – это машина Тьюринга, в которой на каждом этапе вычислений существуют альтернативы. Схематично это можно представить как показано на рис.3.13.

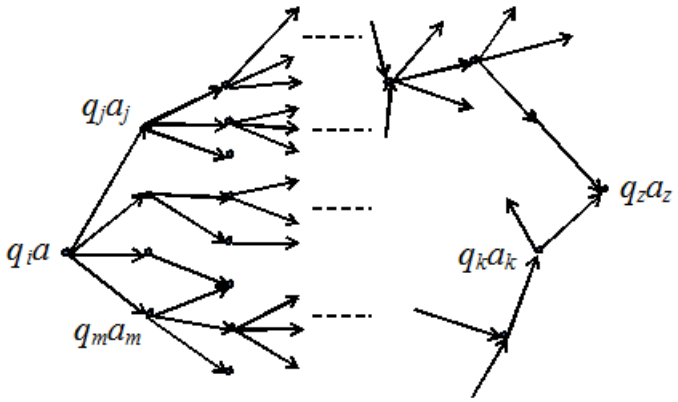


Рис.3.13

Недетерминированная машина допускает слово α, если существует хотя бы одна цепочка конфигураций, ведущая от начальной конфигурации в заключительную. Существование других последовательностей конфигураций, не ведущих в заключительное (допускающее) состояние не

имеет значение. Работу недетерминированной машины на входе α можно представить в виде дерева, где каждый путь из корня α в лист представляет некоторую последовательность возможных шагов машины. Если $s\alpha$ кратчайшая последовательность возможных шагов работы машины, которая оканчивается допускающей конфигурацией, время, затраченное машиной на обработку входа α, конечно. Если на входе α никакая последовательность не приводит к допускающей конфигурации, то время, затраченное на обработку α не определено. Считается, что недетерминированная машина Тьюринга на входе α параллельно выполняет все возможные последовательности шагов, пока не достигнет допускающего состояния или окажется, что ее программа не применима к полученной конфигурации.

Остается открытым вопрос, существуют ли языки, допускаемые недетерминированной машиной Тьюринга с данной временной и емкостной сложностью и не допускаемые никакой детерминированной машиной с той же сложностью.

Недетерминированные машины Тьюринга допускают те же языки, что и детерминированные. Однако, надо заметить, что последним приходится за это расплачиваться сильным увеличением временной сложности.

3.5.3Машины с произвольным доступом к памяти.

Существует несколько моделей таких машин. Все они реализуют доступ к памяти по адресу.

Машина с произвольным доступом к памяти (или, иначе, равнодоступная адресная машина — сокращенно РАМ) моделирует вычислительную машину с одним сумматором, в которой команды программы не могут изменять сами себя.

РАМ состоит из входной ленты, с которой она может только считывать, выходной ленты, на которую она может только записывать, и памяти (рис. 3.14). Входная лента представляет собой последовательность ячеек, каждая из которых может содержать целое число (возможно, отрицательное). Всякий раз, когда символ считывается с входной ленты, ее читающая головка сдвигается на одну ячейку вправо. Выход представляет собой ленту, на которую машина может только записывать. Она разбита на ячейки, которые вначале все пусты. При выполнении команды записи в той ячейке выходной ленты, которую в текущий момент обозревает ее головка, печатается целое число и головка затем сдвигается на одну ячейку вправо. Как только выходной символ записан, его уже нельзя изменить.

Память состоит из последовательности регистров R_0,R_1 R_2, … каждый из которых способен хранить произвольное целое число. На число регистров, которые можно использовать, верхняя граница не устанавливается. Такая идеализация допустима в случаях, когда:

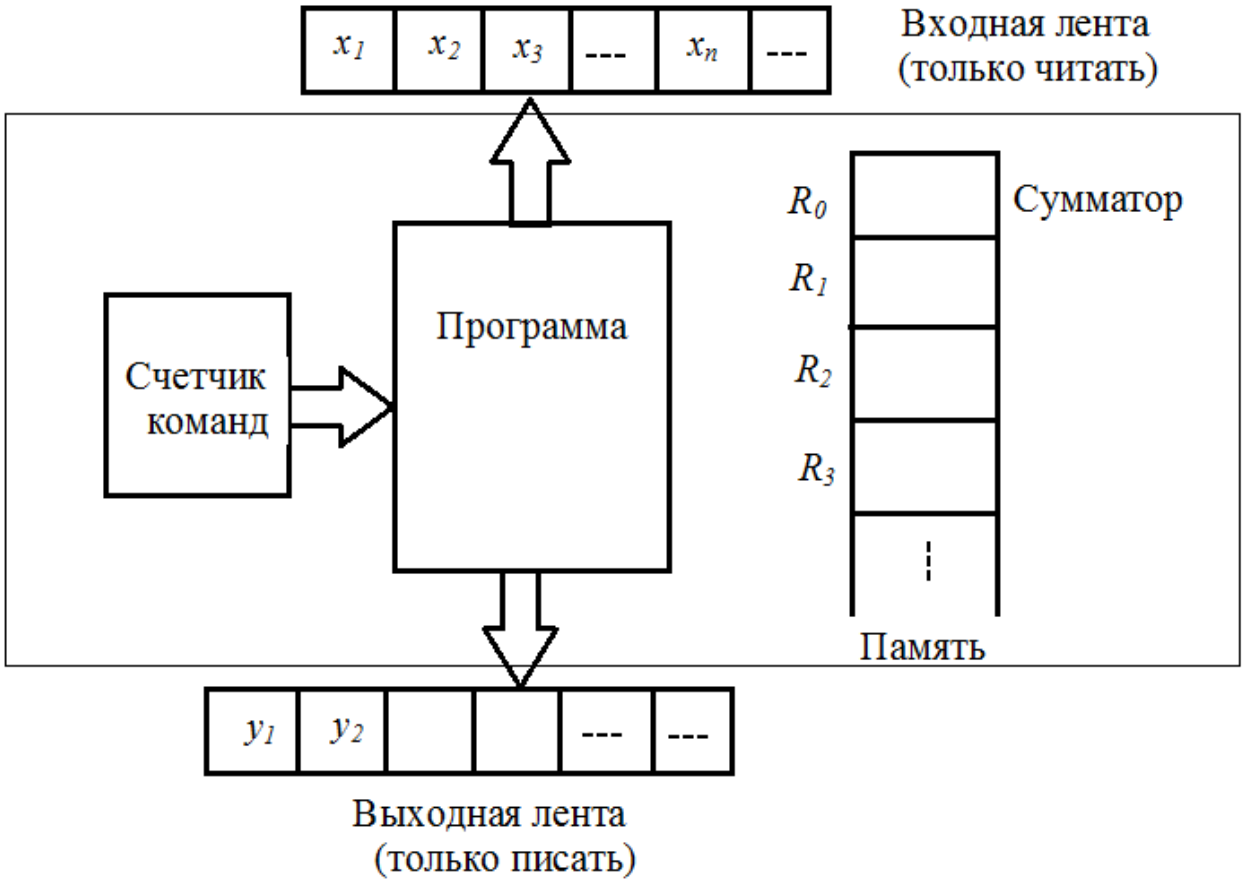


Рис. 3.14 Машина с произвольным доступом к памяти.

1) размер задачи достаточно мал, чтобы она поместилась в основную память вычислительной машины,

2) целые числа, участвующие в вычислении, достаточно малы, чтобы их можно было помещать в одну ячейку.

Программа для РАМ (или РАМ-программа) не записывается в память. Поэтому программа не изменяет сама себя. Программа является, в сущности, последовательностью (возможно) помеченных команд. Точный тип команд, допустимых в программе, не слишком важен, и они напоминают те, которые обычно встречаются в реальных вычислительных машинах. Предполагаем, что имеются арифметические команды, команды ввода-вывода, косвенная адресация (например, для индексации массивов) и команды разветвления. Все вычисления производятся в первом регистре R_0, называемом сумматором, который, как и всякий другой регистр памяти, может хранить произвольное целое число. Пример набора команд для РАМ показан на рис. 3.15. Каждая команда состоит из двух частей — кода операции и адреса.

Код операции		Адрес
1.	LOAD	Операнд
2.	STORE	Операнд
3.	ADD	Операнд
4.	SUB	Операнд
5.	MULT	Операнд
6.	DIV	Операнд
7.	READ	Операнд
8.	WRITR	Операнд
9	JUMP	Метка
10.	JGTZ	Метка
11.	JZERO	Метка
12.	HALT	

Рис. 3.15

В принципе можно было бы добавить к этому набору любые другие команды, встречающиеся в реальных вычислительных машинах, такие, как логические или литерные операции, и при этом порядок сложности задач не изменится.

Операнд может быть одного из следующих типов:

1) $=i$ означает само целое число i и называется литералом;
2) i - содержимое регистра i (i должно быть неотрицательным);
3) $*i$ означает косвенную адресацию, т. е. значением операнда служит содержимое регистра j, где j целое число, находящееся в регистре i; если j<0, машина останавливается.

Эти команды хорошо знакомы всякому, кто программировал на языке ассемблера. Можно определить значение программы P с помощью двух объектов: отображения C из множества неотрицательных целых чисел в множество целых чисел и «счетчика команд», который определяет очередную выполняемую команду. Функция C есть отображение памяти, а именно $C(i)$- целое число, содержащееся в регистре i (содержимое регистра i).

Вначале $C(i)=0$ для всех $i \geq 0$, счетчик команд установлен на первую команду в P, а выходная лента пуста. После выполнения k-й команды из P счетчик команд автоматически переходит на $k+1$ (т. е. на очередную команду), если k-я команда не была командой вида JUMP, HALT, JGTZ или JZERO.

Для описания действий команд зададим значение $V(a)$ операнда a:

$V(=i) = i$ - само значение операнда i;
$V(i) = C(i)$ - адрес - целое число, содержащееся в регистре i;
$V(*i) = C(C(i))$ - содержимое регистра $C(i)$.

№	Код операции	Действие	Команда	Описание действия
1.	LOAD	Загрузка (вызов в сумматор)	L*a*	$C(R_0) \leftarrow V(a)$ В сумматор R_0 загружается *a*
2.	STORE	Поместить в память	ST*i*	$C(i) \leftarrow C(R_0)$ В регистр с адресом *i* поместить содержимое сумматора
			ST**i*	$C(C(i)) \leftarrow C(R_0)$ В регистр с адресом C(*i*) поместить содержимое сумматора
3.	ADD	Сложение	A*a*	$C(R_0) \leftarrow C(R_0)+V(a)$
4.	SUB	Вычитание	S*a*	$C(R_0) \leftarrow C(R_0)-V(a)$
5.	MULT	Умножение	M*a*	$C(R_0) \leftarrow C(R_0)*V(a)$
6.	DIV	Деление	D*a*	$C(R_0) \leftarrow [C(R_0)/V(a)]$ – целая часть от деления.
7.	READ	Ввод	R*i*	$C(i) \leftarrow$ Ввод очередного символа
			R**i*	$C(C(i)) \leftarrow$Ввод очередного символа.
8.	WRITE	Вывод	W*a*	$V(a)$ Значение операнда *a* печатается в обозреваемой ячейке выходной ленты, после чего головка сдвигается на одну ячейку вправо.
9.	JUMP	Безусловный переход	J*b*	Счетчик команд устанавливается на команду с меткой *b*.
10	JGTZ	Условный переход	JG*b*	Если $C(R_0)>0$, то счетчик команд устанавливается на команду с меткой *b*, в противном случае на следующую команду.
11	JZERO	Условный переход при равенстве 0	JZ*b*	Если $C(R_0)=0$, то счетчик команд устанавливается на команду с меткой *b*, в противном случае на следующую команду.
12	HALT	Останов	H	Работа программы прекращается.

В общем случае РАМ-программа задаёт отображение множества символов, записанное на входной ленте, во множество выходных символов, т.е. $P : RX \rightarrow RY$. Так как при некоторых входных данных программа *P* может не останавливаться, то это отображение является частичным (т.е. для некоторых входов оно может быть не определено). Это отображение может быть использовано для преобразования числовых и символьных данных.

В первом случае его можно интерпретировать как функцию, заданную на множестве входных числовых данных, и имеющую значения на множестве числовых данных, которые записываются на ленту, т.е.

$$y = f(x_1, x_2, \ldots, x_n).$$

Во втором случае программа P является распознавателем некоторого языка. (Языком, допускаемым программой P, называется множество всех входных цепочек (слов), которые воспринимаются и обрабатываются программой.)

Из вышеприведенного примера модели машины с произвольным доступом к памяти видно, что она значительно расширяет возможности машины Тьюринга и очень существенно приближена к реальным вычислительным устройствам.

Еще одной разновидностью модели машины с произвольным доступом к памяти, расширяющей возможности машины Тьюринга, является **машина с неограниченными регистрами** (МНР).

МНР состоит из памяти данных и программной памяти. Ячейки памяти данных называют регистрами и обозначают R_0, R_1, R_2 и т.д. до бесконечности. В каждом регистре может содержаться любое целое неотрицательное число. Число, содержащееся в R_i, или значение регистра R_i обозначим как r_i. В каждый момент времени только конечное число регистров содержит какие-то числа, т.е. память данных является не бесконечной, а потенциально бесконечной. МНР работает по программе. Программой называется пронумерованная конечная последовательность команд, хранящаяся в программной памяти. Программная память МНР также является потенциально бесконечной. Условно схему МНР можно представить как на рис. 3.16.

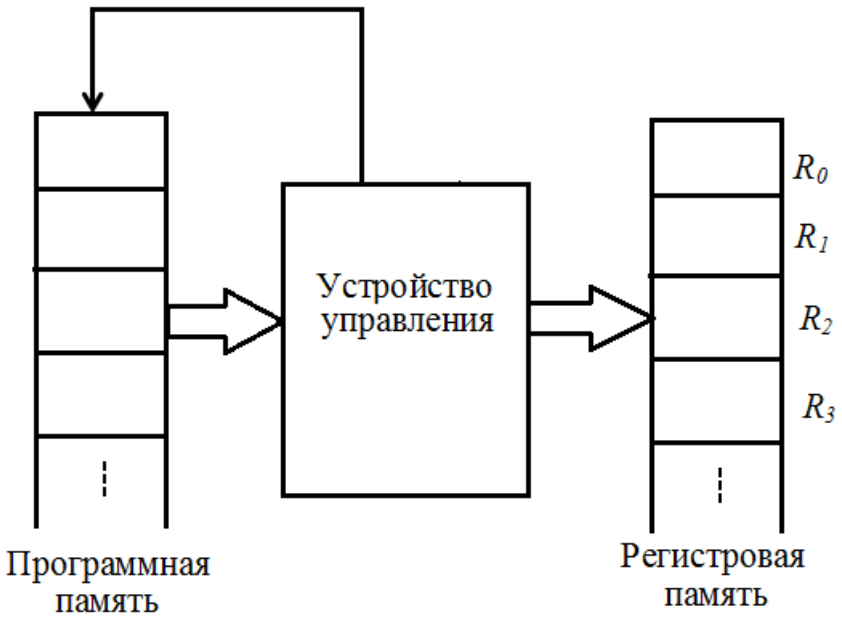


Рис. 3.16 Машина с неограниченными регистрами

Команды программы пронумерованы.

Команды (для любых *m, n, q):*

1) обнуление $Z(n): r_n := 0$ (записать 0 в регистр R_n);

2) прибавление единицы $S(n): r_n := r_n + 1$ (прибавить единицу к содержимому регистра R_n);

3) переадресация $T(m, n): r_n := r_m$ (записать в регистр R_n содержимое регистра R_m);

4) условный переход $J(m, n, q):$ если $r_n = r_m$, то выполнять *q-ю* команду программы; иначе выполнять следующую команду;

5) останов STOP (машина останавливается).

Команды обнуления и прибавления единицы называются арифметическими командами.

В МНР есть особый регистр, который называют счетчиком адресов команд (САК). При запуске программы на выполнение в САК заносится 1.

Выполнение каждой команды МНР состоит из четырех этапов:

1) читается адрес из САК;
2) запоминается команда из памяти по этому адресу;
3) САК увеличивается на единицу;
4) выполняется запомненная команда.

МНР выполняет каждую команду программы в 4 этапа, пока не встретит команду STOP. После каждой команды типа 1-3 выполняется команда со следующим номером.

Работа на МНР состоит из следующих этапов:

1) Заполняются регистры памяти данных какими-то числами, т.е. задается так называемая начальная конфигурация.
2) Заполняются командами ячейки программной памяти.
3) В САК заносится единица.
4) Осуществляется пуск МНР.

Работа МНР продолжится в соответствии с заданной программой до тех пор, пока не встретит команду STOP. При этом полученное по окончании работы машины содержимое регистров памяти данных называется конечной конфигурацией.

Пример 3.10 Составить программу МНР сложения двух чисел:

$$f(x,y)=x+y.$$

В регистр R_1 занесем значение x, а в регистр R_2 значение y. Значение суммы будем получать в R_1. В R_3 занесем 0. Программа будет иметь вид:

1:Z(3);
2:J(2,3,6);

3:S(1);
4:S(3);
5:J(2,2,2);
6:STOP.

Следует отметить, что машины с произвольным доступом к памяти являются простейшими моделями реализации алгоритмов с возможностью их распараллеливания.

3.5.4 Имитация машины Тьюринга на компьютере и компьютера на машине Тьюринга.

Как известно, к основным компонентам вычислительной машины относятся оперативная память и процессор. Программы и данные, представленные в двоичном алфавите, помещаются в память. При выполнении программы отдельные ее команды и нужные данные извлекаются из памяти в процессор и наоборот – значения, получаемые при выполнении команд, записываются в ячейки памяти.

Память состоит из некоторого числа запоминающих ячеек (регистров), предназначенных для промежуточного хранения значений операндов и для хранения другой информации, необходимой для выполнения команд, регистров для управления запоминающими ячейками, адресов ячеек и полей самих ячеек.

Процессор состоит из устройства управления (УУ) и арифметического устройство (АУ). Устройство управления содержит счетчик тактов, команд и т.д., вырабатывает управляющие сигналы для выполнения команд, передачи данных и т.д. Процессор содержит регистры операндов, линии связи и линии задержки для непосредственной реализации процессов вычислений.

Наряду с процессором и памятью компьютеру необходимы еще устройства ввода/вывода.

Имитация машины Тьюринга на компьютере.

Пусть T - машина Тьюринга, одним из составляющих которой является ее конечное управление. Поскольку T имеет конечное число состояний и конечное число правил перехода, программа компьютера может закодировать состояния в виде цепочек символов, как и символы ее внешнего алфавита, и использовать таблицу переходов машины T для преобразования цепочек. Бесконечную ленту машины Тьюринга можно имитировать сменными дисками, размещаемыми в двух магазинах, соответственно для данных, расположенных слева и справа от

считывающей головки на ленте. Чем дальше в магазине расположены данные, тем дальше они от головки на ленте.

Для имитации компьютера на машине Тьюринга существенны две вещи:

- существуют ли инструкции, выполняемые компьютером, и недоступные для машины Тьюринга;
- работает ли компьютер быстрее машины Тьюринга, т.е. более, чем полиномиальная зависимость разделяет время работы компьютера и машины Тьюринга при решении какой-то проблемы.

Неформальная модель реального компьютера:

- память, состоящая из последовательности слов и их адресов. В качестве адресов будут использоваться натуральные числа 0,1, …;
- программа компьютера, записанная в слова памяти, каждое из которых представляет простую инструкцию. Допускается “непрямая адресация” по указателям;
- каждая инструкция использует конечное число слов и изменяет значение не более одного слова;
- имеются слова памяти с быстрым доступом (регистры), но скорость доступа к различным словам влияет лишь на константный сомножитель, что не искажает полиномиальную зависимость.

  Возможная конструкция машины Тьюринга для имитации компьютера представлена на рис.3.12

Рис. 3.12

Машина имеет несколько лент. Первая лента представляет всю память компьютера – адреса и значения (в двоичной системе). Адреса заканчиваются маркером *, значения – маркером #. Начало и конец записей 1-й ленты обозначаются маркером *$*. Вторая лента – "счетчик инструкций", содержит одно двоичное целое, представляющее одну из позиций считывающей головки на первой ленте, адрес инструкции, которая должна быть выполнена следующей. Третья лента содержит адрес и значение по нему после того, как этот адрес устанавливается на первой ленте. Для выполнения инструкции машина Тьюринга должна найти значение по одному или нескольким адресам памяти, где хранятся данные, участвующие в вычислении. Нужный адрес копируется на ленту 3 и сравнивается с адресами на ленте 1 до совпадения. Значение по этому адресу копируется на третью ленту и перемещается на нужное место, как правило, по одному из начальных адресов, представляющих регистры компьютера. Четвертая лента имитирует входной файл. Пятая лента - рабочая память, служит для выполнения вычислений.

4. Машина Поста

4.1 Математическая модель Э. Поста

Еще одной абстрактной моделью алгоритма является машина Поста, предложенная Эмилем Постом в 1936 году независимо от машины Тьюринга. (Сообщение о машине Поста было опубликовано на несколько месяцев позднее публикации о машине Тьюринга). Существенного отличия машины Поста от машины Тьюринга нет, обе машины алгоритмически «эквивалентны» и обе разработаны для формализации понятия алгоритма и решения задач об алгоритмической разрешимости, то есть, демонстрации алгоритмического решения задач в форме последовательности команд для машины Поста.

Э. Пост рассматривает общую проблему, состоящую из множества конкретных проблем, при этом решение общей проблемы это такое решение, которое доставляет ответ для каждой конкретной проблемы. Например, решение уравнения $3 \cdot x+9=0$ – это одна из конкретных проблем, а решение уравнения $a \cdot x+b=0$ – это общая проблема, тем самым алгоритм (сам термин «алгоритм» не используется Постом) должен быть универсальным, т.е. должен быть соотнесен с общей проблемой.

Основные понятия алгоритмического формализма Поста – это пространство символов (язык L) в котором задаётся конкретная проблема и

получается ответ, и набор инструкций, т.е. операций в пространстве символов, задающих как сами операции, так и порядок выполнения инструкций.

Постовское пространство символов – это бесконечная лента ячеек (ящиков):

	v			v	v	v		v

Каждый ящик или ячейка могут быть помечены или не помечены. Конкретная проблема задается «внешней силой» (термин Поста) пометкой конечного количества ячеек, при этом, очевидно, что любая конфигурация начинается и заканчивается помеченной ячейкой. После применения к конкретной проблеме некоторого набора инструкций решение представляется так же в виде набора помеченных и непомеченных ячеек, распознаваемое той же внешней силой.

Пост предложил набор инструкций (элементарных операций), которые выполняет «работник», отметим, что в 1936 году не было еще ни одной электронной вычислительной машины. Этот набор инструкций является, очевидно, минимальным набором битовых операций:

1. пометить ячейку, если она пуста;
2. стереть метку, если она есть;
3. переместиться влево на 1 ячейку;
4. переместиться вправо на 1 ячейку;
5. определить помечена ячейка или нет, и по результату перейти на одну из двух указанных инструкций;
6. остановиться.

Ниже на рис. 4.1 приведены команды для машины Поста и результат ее реализации. Информация о заполненных метками клетках ленты характеризует состояние ленты, которое может меняться в процессе работы машины. В каждый момент времени головка (⇳) находится над одной из клеток ленты и, как говорят, обозревает ее. Информация о местоположения головки вместе с состоянием ленты характеризует состояние машины Поста.

Ситуации, в которых головка должна наносить метку там, где она уже имеется, или, наоборот, стирать метку там, где ее нет, являются аварийными (недопустимыми).

Программой для машины Поста называют непустой список команд, такой что:

1) на *n*-м месте команда с номером *n*;

2) номер *m* каждой команды совпадает с номером какой-либо команды списка.

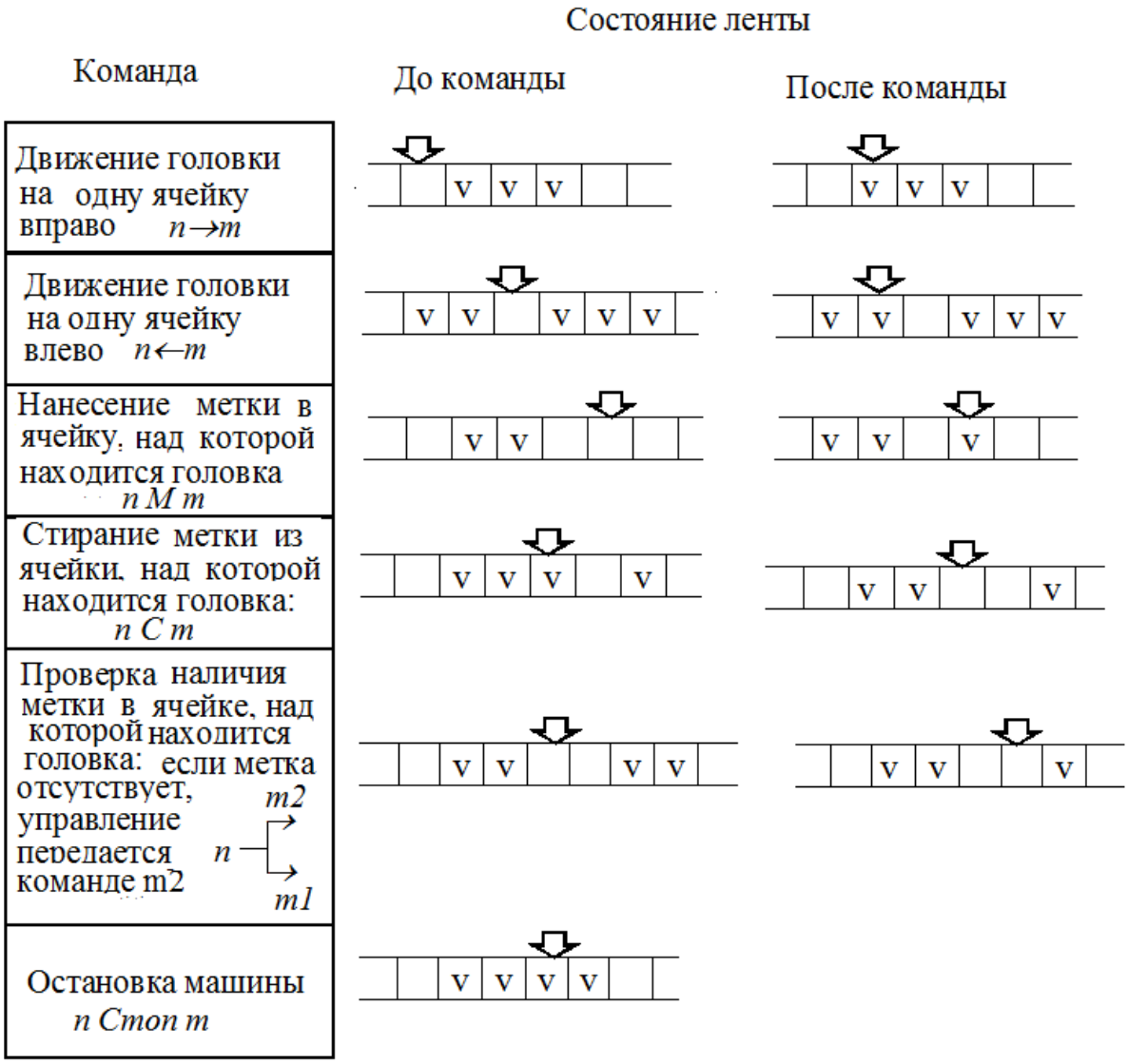


Рис. 4.1 Команды машины Поста

С точки зрения свойств алгоритмов, изучаемых с помощью машины Поста, наибольший интерес представляют причины останова машины при выполнении программы:

1) останов по команде «стоп»; такой останов называется результативным и указывает на корректность алгоритма (программы);
2) останов при выполнении недопустимой команды; в этом случае останов называется безрезультативным;
3) машина не останавливается никогда; в этом и в предыдущем случае мы имеем дело с некорректным алгоритмом (программой).

Под начальным состоянием понимается состояние, при котором головка находится против пустой ячейки левее самой левой метки на ленте.

Рассмотрим реализацию некоторых типичных элементов программ машины Поста.

Пример 4.1 Пусть задано исходное состояние головки и требуется на пустой ленте написать две метки: одну в секцию под головкой, вторую справа от нее. Это можно сделать по следующей программе (справа от команды показан результат ее выполнения):

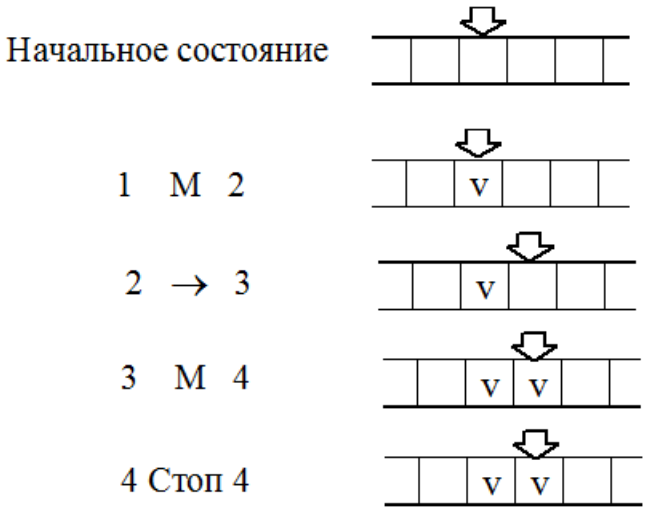
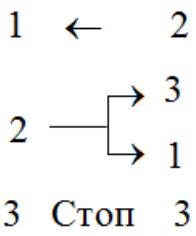


Рис. 4.2 Программа машины Поста для примера 4.1

Пример 4.2 На ленте имеется запись из нескольких меток подряд и головка находится над самой крайней меткой справа. Требуется перевести головку влево до первой пустой позиции.

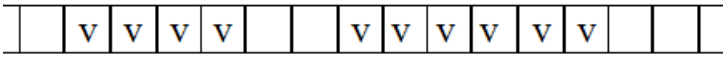
Программа будет иметь следующий вид:



Т.е. здесь мы воспользовались командой условного перехода для организации циклического перехода. Команда условного перехода является одним из основных средств организации циклических процессов, например, для нахождения первой метки справа (или слева) от головки, расположенной над пустой клеткой; нахождение слева (или справа) от головки пустой клетки, если она расположена над меткой и т.д.

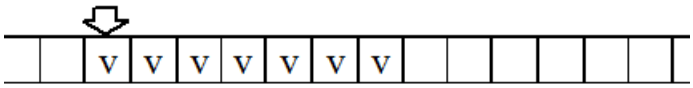
Теперь рассмотрим представление чисел на ленте машины Поста и выполнении операций над ними.

Число *k* представляется на ленте машины Поста идущими подряд *k + 1* метками (одна метка означает число «0»). Между двумя числами делается интервал как минимум из одной пустой секции на ленте. Например, запись чисел 3 и 5 на ленте машины Поста будет выглядеть так:



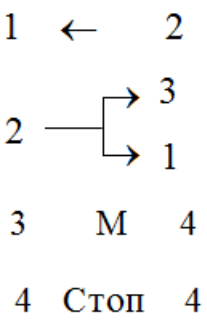
Следует обратить внимание, что используемая в машине Поста система записи чисел является непозиционной.

Составим программу для прибавления к произвольному числу единицы. Предположим, что на ленте записано только одно число и головка находится над одной из ячеек, в которой находится метка, принадлежащая этому числу:

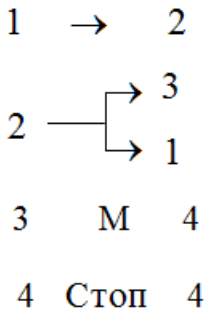


Для решения задачи можно переместить головку влево (или вправо) до первой пустой клетки, а затем нанести метку.

Программы добавления метки к числу имеющихся меток слева и слева будут иметь вид:



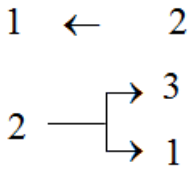
(слева)



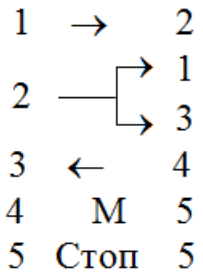
(справа)

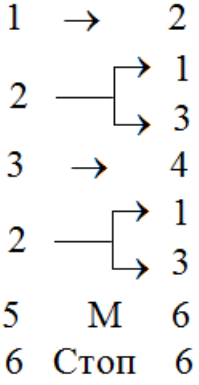
Как видно, отличие только в направлении движения головки в первой команде.

Предположим, что головка расположена на расстоянии нескольких ячеек слева от числа, к которому нужно прибавить единицу. В этом случае программа усложняется. Появится «блок поиска числа» - две команды, приводящие головку в состояние, рассмотренное в предыдущем примере:



И тогда программы, добавляющие единицу слева и справа, будут соответственно иметь вид:

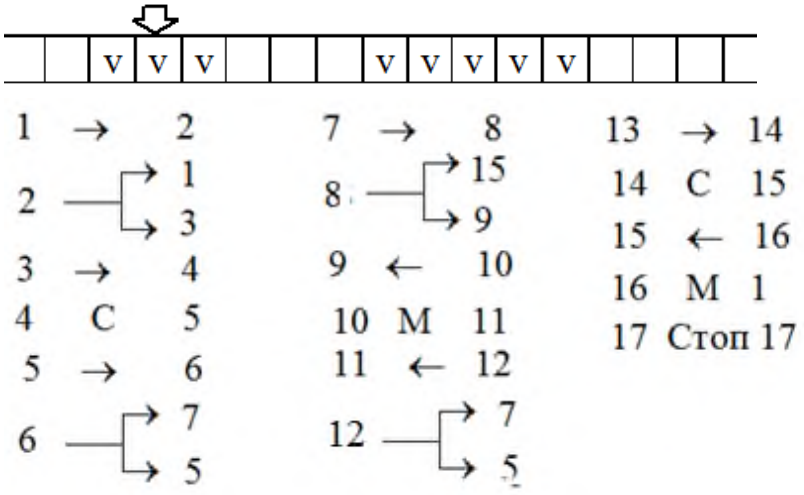




(слева) (справа)

В первом случае не нужно перемещать головку к крайней левой метке числа

Рассмотрим программу для сложения целых неотрицательных чисел *a* и *b* на машине Поста, когда головка находится над числом *a*, а число *b* находится правее числа *a* на некоторое число ячеек. Эта программа реализует следующий алгоритм: первое число постепенно придвигается ко второму до их слияния, а потом стирается одна метка (иначе результат оказался бы на единицу больше правильного).



Таким образом, машину Поста можно рассматривать как упрощенную модель ЭВМ. В самом деле, как ЭВМ, так и машина Поста имеют:

- неделимые носители информации (клетки - биты), которые могут быть заполненными или незаполненными;
- ограниченный набор элементарных действий - команд, каждая из которых выполняется за один такт (шаг).

Обе машины работают на основе программы. Однако, в машине Поста информация располагается линейно и читается подряд, а в ЭВМ можно

читать информацию по адресу; набор команд ЭВМ значительно шире и выразительнее, чем команды машины Поста и т.д.

4.2 Тезис Поста

Как видно из изложенного выше, машина Поста позволяет реализовать основные конструктивные процедуры алгоритма. Естественно, возникает вопрос: “Все ли алгоритмы можно представить процедурами машины Поста?” Ответ на этот вопрос содержится в ***тезисе Поста***: “***Всякий алгоритм представим в форме машины Поста”.*** Этот тезис одновременно является формальным определение алгоритма.

Тезис Поста, как и тезис Тьюринга, является гипотезой. Его невозможно строго доказать (так же, как и тезис Тьюринга), потому что в нём фигурирует, с одной стороны нестрогое понятие алгоритма, а с другой стороны – точное понятие «машина Поста». В теории алгоритмов доказано, что машина Поста и машина Тьюринга эквивалентны по своим возможностям.

5. Нормальные алгоритмы Маркова

5.1 Базовые положения

Теория нормальных алгоритмов (или алгорифмов, как называл их создатель теории) была предложена советским математиком А. А. Марковым (1903–1979) в 1947 с целью уточнения понятия "алгоритм", что позволило решать задачи по определению алгоритмически неразрешимых проблем. Эти алгоритмы представляют собой некоторые правила по переработке слов в каком-либо алфавите, так что исходные данные и искомые результаты для алгоритмов являются словами в некотором алфавите.

Введем необходимые определения. ***Алфавит*** - конечное непустое множество символов, при помощи которых описывается алгоритм и данные. Алфавит символов зависит от конкретной задачи и, как правило, включает в себя символы латинского алфавита и цифры, а также различные специальные знаки. Любая последовательность непустых символов данного алфавита называется ***словом***. Пустое слово не имеет в своем составе ни одного символа и обозначается символом Λ. Если A и B – два алфавита, причем $A \subseteq B$, то алфавит B называется ***расширением алфавита*** A. Слова, как правило, обозначают латинскими буквами P, Q, R…(часто с индексами). Алгоритмы будем обозначать символами M_1, M_2, ….

Процесс работы алгоритма над алфавитом A заключается в последовательном порождении слов в A. Если некоторый алгоритм M в алфавите A слово P перерабатывает в слово Q, то это записывается так:

$$M: P \Rightarrow Q. \qquad (5.1)$$

Говорят, что алгоритм M применим к слову P, если процесс работы над этим словом заканчивается и получен результат.

Запись

$$!M(P) \qquad (5.2)$$

означает, что алгоритм M применим к слову P.

Пусть B – алфавит, а M_1,и M_2 – алгоритмы над B. Алгоритмы M_1,и M_2 называются эквивалентными над B, если для любых слов P и Q в алфавите B выполняются условия:

1) Если M_1: $P \Rightarrow Q$, то M_2: $P \Rightarrow Q$,
2) Если M_2: $P \Rightarrow Q$, то M_1: $P \Rightarrow Q$.

Одно слово может быть составной частью другого слова. Тогда первое называется подсловом второго или вхождением во второе. Например, если A - алфавит русских букв, то можем рассмотреть такие слова: P_1 = *параграф,* P_2=*граф,* P_3=*ра*. Слово P_2 является подсловом слова P_1, а P_3 - подсловом P_1 и P_2, причем в P_1 оно входит дважды. Особый интерес представляет первое вхождение.

Определение. ***Марковской подстановкой*** называется операция над словами, задаваемая с помощью упорядоченной пары слов (P, Q), состоящая в следующем. В заданном слове R находят первое вхождение слова P (если таковое имеется) и, не изменяя остальных частей слова R, заменяют в нем это вхождение словом Q. Полученное слово называется результатом применения марковской подстановки (P, Q), к слову R. Если же первого вхождения P в слово R нет (и, следовательно, вообще нет ни одного вхождения P в R), то считается, что марковская подстановка (P, Q), неприменима к слову R.

Частными случаями марковских подстановок являются подстановки с пустыми словами: (Λ, Q), (P, Λ), (Λ, Λ).

Пример 5.1 К слову R=*Саров* применим подстановку *(р, рат)*. Т.е. в нашем случае P=*р,* Q=*рат*. Результатом подстановки будет слово *Саратов.*

Пример 5.2 В таблице 5.1 приведены примеры марковских подстановок.

В каждой строке которой сначала дается преобразуемое слово, затем применяемая к нему марковская подстановка и, наконец, получающееся в результате словно:

Таблица 5.1

Преобразуемое слово *(R)*	Марковская подстановка *(P, Q)*	Результат
138 578 926	*(8 578 9, 00)*	*130 026*
Тарарам	*(ара, Λ)*	*трам*
Шрам	*(ра, ар)*	*шарм*
Функция	*(Λ, ζ-)*	*ζ-функция*
Логика	*(ика, Λ)*	*лог*
Книга	*(Λ, Λ)*	*книга*
Поляна	*(пор, т)*	[неприменима]

Для обозначения марковской подстановки используется запись $P \to Q$. Она называется формулой подстановки *(P, Q)*. Некоторые подстановки называют заключительными (смысл названия станет ясен чуть позже). Для обозначения таких подстановок используют специальную запись, называя ее формулой заключительной подстановки. Слово P называется левой частью, а Q - правой частью в формуле подстановки. Для обозначения заключительных подстановок часто используют запись $P|\to Q$ (***термальное правило*** подстановки) или $P \to \cdot\, Q$, называемая формулой заключительной подстановки.

Нормальный алгоритм Маркова представляет собой совокупность правил подстановки $S_1, S_2, \ldots, S_n$.

$$\begin{aligned} &S_1: P_1 \to Q_1; \\ &S_2: P_2 \to Q_2; \\ &\ldots\ldots\ldots; \\ &S_n: P_n \to Q_n. \end{aligned} \qquad (5.3)$$

Преобразование слова состоит в поиске в нем подслова, совпадающего с левой частью какого-либо правила подстановки, и замене этого подслова правой частью данного правила подстановки.

При рассмотрении алгоритмов Маркова более удобно обозначать правила подстановки номером, а не символом правила типа S_i, а при

выполнении шага преобразования указывается исходное слово, номер применяемого правила и результат.

Пример 5.4. Задан алфавит $A=\{a,b\}$ и алгоритм Маркова содержит два правила подстановки:

1. $aa \rightarrow aba;$
2. $bb \rightarrow bab.$

Исходное слово *aabbaa*.

В результате выполнения алгоритма получаем:

1) $aabbaa \rightarrow (1)ababbaa;$
2) $ababbaa \rightarrow (1)ababbaba;$
3) $ababbaba \rightarrow (2)abababababa,$

т.е. результатом работы алгоритма над исходным словом *aabbaa* является слово *abababababa*.

Дадим теперь четкое описание алгоритма преобразования слова.

Согласно концепции нормальных алгоритмов Маркова, преобразование слова P по правилам подстановки $S_1, S_2, \ldots, S_n$ представляет собой последовательность однотипных шагов.

Опишем один шаг преобразования.

1. Последовательно просматриваются правила преобразования S_1, S_2….
2. Если левая часть P_i некоторого правила S_i входит в слово, то входное слово P_i в слове заменяется правой частью Q_i.

Данные шаги последовательно выполняются друг за другом.

Важнейшим вопросом является процесс завершения преобразования слов. Согласно концепции нормальных алгоритмов Маркова, завершение преобразования слова происходит в двух случаях:

1. На некотором шаге преобразования ни одно из правил подстановки $S_1, S_2, \ldots, S_n$ оказывается неприменимым к слову, т.е. ни одна из левых частей $P_1, P_2, \ldots. P_n$ в слове не содержится;
2. Если на некотором шаге преобразования к слову было применено ***терминальное правило***, то процесс преобразования считается завершенным, т.е. новый просмотр правил не производится.

Заметим, что в терминальных правилах вместо стрелки $\rightarrow$ используется обозначение $|\rightarrow$.

Следует также отметить несколько важных замечаний:

1. Если левая часть некоторого правила входит в слово более одного раза, то заменяется только самое левое вхождение (на одном шаге преобразования);

2. После каждого применения некоторого правила S_i к слову, новый просмотр правил всегда начинается с первого правила S_1;
3. Левая часть правила может быть пустой. Правило с пустой левой частью считается применимым к любому слову. Его действие состоит в том, что к началу слова добавляется правая часть данного правила;
4. Правая часть также может быть пустой. Результатом применения такого правила является удаление из слова последовательности символов, совпадающих с левой частью правил.

Еще раз вернемся к примеру 5.4, где задан алфавит $A=\{a,b\}$ и два правила подстановки:

1. $aa \rightarrow aba;$
2. $bb \rightarrow bab.$

Исходное слово *aabbaa*.

В результате выполнения алгоритма получаем:

1)aabbaa $\rightarrow$ *(1)ababbaa;*

2)ababbaa $\rightarrow$*(1)ababbaba;*

3)ababbaba $\rightarrow$ *(2)ababababa,*

Как видно, исходно слово содержит комбинацию *aa* и, следовательно, к нему применимо первое правило подстановки. Полученное слово *ababbaa* также содержит комбинацию *aa*, и поэтому к нему вновь будет применимо первое правило. (Напомним, что после каждого правила новый просмотр всегда начинается с первого правила). Получившееся после второй замены слово уже не содержит комбинацию *aa*, но содержит комбинацию *bb*. Следовательно, к нему неприменимо первое правило подстановки, но применимо второе правило подстановки. К получившемуся после третьей подстановки слову неприменимо ни одно из правил подстановки, и, следовательно, преобразование завершено.

Рассмотрим два простых примера.

Пример 5.5. Пусть на множестве $A=\{a,b\}$ задан нормальный алгоритм Маркова M_1 правилами подстановки:

1. $ba \rightarrow ab$
2. $ab \rightarrow$

Пусть задано исходное слово *R=baaab.*

Тогда в результате работы алгоритма получаем:

1) baaab $\rightarrow$ *(1)abaab;*

2) abaab $\rightarrow$*(1)aabab;*

3) aabab $\rightarrow$*(1)aaabb;*

4) aaabb →(2)aab;

5) aab → (2)a.

Итак, в соответствии с заданным алгоритмом исходной слово *R=baaab* переработано в *a*.

Пусть теперь на этом же множестве для этого же исходного слова *R=baaab* задан алгоритм M_2 правилами подстановки:

1. ba→

2. ab→

Результат работы алгоритма M_2:

1) baaab → (1)aab;

2) aab → (2) a.

Т.е. в результате выполнения алгоритма M_2 получен такой же результат, что и для алгоритма M_1. Следовательно, алгоритмы M_1 и M_2 эквивалентны. Но сразу отметим, что алгоритм M_2 работает быстрее, т.е. приводит к результату всего за два шага, вместо пяти для алгоритма M_1.

5.2 Особенности работы нормальных алгоритмов Маркова.

1. Перестановка местами правил подстановки может влиять как на последовательность преобразований, так и на конечный результат. Чтобы этот факт проиллюстрировать, рассмотрим пример 5.5, но правила подстановки поменяем местами, т.е:

1. ab→

2. ba→ab

Преобразуем тоже слово *R=baaab*:

1) *baaab→(1)baa;*

2) *baa →(2)aba;*

3) *aba →(1)a.*

Видно, что в данном случае, поменяв последовательность преобразований, результат получается прежним.

Рассмотрим другой пример.

Пример 5.6. Пусть имеется алфавит символов *A={a,b}* и задан нормальный алгоритм Маркова правилами подстановки:

1. aba → bb

2. bab → aa

Применим этот алгоритм к слову *abababa.*

Получаем:

1) abababa → bbbaba;

2) bbbaba → bbbbb.

Теперь изменим порядок следования правил подстановки и применим этот алгоритм к тому же слову.

1. bab →aa

2. aba → bb

В результате получаем:

1) abababa → (1)aaaaba;

2) aaaaba → (2)aaabb.

Т.е., из этого примера видно, что изменение последовательности преобразований в алгоритмах Маркова ведет к изменению результата.

2. Правило подстановки с пустой левой частью применимо к любому слову. Это означает, что оно должно всегда быть последним в списке правил. Иначе оно блокирует все последующие правила.

Пример 5.7. Задан алфавит символов *A={a,b}* и нормальный алгоритм Маркова содержит три правила подстановки:

1. aa → b

2. → b

3. b → a

Данный алгоритм нужно применить с слову *aabb*.

1) aabb → (1)bbb;

2) bbb → (2) bbbb;

3) bbbb → (2) bbbbb;

……

и т.д.

Как видно, третье правило заблокировано и оно недостижимо, и поэтому никогда не выполнится.

3. Если в списке правил подстановки есть правило с пустой левой частью, то для завершения работы алгоритма Маркова в нем должно присутствовать хотя бы одно терминальное правило подстановки.

В самом деле, условие окончания преобразования слова – либо нет подходящего правила подстановки, либо применено терминальное правило. Так как правило с пустой левой частью применимо к любому слову, то единственным условием окончания преобразования является применение к нему терминального правила подстановки.

Пример 5.8. Задан алфавит символов *A={a,b}* и правила подстановки:

1. aba → bb

2. → ab

Т.к. среди заданных правил подстановки нет терминального правила подстановки и имеется правило с пустой левой частью *(2)*, которое всегда применимо и добавляет к началу любого слова буквы *ab*, то процесс преобразования будет бесконечным. Действительно, пусть имеется начальное слово *abb*. Применив алгоритм преобразования, получаем:

abb → ababb → bbbb → abbbbb → ababbbbb → bbbbbbb → ababbbbbbb ....

(2) (1) (2) (2) (1) (2)

(Здесь мы для удобства изменили формат представления алгоритма (3.3) последовательной цепочкой преобразования, указав алгоритмические правила преобразования в скобках).

Т.е. видно, что процесс преобразования будет бесконечным.

Изменим алгоритм преобразования:

1. aba |→ bb

2. → ab

Т.е., как только будет применено первое правило, процесс преобразования завершится. Для нашего начального слова *abb* результат преобразования будет выглядеть как:

abb → ababb → bbbb.

(2) (1)

Как видно, процесс преобразования завершится на втором шаге.

Но возможны случаи, когда, не смотря на наличие терминального правила подстановки, процесс преобразования слова является бесконечным. Такие примеры будут приведены ниже.

4. Пусть имеется два правила подстановки S_i, S_k такие, что левая часть одного правила P_i является подстрокой левой части P_k другого правила подстановки. Тогда правило S_k (более длинное слово) должно стоять в списке правил выше правила S_i (короткое слово). В противном случае правило S_k никогда не выполнится.

Пример 5.9 Задан алфавит символов *A={a,b}*. Рассмотрим следующие правила подстановки:

1. a → bbb

2. aa → ab

Видно, что левая часть первого правила - это подстрока левой части второго правила. Второе правило подстановки никогда не будет применено ни к какому слову. Действительно, если в некоторое слово входит *aa*, то

входит и одна буква *a*, и, следовательно, к такому слову всегда будет применяться только первое правило подстановки. Здесь как бы первое правило блокирует выполнение второго правила.

Рассмотрим применение данного алгоритма к слову *aa*:

$aa \to bbba \to bbbbbb.$

(1) (1)

Теперь поменяем правила местами:

1. $aa \to ab$

2. $a \to bbb$

Тогда получаем:

$aa \to ab \to bbbb$

(1) (2)

Как видим, здесь уже применяется и первое, и второе правила преобразования.

Рассмотрим еще несколько практических примеров на составление нормальных алгоритмов Маркова.

Пример 5.10. Задан алфавит символов *A={a}*. Необходимо составить нормальный алгоритм Маркова, удваивающий в слове каждую букву *a*.

Решение.

Для начала рассмотрим алгоритм Маркова, состоящий из одного правила подстановки:

1. $a \to aa$

Сразу отметим, что этот алгоритм задачу удвоения не решает, так как он просто добавляет по одной букве *a* к слову на каждом шаге преобразования. Например, для входного слова *aaa* получаем:

$aaa \to aaaa \to aaaaa \to ....$

Т.е. удваивания символа не происходит, и, следовательно, алгоритм не верный.

Для решения поставленной задачи составим следующий алгоритм преобразования слова:

1) К началу слова добавляется дополнительный символ *. Этот символ не входит в алфавит.

2) Дополнительный символ "прогоняется" через слово с одновременной заменой каждого символа в слове согласно условию задачи.

3) В конце преобразований дополнительный символ удаляется.

Эти действия могут быть реализованы следующим нормальным алгоритмом Маркова:

1. *a → aa*
2. * |→
3. → *

Т.е. Имеем три правила подстановки. Третье правило добавляет к слову дополнительный символ *. Это правило состоит из пустой левой части, его можно применить всегда к любому слову. С помощью первого правила подстановки символ * "прогоняется" через слово с одновременной заменой *a* на *aa*. В результате этого символ * оказывается в конце слова. Второе правило удаляет символ * по окончании преобразования слова. Это правило является терминальным, поэтому его применение означает завершение преобразования слова. Новый просмотр правил не производится.

Рассмотрим, как данный алгоритм Маркова преобразует входные слова *aa* и *aaa*.

$$aa \xrightarrow{(3)} *aa \xrightarrow{(1)} aa*a \xrightarrow{(1)} aaaa* \xrightarrow{(2)} aaaa$$

$$aaa \xrightarrow{(3)} *aaa \xrightarrow{(1)} aa*aa \xrightarrow{(1)} aaaa*a \xrightarrow{(1)} aaaaaa$$

По опыту предыдущей задачи можно сделать следующие важные замечания.

Замечание 1. Правило с пустой левой частью всегда должно стоять в списке правил на последнем месте. Применяться же оно должно раньше остальных правил. То есть никакое другое правило не должно быть применено к слову до применения к нему последнего правила. А это в свою очередь означает, что все правила (кроме последнего) должны содержать в своей левой части дополнительный символ *.

Согласно концепции нормальных алгоритмов Маркова, дополнительные символы (символы, не входящие в алфавит) не могут содержаться в исходном слове. Они могут входить в слово в процессе его преобразования. Поэтому, любое правило, содержащее в своей левой части дополнительный символ, к исходному слову не применимо.

Замечание 2. Правило, удаляющее дополнительный символ, должно быть применено лишь после того, как будет завершен процесс преобразования слова согласно условию задачи, т.е. после "прохода" символа * через слово. В нашем примере это происходит только тогда, когда становится неприменимым первое правило.

Замечание 3. Правило, удаляющее дополнительный символ, должно быть терминальным. Иначе процесс преобразования слова станет бесконечным.

Пример 5.11 Задан алфавит символов *A={a,b}.* Составить нормальный алгоритм Маркова, выполняющий замену буквы *a* на *b* и наоборот – замену *b* на *a*.

Решение.

Следует сразу отметить, что алгоритм Маркова, состоящий из двух правил:

1. $a \to b$
2. $b \to a$

поставленную задачу не решает.

Эта задача также не решается и с помощью алгоритма Маркова, состоящего из трех правил (обмен значений двух переменных с использованием дополнительной третьей переменной):

1. $a \to c$
2. $b \to a$
3. $c \to b$

В случае алгоритмов Маркова этот метод не работает.

Для решения этой задачи необходимо действовать так же, как в предыдущей задаче (по пунктам 1-3 плана). Все эти действия реализуют нормальный алгоритм Маркова:

1. $*a \to b*$
2. $*b \to a*$
3. $* \;\; |\to$
4. $\to *$

Посмотрим, как данный нормальный алгоритм Маркова преобразует входное слово *abaa*:

$$abaa \underset{(4)}{\to} *abaa \underset{(1)}{\to} b*baa \underset{(2)}{\to} ba*aa \underset{(1)}{\to} bab*a \underset{(1)}{\to} babb* \underset{(3)}{\to} babb.$$

Пример 5.12 Задан алфавит символов *A={a,b}.* Составить нормальный алгоритм Маркова, который удваивает последнюю букву в слове.

Решение.

Принцип построения алгоритма для данной задачи аналогичен принципам построения, рассмотренных в предыдущих примерах.

1. $*a \to a*$

2. $*b \to b*$
3. $a* \mid \to aa$
4. $b* \mid \to bb$
5. $* \mid \to$
6. $\to *$

Сначала вводим дополнительный символ * (правило 6). "Прогоняем" его через слово без изменения слова (правила 1 и 2). После этого (справа от * не осталось символов) удваиваем символ слева от * (терминальные правила 3 и 4). Удаляем вспомогательный символ. Правило 5 необходимо для корректной обработки пустого слова. Согласно концепции нормальных алгоритмов Маркова, исходное слово может быть пустым, т.е. не содержать ни одного символа. В данном случае пустое слово нужно оставить без изменения.

Посмотрим, как данный нормальный алгоритм Маркова преобразует входные слова *abba* , *abbb* и пустое слово (Λ):

$abba \to *abba \to a*bba \to ab*ba \to abb*a \to abba* \to abbaa.$
(6) (1) (2) (2) (1) (3)

$abbb \to *abbb \to a*bbb \to ab*bb \to abb*b \to abbb* \to abbbb.$
(6) (1) (2) (2) (2) (4)

$\Lambda \to * \to \Lambda$
(6) (5)

Пример 5.13 Задан алфавит символов $A=\{a,b\}$. Составить нормальный алгоритм Маркова, который удваивает все буквы в слове, кроме последней.

Решение.

Идея этого алгоритма состоит в том, что символ * "прогоняется" через слово с одновременным удваиванием в нем каждой буквы (правила 1 и 2). После этого * удаляется вместе с последним символом строки (терминальные правила 3 и 4). Правило 5 необходимо для корректной обработки пустого слова. Эти этапы реализуются следующим нормальным алгоритмом Маркова:

1. $*a \to aa*$
2. $*b \to bb*$
3. $a* \mid \to$
4. $b* \mid \to$

5. * $|\rightarrow$

6. $\rightarrow$ *

Возьмем исходное слово *aba* и применим к нему описанный алгоритм:

$aba \rightarrow *aba \rightarrow aa*ba \rightarrow aabb*a \rightarrow aabbaa* \rightarrow aabba.$

(6) (1) (2) (1) (3)

Пример 5.14 Задан алфавит символов *A={a,b,c}*. Составить нормальный алгоритм Маркова, который удаляет только первую букву слова.

Решение.

В данном случае символ * не надо "прогонять" через все слово. Он будет удален с первой буквой слова. Задачу решает алгоритм Маркова, содержащий пять правил подстановки:

1. *a $|\rightarrow$
2. *b $|\rightarrow$
3. *c $|\rightarrow$
4. * $|\rightarrow$
5. $\rightarrow$ *

Например, для исходного слова *bcc* получаем:

$bcc \rightarrow *bcc \rightarrow cc.$

(5) (2)

А, например, для слова *ccbac* аналогично получаем:

$ccbac \rightarrow *ccbac \rightarrow cbac.$

(5) (3)

Пример 5.15 Задан алфавит символов *A={a,b}*. Составить нормальный алгоритм Маркова, который оставляет только первую букву слова.

Решение.

Идея алгоритма заключается в том, чтобы удалить из слова вторую, третью и последующие буквы (правила 1-4). Эти правила удаляют второй символ после символа *. При этом, символ * находится слева от слова. Как только все буквы, кроме первой, будут удалены, преобразование слова завершается. Пятое терминальное правило удаляет вспомогательный символ.

1. *aa $\rightarrow$ *a
2. *ab $\rightarrow$ *a

3. $*ba \to *b$

4. $*bb \to *b$

5. $* \quad |\to$

6. $\to \ *$

Рассмотрим действие алгоритма на два слова *bba* и *aba*.

$bba \to *bba \to *ba \to *b \to b$
(6) (4) (3) (5)

$aba \to *aba \to *aa \to *a \to a$
(6) (2) (1) (5)

Нужно отметить, что для нормальных алгоритмов разработана техника программирования, позволяющая осуществлять операции композиции алгоритмов, реализовывать операторы «если *A*, то выполнить *F1*, иначе *F2*», «пока *A*, выполнять *F1*, иначе *F2*», что существенно расширяет класс используемых функций.

Приведенная концепция нормальных алгоритмов Маркова, ее возможность организации ветвления и цикличных процессов вычисления и позволяет сделать вывод, что всякий алгоритм может быть задан нормальным алгоритмом Маркова. В этом и состоит тезис Маркова, который следует понимать как определение алгоритма.

Создатель теории нормальных алгоритмов советский математик А. А. Марков выдвинул гипотезу, получившую название **"Принцип нормализации Маркова"**. Согласно этому принципу, для нахождения значений функции, заданной в некотором алфавите, тогда и только тогда существует какой-нибудь алгоритм, когда функция нормально вычислима.

Сформулированный принцип, как и тезисы Тьюринга и Чёрча, носит внематематический характер и не может быть строго доказан.

6. Эквивалентность теорий алгоритмов

Рассмотрев несколько основных теорий, уточняющих понятие алгоритма, следует отметить их взаимосвязанность, которая определяется следующей теоремой:

Теорема 6.1 Классы всех частично-рекурсивных функций, нормально вычислимых функций, функций, вычислимых по Тьюрингу (Посту) совпадают.

Эта теорема имеет строгое математическое доказательство, которое здесь приводить не будем, а уясним смысл этой теоремы, который и

заключается в том, что теории рекурсивных функций, нормальных алгоритмов Маркова, машин Тьюринга (Поста), машин с неограниченными регистрами равносильны. И если бы один из этих классов оказался шире какого-либо другого класса, то соответствующий тезис Чёрча, Маркова или Тьюринга был бы опровергнут. Например, если бы класс нормально вычислимых функций оказался шире класса частично-рекурсивных функций, то существовала бы нормально вычислимая, но не частично-рекурсивная функция, и, следовательно, она была бы алгоритмически вычислима. Но тогда утверждение о том, что она не является частично-рекурсивной, опровергало бы тезис Чёрча. Но в соответствии с приведенной теоремой 6.1 таких функций не существует. Следует отметить, что наряду с выше приведенными теориями алгоритмов, существуют и другие, но для всех них также доказана равносильность с рассматриваемыми теориями.

Итак, всякий алгоритм, описанный, например, в терминах частично-рекурсивных функций, можно реализовать машиной Тьюринга, Поста, нормальным алгоритмом Маркова, МНР и наоборот. Отсюда следует, что любые утверждения о существовании или не существовании алгоритмов, сделанные в одной из теорий, верны и в другой.

Следует обратить внимание на прикладное значение рассматриваемых здесь теорий алгоритмов. Ввиду инвариантности основных результатов общей теории алгоритмов, их прикладное значение никак не связано с тем, насколько близки к практике используемые в них теоретические модели алгоритмов. Конечно, машины Тьюринга далеки от современных компьютеров, а рекурсивные функции весьма далеки от языков программирования из-за предельной скромности своих средств. Но именно скромность средств, во-первых, делает эти модели чрезвычайно удобным языком доказательств, во-вторых, позволяет понять, без чего нельзя обойтись, а без чего можно и какой ценой, т.е. отличать удобства от принципиальных возможностей. Иначе говоря, прикладное значение рассматриваемых моделей заключается в том, что с их помощью удобно строить теорию, верную для любых алгоритмических моделей, в том числе и сколь угодно близких к практике.

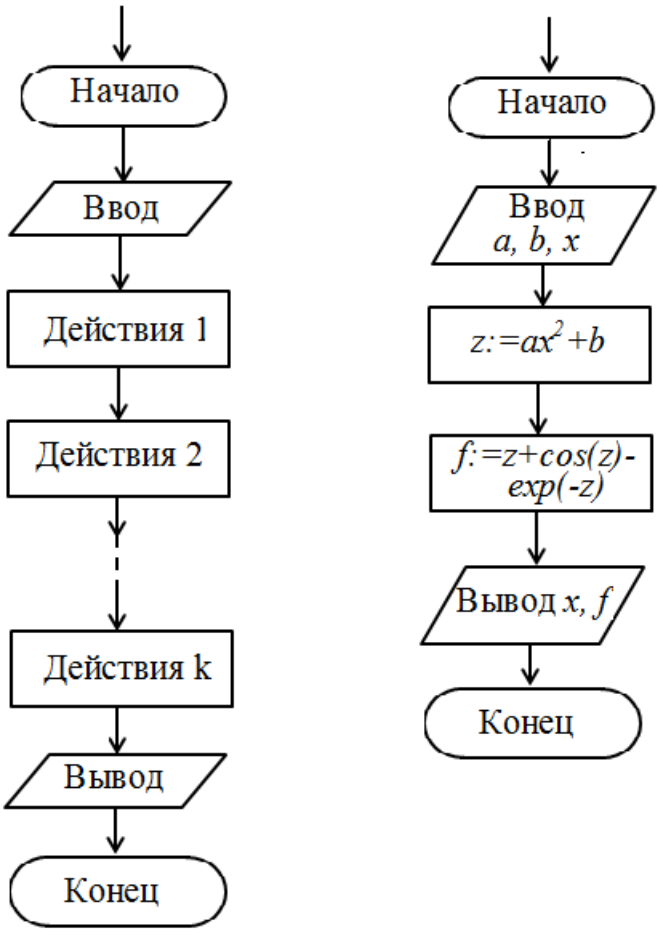
7. Оценка сложности алгоритма

Прежде чем приступить к рассмотрению основных вопросов анализа и оценки сложности алгоритмов, рассмотрим их классификацию, которая является косвенным фактором оценки сложности алгоритмов.

7.1 Классификация алгоритмов

Существует большое многообразие классификации алгоритмов по ряду признаков. В зависимости от базовой структуры различают алгоритмы **линейные, разветвляющиеся и циклические.**

Алгоритм называется **линейным**, если все действия выполняются однократно в строго определенной последовательности (базовая структура следование). На рис. 7.1а приведена структурная блок-схема линейного алгоритма, а на рис. 7.1б приведен пример линейного алгоритма для вычисления функции $f(x)=ax^2+b+cos(ax^2+b)-e^{-ax^2+b}$.



а) б)

Рис. 7.1 Блок-схема (а) и пример (б) линейного алгоритма.

Алгоритм называется ***разветвляющимся*** (***ветвящимся***), если в нем существует выбор одной из нескольких альтернатив последовательных действий в зависимости от исходных или промежуточных данных (базовая

структура ветвления. При ветвлении происходит однократный проход по одной из ветвей решения задачи. Признаком ветвящегося алгоритма является наличие операций условного перехода, когда происходит проверка истинности некоторого логического выражения, и в зависимости от истинности или ложности проверяемого условия выбирается соответствующая ветвь для дальнейшей реализации алгоритма.

Под логическим выражением понимается выражение, записанное с помощью операций сравнения: больше (“>”), меньше (“<”), равно (“=”), больше или равно “(>=”), меньше или равно (“<=”), не равно (“<>”). При составлении логических выражений, включающих составные условия, используются логические связки “И”, “ИЛИ”, “НЕ”.

На практике наиболее часто встречаются четыре варианта ветвления алгоритма:

1. **Если-то** (Рис. 7.2 а)

Описание алгоритма:
если условие
то действия
все

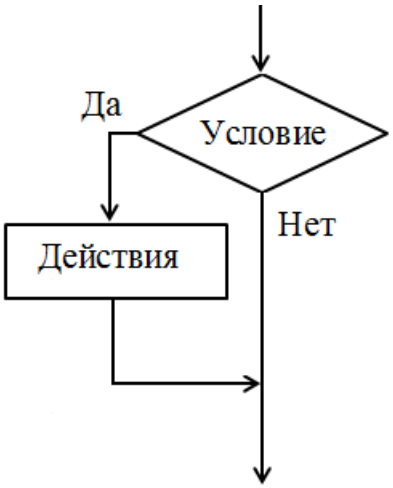


Рис. 7.2 а

2. **Если-то-иначе** (Рис. 7.2 б);

Описание алгоритма:
если условие
то действия 1
иначе действия 2
все

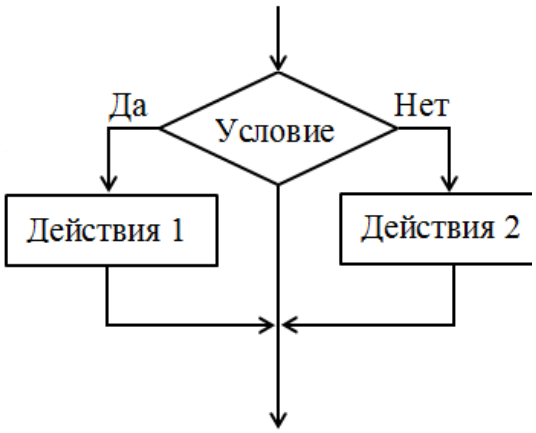


Рис. 7.2 б

3. **Выбор** (Рис. 7.2 в);

Описание
алгоритма**: выбор**
при условие 1:
действия 1
при условие 2:
действия 2

……

при условие N:
действия N
все

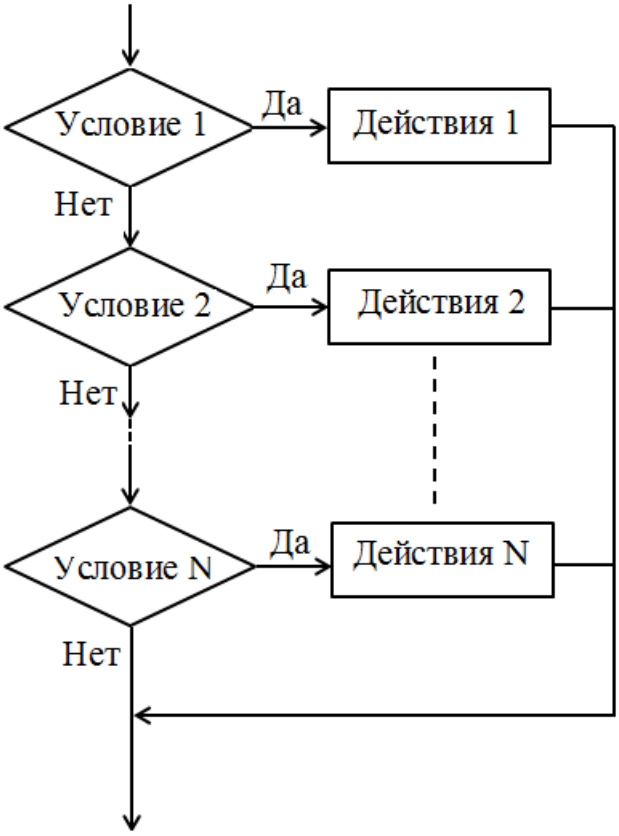


Рис. 7.2 в

4. **Выбор-иначе** (Рис. 7.2 г);

Описание
алгоритма: **выбор**
при условие 1:
действия 1
при условие 2:
действия 2

……

при условие N:
действия N
иначе
действия N+1
все

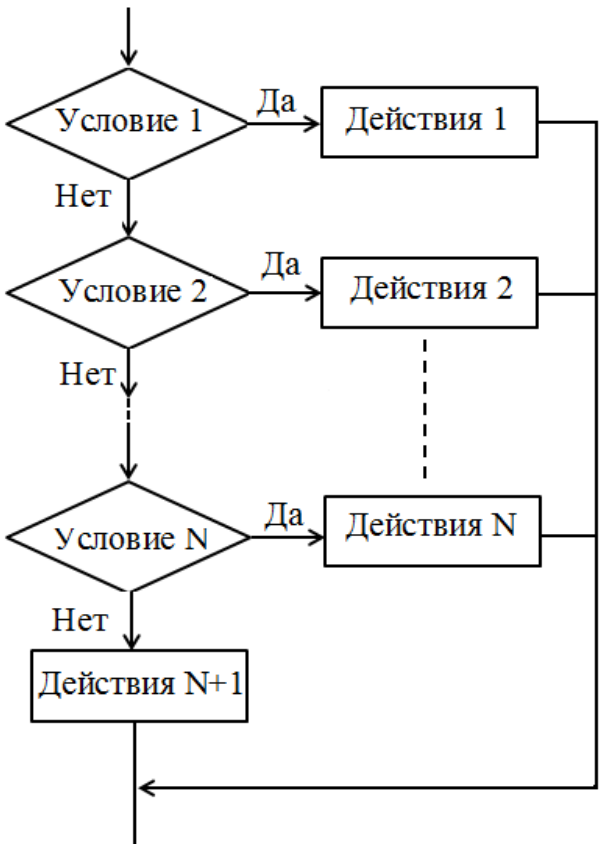


Рис. 7.2 г

Алгоритм называется ***циклическим***, если при каждом исполнении предписывается многократное выполнение одной и той же последовательности действий – тела цикла (базовая структура цикл).

Наиболее распространенными циклами являются:

1. Цикл **пока** с предусловием - Серия шагов повторяется до тех пор, пока условие цикла истинно (Рис. 7.3 а).

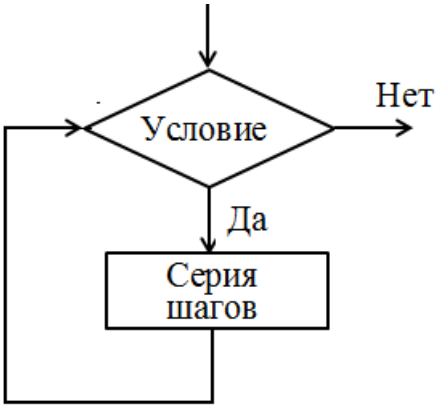


Рис. 7.3 а

Описание алгоритма:

нц пока условие
тело цикла
все

2. Цикл "**до**" с постусловием –Серия шагов выполняется до тех пор, пока условие цикла ложно (Рис. 7.3 б).

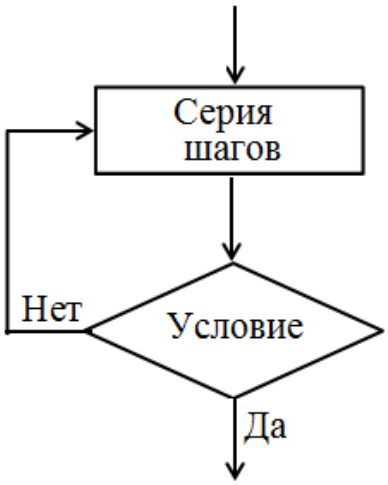


Рис. 7.3б

Описание алгоритма:

нц
тело цикла
до условие
кц

3. Цикл «**для**» (цикл с параметром) – предписывает выполнять тело цикла для всех значений некоторой переменной (параметра цикла) в заданном диапазоне.
Цикл "для" используется для ситуаций, когда заранее известно количество повторений некоторых действий (Рис. 7.3 в).

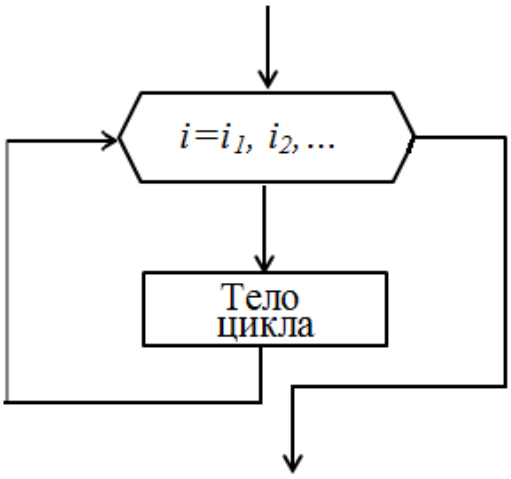


Рис. 7.3 в.

Описание алгоритма:

нц для i **от** N1 **до** N2
тело цикла
кц

Как правило, в языках программирования имеются команды, реализующие показанные выше структуры.

Существенная особенность перечисленных базовых структур состоит в том, что каждая из них имеет один вход и один выход. Их можно соединять друг с другом в любой последовательности. В качестве действия может использоваться любая из перечисленных структур, что обеспечивает возможность вложенности одних структур в другие. Возврат назад выполняется только в циклах.

Следует отметить, что по способу исполнения выделяют также:

Вспомогательные алгоритмы - алгоритмы, целиком используемые в составе других алгоритмов. Вспомогательный алгоритм представляет алгоритм решения некоторой подзадачи из исходной (основной) задачи. Вспомогательный алгоритм, записанный на языке программирования, называется ***процедурой*** или ***подпрограммой***.

Алгоритм может содержать обращение к самому себе как ***вспомогательному*** и в этом случае он называется рекурсивным.

Параллельные алгоритмы – алгоритмы, предназначенные для вычислительных машин, способных выполнять несколько операций одновременно.

В отдельные классы выделяют ***имитирующие*** и ***эмпирические*** алгоритмы.

Имитирующие алгоритмы создаются на основе описаний действий объектов, наблюдений, зависимости исходных данных от изменяющихся условий.

Эмпирические алгоритмы – это алгоритмы, основанные на опыте, изучении фактов и опирающиеся на непосредственное наблюдение и эксперимент.

Выделяют также ***случайные и самоизменяющиеся*** алгоритмы.

Алгоритм называется случайным, если в процессе его выполнения предусматривается возможность случайного выбора отдельных действий.

Алгоритм называется самоизменяющимся, если он не только перерабатывает входные слова, но и сам изменяется в процессе такой переработки.

Эвристические алгоритмы – это алгоритмы теоретически не обоснованные, но позволяющие сократить количество переборов в пространстве поиска. Отметим, что универсальной структуры описания эвристических алгоритмов не существует.

7.2 Основы анализа алгоритмов

Традиционно принято оценивать степень сложности алгоритма по объему используемых им основных ресурсов компьютера: процессорного времени и оперативной памяти. В связи с этим вводятся такие понятия, как ***временная*** сложность алгоритма и ***объемная*** (***пространственная***) сложность алгоритма.

Очень часто при реализации многих алгоритмов существует выбор между объёмом памяти и временем его решения, т.е. поставленную задачу можно решить быстро, использую большой объём памяти, или медленнее, занимая меньший объём. Типовым примером такой ситуации служит, например, алгоритм поиска кратчайшего пути. Представив себе карту некоторой местности в виде сети, можно написать алгоритм для определения кратчайшего расстояния между двумя любыми точками этой сети. Чтобы не вычислять эти расстояния всякий раз, когда они нам нужны, мы можем вывести кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда нам понадобится узнать кратчайшее расстояние между двумя заданными точками, мы можем просто взять готовое расстояние из таблицы. Результат будет получен практически сразу, но это потребует огромного объёма памяти. Карта местности, например большого административного района, может содержать десятки тысяч точек. Тогда, таблица, должна содержать более 10 млрд. ячеек. Т.е. для того, чтобы обеспечить быстродействие алгоритма таким образом, необходимо использовать дополнительные ~10 Гб памяти. Из этой зависимости проистекает идея объёмно-временной сложности. При таком подходе алгоритм оценивается, как с точки зрении скорости выполнения, так и с точки зрения используемой памяти. Поэтому, при оценке трудоемкости алгоритма необходимо учитывать и ***временную***, и ***объемную*** (***пространственную***) сложность алгоритма.

- ***Временная сложность*** алгоритма определяет число шагов, которые должен предпринять алгоритм, в зависимости от объема входящих данных (n).
- ***Объемная сложность*** алгоритма определяет количество памяти, которое потребуется занять для работы алгоритма, в зависимости от объема входящих данных (n).

Сразу следует отметить, что объемная сложность алгоритма (программы) становится критической, когда объем обрабатываемых данных оказывается на пределе объема оперативной памяти используемой ЭВМ. На современных компьютерах острота этой проблемы снижается

благодаря как росту объема ОЗУ, так и эффективному использованию многоуровневой системы запоминающих устройств. Программе оказывается доступной очень большая, практически неограниченная область памяти (виртуальная память). Недостаток основной памяти приводит лишь к некоторому замедлению работы из-за обменов с диском. Используются приемы, позволяющие минимизировать потери времени при таком обмене. Это использование кэш-памяти и аппаратного просмотра команд программы на требуемое число ходов вперед, что позволяет заблаговременно переносить с диска в основную память нужные значения. Исходя из этого, можно сделать вывод, что минимизация емкостной сложности не является первоочередной задачей. Поэтому, при анализе трудоемкости алгоритмов следует, в основном обращать большее внимание на его временную сложность.

7.2.1 Сравнительные оценки алгоритмов.

Одной из основных задач практического применения алгоритмов является проблема выбора наиболее эффективного или рационального алгоритма. Естественно, что задача выбора эффективного алгоритма для практического решения поставленной задачи связана с используемой системой сравнительных оценок, которая в свою очередь опирается на формальную модель алгоритма.

Наиболее часто в качестве формальной модели рассматривается абстрактная машина фон-Неймановской архитектуры, поддерживающая адресную память и набор элементарных операций соотнесенных с языком высокого уровня.

В целях анализа приняты следующие допущения:

- каждая команда выполняется за фиксированное время;
- исходные данные алгоритма представляются машинными словами по β битов каждое.

Алгоритм задается N словами памяти, так что на входе алгоритма имеется $N_\beta = N \cdot \beta$ бит информации.

Следует сразу отметить, что в ряде случаев, и особенно при рассмотрении матричных задач N является мерой длины входа алгоритма, отражающий линейную размерность.

Программа, реализующая алгоритм состоит из M машинных инструкций по β_M битов, так что $M_\beta = M \cdot \beta_M$ бит информации.

Кроме того, как правило, на реализацию алгоритма необходимы дополнительные ресурсы:

- S_d – память для хранения промежуточных результатов;
- S_r - память для организации вычислительного процесса (память, необходимая для реализации рекурсивных вызовов и возвратов).

При решении конкретной задачи, заданной $N+M+S_d+S_r$ словами памяти, алгоритм выполняет конечное количество «элементарных» операций абстрактной машины. В связи с этим вводится определение ***трудоёмкости алгоритма.***

Под ***трудоемкостью алгоритма*** $F_a(n)$ для данного конкретного входа, для решения конкретной задачи (проблемы), в данной формальной системе понимается количество элементарных операций, совершаемых алгоритмом. Вполне понятно, что трудоемкость алгоритма $F_a(n)$ будет зависеть не только от величины входного слова (размера задачи), но и от выбранного метода, которому можно поставить в соответствие базовую структуру.

Комплексный анализ алгоритма может быть выполнен на основе комплексной оценки ресурсов формальной системы, необходимых алгоритму для решения конкретных поставленных задач. Вполне очевидно, что для различных областей применения веса ресурсов будут различны, что приводит к следующей комплексной оценке алгоритма:

$$\Psi_A = c_1 \cdot F_a(N) + c_2 \cdot M + c_3 \cdot S_d + c_4 \cdot S_r, \qquad (7.1)$$

где c_i – веса ресурсов.

7.2.2 Классификация алгоритмов по виду функции трудоемкости

1.*Количественно-зависимые по трудоемкости алгоритмы.*

Их функция трудоемкости зависит только от размерности входных данных и не зависит от их конкретных значений:

$$F_a(n), \quad n=f(N). \qquad (7.2)$$

К таким алгоритмам относятся, например, алгоритмы для стандартных операций с массивами и матрицами: умножение матриц, умножение матрицы на вектор и т.д.

2. *Параметрически-зависимые по трудоемкости алгоритмы.*

Их функция трудоемкости определяется конкретными значениями обрабатываемых слов памяти:

$$F_a(n), \quad n=f(p_1,p_2, \ldots p_i). \qquad (7.3)$$

У таких алгоритмов на входе два числовых значения: аргумент функции и точность.

Пример 7.1 . Алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов.

a) Вычисление x^k последовательным умножением: $F_a\,(x,\,k) = F_a\,(k)$.

b) Вычисление $e^x = \Sigma(x^n/n!)$, с точностью до $\varepsilon > 0$: $F_a = F_a\,(x,\,\varepsilon)$.

3.*Количественно-параметрические по трудоемкости алгоритмы.*

В большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от их значений:

$$F_a(n),\quad n=f(N,p_1,p_2,\ \ldots\ p_i). \qquad (7.4)$$

Пример 7.2. Алгоритмы численных методов, в которых существует параметрически-зависимый цикл по точности и цикл количественно-зависимый по размерности.

Среди параметрически-зависимых алгоритмов выделяют группу алгоритмов, для которой количество операций зависит от порядка расположения исходных объектов.

Пример 7.3

- алгоритмы сортировки;
- алгоритмы поиска минимума/максимума в массиве.

7.2.3 Временной анализ трудоемкости алгоритмов

Как уже было сказано выше, анализ вычислительных алгоритмов связан в основном с анализом их временной сложности, с которой обычно связывают две теоретические проблемы. Первая состоит в поиске ответа на вопрос: до какого предела значения временной сложности можно дойти, совершенствуя алгоритм решения задачи? Этот предел зависит от самой задачи и, следовательно, является ее собственной характеристикой. Вторая проблема связана с классификацией алгоритмов по временной сложности. Если известна функция *f(n),* определяющая количество выполняемых операций при реализации алгоритма от входного слова *n*, то как она быстро растет? Существуют алгоритмы с линейной зависимостью, с квадратичной зависимостью и с зависимостью более высоких степеней, которые называются ***полиномиальными*** и реализация которых осуществляется за вполне реальное время. Но существуют алгоритмы, сложность которых растет быстрее любого полинома, например, ***экспоненциальные*** алгоритмы, реализация которых при больших *n* требует фантастические производительные ресурсы. Так вторая проблема заключается в ответе на вопрос: возможен ли для решения поставленной задачи полиномиальный алгоритм?

Результатом временной сложности алгоритма является его асимптотическая оценка количества задаваемых алгоритмом операций, таких, как функции длины входа, которая коррелирует с асимптотической

оценкой времени выполнения программной реализации алгоритма. Однако асимптотические оценки указывают не более чем порядок роста функции, и результаты сравнения алгоритмов по этим оценкам будут справедливы только при очень больших длинах входов. Для сравнения алгоритмов в диапазоне реальных длин входов, определяемых областью применения программной системы, необходимо знание о точном количестве операций, задаваемых алгоритмом, т.е. его функция трудоемкости.

При вводе в ЭВМ исходные данные кодируются каким–либо способом. Числовые данные обычно представляют в позиционных системах счисления. Символьная информация задаётся в стандартных кодах либо в специальном графическом представлении. Как правило, с точки зрения принципиального решения задачи все способы кодирования эквивалентны. Однако в ряде случаев за счёт более удобной формы представления можно использовать более простой или быстрый алгоритм.

Еще раз отметим, что под *размером задачи* понимают длину последовательности символов, в которой закодированы все исходные данные, и с ростом n размер задачи всегда возрастает. Однако в одних случаях необходимое число символов возрастает быстрее, чем в других. Для оценки поведения функций, зависящих от целочисленных натуральных характеристических параметров, вводят понятие **скорости роста**.

Рассмотрим на множестве натуральных чисел N функции $f(n)$и $g(n)$.

А). Функции $g(n)$ и $f(n)$ имеют ***одинаковую скорость роста***, если при всех достаточно больших n, начиная с некоторого n_0, выполняется условие:

$C_1 f(n) \le g(n) \le C_2 f(n)$, где $C_1>0$, $C_2>0$ – некоторые константы.

Б). Скорость роста функции $g(n)$ ***больше*** скорости роста функции $f(n)$, если для любой сколь угодно большой константы C существует некоторое n_0, начиная с которого выполняется условие:

$$C f(n) \le g(n).$$

В). Скорость роста функции $g(n)$ ***ограничена снизу*** скоростью роста функции $f(n)$, если при всех достаточно больших n, начиная с некоторого n_0, выполняется:

$C f(n) \le g(n)$, где C – константа.

Г). Скорость роста функции $g(n)$ ***ограничена сверху*** скоростью роста функции $f(n)$, если при всех достаточно больших n, начиная с некоторого n_0, выполняется условие:

$g(n) \le C f(n)$, где C – константа.

Данные определения позволяют сравнивать между собой скорости роста функций достаточно произвольного вида – как имеющих, так и не имеющих предел отношения $\lim_{n\to\infty}\frac{g(n)}{f(n)}$.

Замечание. Если скорость роста функции $g(n)$ равна скорости роста функции $f(n)$, то скорость роста $f(n)$ ограничивает скорость роста $g(n)$ и сверху и снизу.

Пример 7.4. Рассмотрим задачу с характерным параметром n, у которой входными данными являются все простые делители числа n, не равные единице. С увеличением n число делителей $k(n)$ будет постоянно изменяться в пределах от 1 (у простого числа) до $log_2 n$ (у степеней 2). Поэтому для длины входа $k(n)$ не существует элементарной монотонной функции с одинаковой скоростью роста. Для неё можно только лишь указать ограничения, верные при любых n: $1 \le k(n) \le log_2 n$.

В общем случае для каждой положительной функции натурального параметра $k(n)$ возможны следующие варианты:

A. Не существует монотонной функции $f(n)$ со скоростью роста, равной $k(n)$, но при всех n, начиная с некоторого n_0, возможны лишь оценки:
$C_н\, f_н(n) \le k(n) \le C_в\, f_в(n)$, где $C_н > 0$, $C_в > 0$ –нижняя и верхняя константы, $f_н(n)$, $f_в(n)$– различные функции.

B. Существует монотонная функция $f(n)$, имеющая одинаковую скорость роста с $k(n)$, для которой по определению при всех n, начиная с некоторого n_0, выполняется: $C_н\, f(n) \le k(n) \le C_в\, f(n)$, где $C_н$, $C_в$ – константы,
В варианте B. предел отношения $k(n)/f(n)$ при $n \to \infty$ может:
а) не существовать и
б) существовать, при этом: $\lim_{n\to\infty}\frac{g(n)}{f(n)} = C_p > 0$.

Случай б) является наиболее распространенным при анализе скорости роста реальных функций натурального параметра. Константу C_p в приведенном выше пределе называют ***константой скорости роста***, а соответствующую скорость роста – ***устойчивой***.

Функции $g(n)$, зависящие от натурального параметра n, могут иметь любой вид, однако скорости их роста $f(n)$ для большей наглядности и упрощения сравнений выражают через модельные элементарные функции (логарифмы, степенные, показательные и др.) либо их произведения. Наиболее часто используют степенные функции n^α. При $\alpha = 1$ скорость роста функции n^α называют ***линейной***, при $\alpha = 2$ – ***квадратичной,*** при α

=3 – ***кубической*** и т.д. Все скорости роста вида n^{α} называют ***полиномиальными***. С увеличением показателя α полиномиальная скорость роста также возрастает.

Существуют функции, скорости роста которых превосходят при $n\to\infty$ любую (со сколь угодно большим показателем α) полиномиальную скорость. Например, 2^n, e^n, $n!$. Скорости такого типа называют ***экспоненциальными***.

Из двух функций $g_1(n)$, $g_2(n)$ с одинаковой устойчивой скоростью роста $f(n)$ меньшие значения в пределе будет иметь функция с меньшей константой скорости роста C_p.

Пример 7.5 Определить скорость роста $f(n)$функции $g_1(n)=(n–5)/3$, константу скорости роста C_p, а также определить пороговое значение параметра n_0 и константы C_1 , C_2, при которых справедливо соотношение:

$$C_1 f(n) \le g(n) \le C_2 f(n).$$

Решение. Так как при $n \to \infty$ предел отношения $g_1(n)/n$ существует и равен 1/3, то скорость роста функции $g_1(n)$ равна линейной полиномиальной $f(n)=n$, а константа скорости роста C_p =1/3.

Определим параметры n_0, C_1, C_2. Для этого используем следующие соотношения:

$$\frac{1}{6}\mathrm{n} \le \frac{\mathrm{n}-5}{3} \le \frac{1}{3}\mathrm{n}.$$

Правое неравенство очевидно и выполняется при любых n, константа (1/6) в левой части задана произвольно из условия (1/6) < (1/3), величину n_0 определим из левого неравенства:

$$\frac{1}{6}\mathrm{n_0} \le \frac{\mathrm{n_0}-5}{3},$$

откуда получаем $\mathrm{n_0} = 10$.

Ответ: $f(n)=n$, C_p =1/3, n_0=10, C_1 = 1/6, C_2 =1/3.

Пример 7.6 Определить скорость роста $f(n)$ функции $g(n)=0{,}25\times(n–1)(n+2)$, константу скорости роста C_p, а также определить пороговое значение параметра n_0, нижнюю и верхнюю константы C_1 , C_2 , при которых справедливо соотношение $C_1 f(n)\le g(n) \le C_2 f(n)$.

Решение. Функция g(n) квадратичная, т.е. скорость ее роста пропорциональна n^2. $\lim_{\mathrm{n}\to\infty} \frac{\mathrm{g(n)}}{\mathrm{n^2}} = \frac{1}{4}$. Следовательно, скорость роста функции g(n) равна квадратичной полиномиальной $f(n)=n^2$ *и* $C_p = \frac{1}{4}$.

Далее находим n_0, C_1, C_2. Для этого используем соотношение:

$$\frac{1}{4}n^2 \le 0{,}25 \times (n-1)(n+2) \le \frac{1}{3}n^2.$$

Определим граничное значение n_0, начиная с которого будут выполняться неравенства в левой и правой частях.

Левая часть: $\frac{1}{4}n_0^2 \le 0{,}25 \times (n_0-1)(n_0+2)$. Отсюда получаем $n_0 = 2$.

Правая часть: $0{,}25 \times (n_0-1)(n_0+2) \le \frac{1}{3}n_0^2$. Но данное неравенство справедливо при любых n_0.

Таким образом, $f(n)=n^2$, $C_p=\frac{1}{4}$, $n_0=2$, $C_1=\frac{1}{4}$, $C_2=\frac{1}{3}$, что является ответом.

7.2.4. Асимптотический анализ трудоемкости алгоритмов

Оценка скорости изменения функции при росте ее аргумента лежит в основе ***асимптотического анализа***, используемого для оценки функции трудоемкости алгоритма, позволяющей определить, как быстро растет трудоёмкость алгоритма с увеличением объема данных.

Для обозначения оценок, позволяющих определить скорость роста функции трудоемкости алгоритма, используются оценки:

- Оценка Θ (тетта);
- Оценка O (о большое);
- Оценка Ω (омега).

Оценка Θ (тетта);

Пусть $f(n)$ и $g(n)$ – положительные функции положительного аргумента, $n \in \mathbb{N}$ и $n \ge 1$, тогда (см. рис. 7.4)

$$\Theta(g(n)) = \begin{cases} f(n), \text{если существуют положительные константы } c_1, c_2, n_0 \\ \text{такие, что } 0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \text{ для всех } n \ge n_0. \end{cases}$$

В этом случае говорят, что при этом функция $g(n)$ является асимптотически точной оценкой функции $f(n)$, т.к. по определению функция $f(n)$, не отличается от функции $g(n)$ с точностью до постоянного множителя.

Заметим, что из $f(n)=\Theta(g(n))$ следует, что $g(n)=\Theta(f(n))$, т.е. функции $f(n)$ и $g(n)$ имеют одинаковую скорость роста.

Пример 7.6

1) $f(n) = 4n^2 + n\ln(n) + 174$, здесь $f(n) \to \Theta(n^2)$;

2) $f(n)=12+5 \cdot n$, здесь $f(n) \to \Theta$

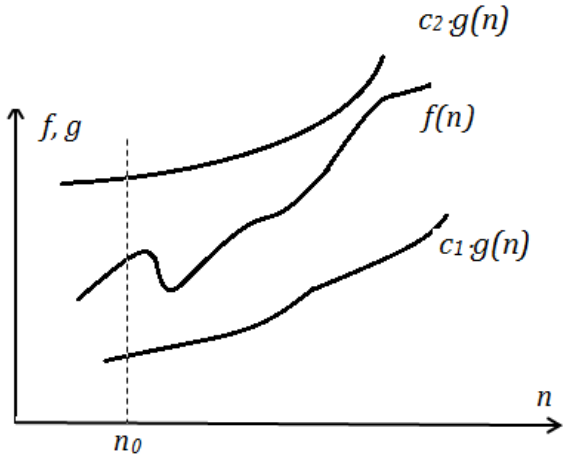


Рис. 7.4

Запись $f(n) \to \Theta(n^2)$ (или $f(n) = \Theta(n^2)$) для примера 1) означает, что функция временной сложности алгоритма $f(n)$ от величины входного слова n растет не быстрее квадратичной функции, а для примера 2) $f(n)$ или равна константе, не равной нулю, или ограничена константой $\Theta(1)$.

Пример 7.7

Пусть $c_1 \cdot n^2 \le \frac{1}{2}n^2 - 3n \le c_2 \cdot n^2$, откуда получаем $c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$.

Неравенство выполняется, если:

$n \ge 7, \quad c_1 \le \frac{2}{7}$,

$n \to \infty, \quad c_2 \ge \frac{1}{2}$.

Таким образом:

$$\frac{2}{7}n^2 \le \frac{n^2}{2} - 3n \le \frac{1}{2}n^2.$$

Оценка *O* (О большое)

В отличие от оценки Θ, оценка O требует только, чтобы функция $f(n)$ не превышала $g(n)$ начиная с $n>n_0$, с точностью до постоянного множителя (рис.7.5).

$$O(g(n)) = \begin{cases} f(n), \text{если существуют положительные константы c, и } n_0 \\ \quad \text{такие, что } 0 \le f(n) \le c \cdot g(n) \text{ для всех } n \ge n_0. \end{cases}$$

Вообще, запись $O(g(n))$ обозначает класс функций, таких, что все они растут не быстрее, чем функция $g(n)$ с точностью до постоянного множителя, и поэтому иногда говорят, что $g(n)$ мажорирует функцию $f(n)$.

Например, для функций $f(n) = \frac{1}{n}$, $f(n) = 22$, $f(n) = 5n + 27$, $f(n) = n \cdot \ln(n)$, $f(n) = 6n^2 + 24n + 77$ справедлива оценка $O(n^2)$.

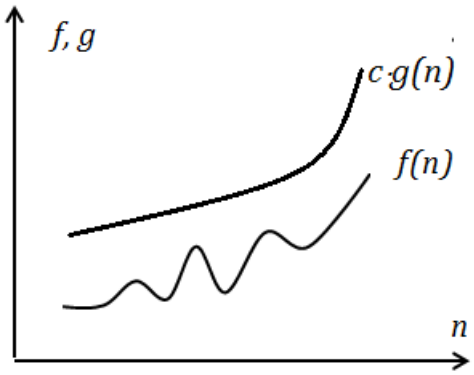


Рис. 7.5 Оценка O

Оценка Ω (Омега).

В отличие от оценки O, оценка Ω является оценкой снизу, т.е. определяет класс функций, которые растут не медленнее, чем $g(n)$ с точностью до постоянного множителя (рис. 7.6).

$$\Omega(g(n)) = \begin{cases} f(n), \text{если существуют положительные константы c, и } n_0 \\ \text{такие, что } 0 \le c \cdot g(n) \le f(n) c \cdot g(n) \text{ для всех } n \ge n_0. \end{cases}$$

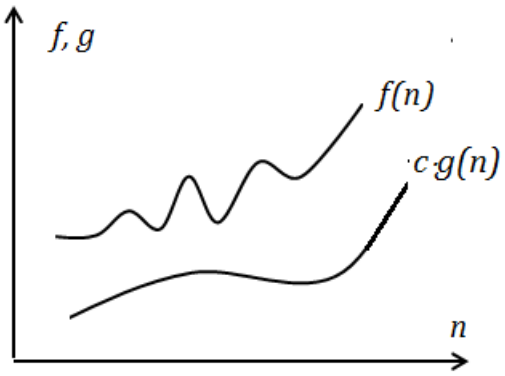
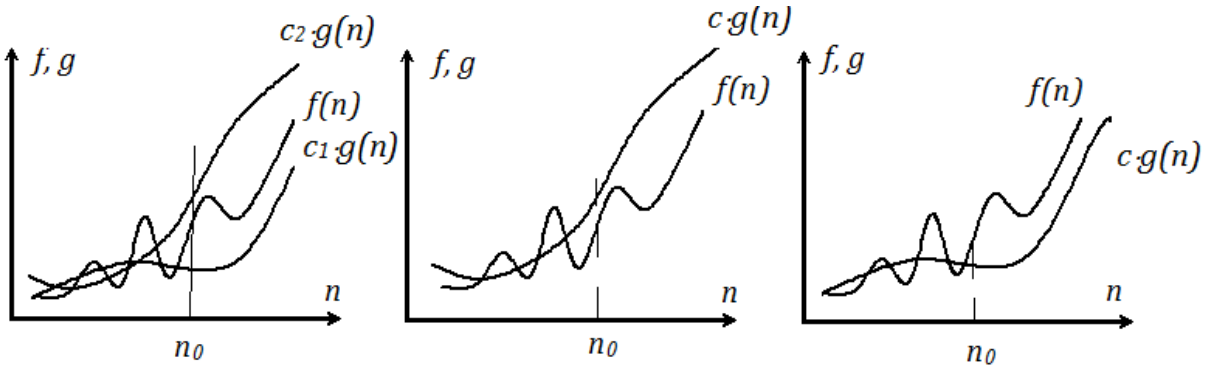


Рис. 7.6 Оценка Ω

Например, запись $\Omega(n \cdot \ln(n))$ обозначает класс функций, которые растут не медленнее, чем $g(n) = n \cdot \ln(n)$. В этот класс попадают все полиномы степени $n > 2$ и все степенные функции с основанием большим единицы.

Для любых двух функций $f(n)$ и $g(n)$ соотношение $f(n) = \Theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$ (рис. 7.7). В качестве n_0 выбирается такое минимальное его значение, при котором при любом $n > n_0$ соответствующая оценка также будет выполниться.

Есть примеры, когда для пары функций не выполняется ни одно из асимптотических соотношений. Например, $f(n) = n^{1+\sin(n)}$ и $g(n) = n$.



а) $f(n)= \Theta(g(n))$ б) $f(n) = O(g(n))$ в) $f(n) = \Omega\ (g(n))$

Рис. 7.7 Графические примеры обозначений Θ, O и Ω оценок

Итак, из вышеизложенного следует, что оценка $\boldsymbol{O}$ (О большое, или Big O), дает верхнюю асимптотическую границу, а большая Ω дает нижнюю асимптотическую границу. А вот оценка Θ (тета) дает как верхнюю, так и нижнюю границы. Простыми словами:

- О большое (O()) описывает верхнюю границу сложности;
- Омега (Ω ()) описывает нижнюю границу сложности;
- Тета (Θ ()) описывает точную оценку сложности.

Например, функция $g(n) = n^2 + 3n$ – это $O(n^3)$, $\Theta\ (n^2)$, $\Omega(n)$. Но правильно будет и то, что это есть $\Omega\ (n^2)$ или $O(n^2)$. Вообще, когда говорят об оценке O, то на самом деле имеют в виду оценку Θ. Ведь не имеет смысла определять верхнюю границу, намного превышающую объем анализа. Поэтому, если учитывать только доминирующий терм функции, то нужно учитывать оценку O для конкретной функции $g(n)$. Доминирующий терм (оператор) функции – это такой терм, который растет быстрее всего. Например, если n^2 растет быстрее, чем n, поэтому, если функция $g(n)$ имеет вид: $g(n)=n^2+10n+15$, то оценка O будет (n^2).

Вычислить точную оценку и трудно, да и бессмысленно. Для практического понимания сложности алгоритма достаточно оценить время выполнения с помощью O-символики, что, как правило, и используется на практике. В зависимости от входных данных, алгоритм может выполняться различное время. Обычно оценивается средняя сложность и сложность в худшем случае. Так же есть зависимость от количества входных данных n. Обычно оценивается именно порядок роста от n. Так, например, чтение данных и сохранение их в памяти в виде массива будет иметь сложность $O(n)$, или линейную сложность, а умножение матриц уже кубическую $O(n^3)$.

Для временной оценки сложности алгоритма *T(n)* наиболее часто используются функции:

- (1) – константное (постоянное) время;
- ($log\ n$) - логарифмическое время;
- (n) - линейное время;
- ($n \cdot log\ n$) – линейно-логарифмическое время;
- (n^k) - полиноминальное время;
- (2^n) - экспоненциальное время;
- ($n!$) - факториальное время.

Первые пять функций имеют невысокую скорость роста и алгоритмы, время работы которых оценивается этими функциями, можно считать быстродействующими. Скорость роста экспоненциальной функции иногда характеризуют как «взрывную». Для сравнения допустим, что имеются алгоритмы, трудоемкость которых (число операций) достаточно точно отражается этими функциями. Пусть эти алгоритмы выполняются на компьютере, производительность которого составляет 10Гфлопс. При длине входа $n \leq 100000$ алгоритмы, скорость работы которых оценивается первыми четырьмя функциями, получат ответ за ничтожные доли секунды. Для алгоритма с трудоемкостью 2^n время работы оценивается следующим образом:

- $n = 50 \approx 2$ минуты,
- $n = 60 \approx 32$ часа,
- $n = 70 \approx 3{,}7$ года.

Ну а при реализации алгоритма с трудоемкостью $n!$ время его работы составит чудовищно большие значения! Например, даже при относительно небольшой длине входа $n = 50$ время работы составит $3 \cdot 10^{51}$сек., что составит примерно 10^{44} лет! Более, чем фантастика!

Сразу отметим, что функция $n!$ растет быстрее, чем 2^n.

Существуют и другие функции оценки временной сложности алгоритма *T(n)*, как, например:

- ($log^k n$) – полилогарифмическое время;
- ($n \cdot log^k n$) – квазилинейное время.

Знание асимптотики поведения функции трудоемкости алгоритма (сложности), дает возможность делать прогнозы по выбору более рационального с этой точки зрения алгоритма для больших размерностей исходных данных.

Часто приходится сталкиваться с показателями времени, выраженных в секундах. В следующей таблице содержится перевод в другие единицы измерения времени.

Время в секундах	Другие единицы
10^2	1.7 мин.
10^3	16.7 мин.
10^4	2.8 часа
10^5	1.1 дня
10^6	1.6 недели
10^7	3.8 мес.
10^8	3.17 года
10^9	31.7 года
10^{10}	317 лет
10^{11}	3170 лет

7.2.4. Классы сложности алгоритма

В теории алгоритмов множество вычислительных задач разбиваются на ***классы сложности***, каждый их которых определяется примерно одинаковым критерием сложности вычисления.

Конечно, кроме временной сложности алгоритма, важной оказывается так же пространственная (объемная) сложность алгоритма, которая определяется количеством дополнительной памяти S, которое алгоритм требует для работы. Память M, необходимая для хранения входных данных, не включается в S. S в общем случае тоже зависит от исполнительного устройства. Скажем, если два исполнительных устройства поддерживают целые длинной 8 и 16 байт соответственно, то пространственная сложность алгоритма на 16-байтных целых будет вдвое больше, чем на 8-байтных целых. Поэтому пространственная сложность оценивается так же порядком роста.

Выделяют следующие основные классы сложности их определения:

DTIME

Машина Тьюринга находит решение задачи за конечное время (количество шагов). Часто уточняется асимптотика алгоритма, так, скажем, если порядок роста времени работы $T(n)=O(f(n))$, то указывают $DTIME(f(n))$.

P

Машина Тьюринга находит решение задачи за полиномиальное время (количество шагов), т.е . $T(n)=O(n^k)$, *где* $k \in N$. $P=DTIME(n^k)$

EXPTIME

Машина Тьюринга находит решение задачи за экспоненциальное время (количество шагов), т.е. $T(n)=O(2^{n^k})$, где $k \in N$. $EXPTIME=DTIME(2^{n^k})$.

DSPACE

Машина Тьюринга находит решение задачи, используя конечное количество дополнительной памяти (ячеек). Часто уточняется асимптотика алгоритма, так, скажем, если порядок роста потребления памяти $S(n)=O(f(n))$, то указывают $SPACE(f(n))$.

L

Машина Тьюринга находит решение задачи с логарифмической пространственной сложностью, то есть $S(n)=O(\log n)$. $L=DSPACE(\log n)$.

PSPACE

Машина Тьюринга находит решение задачи с полиномиальной пространственной сложностью, то есть $S(n)=O(n^k)$, где $k \in N$ $PSPACE=DSPACE(n^k)$.

EXPSPACE

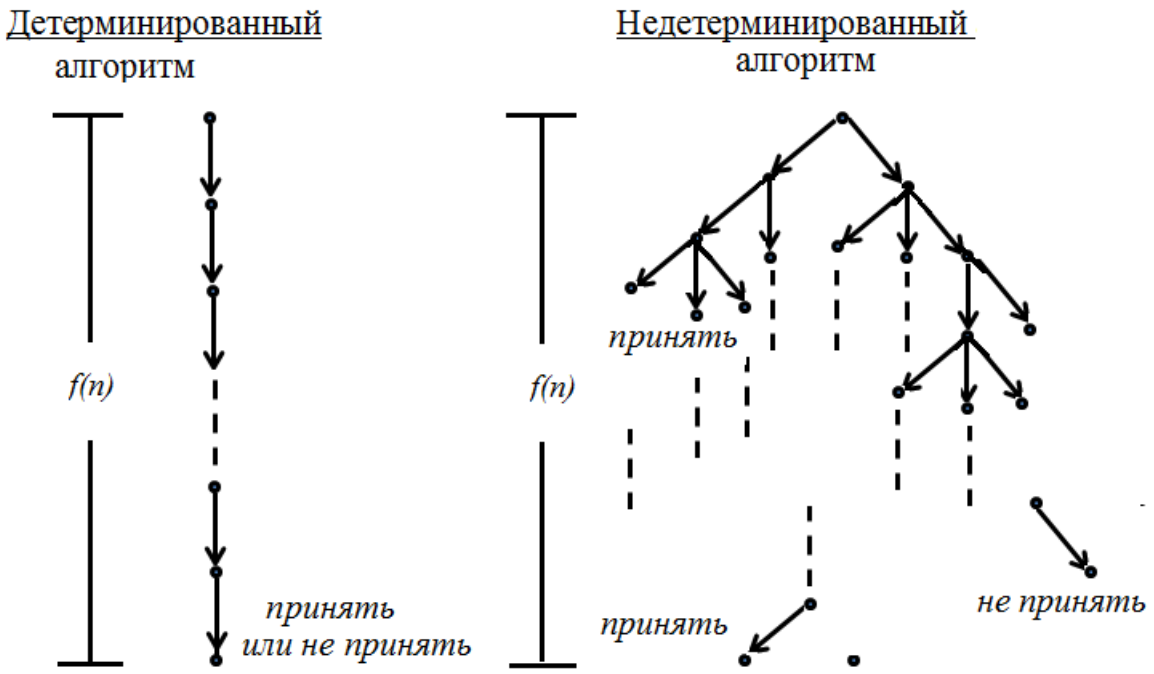
Машина Тьюринга находит решение задачи с экспоненциальной пространственной сложностью, то есть $S(n)=O(2^{n^k})$, где $k \in N$ $EXPSPAC=DSPACE(2^{n^k})$.

Как известно, одним из характерных свойств алгоритма является его ***детерминированность,*** означающая, что система величин, получаемых в какой-то момент времени, однозначно определяется системой величин, полученных в предшествующие моменты времени. В рамках машины Тьюринга, уточняющей понятие алгоритма, детерминированность означает, что новое состояние, в которое машина переходит на очередном шаге, полностью определяется текущим состоянием и тем символом, который обозревает головка на ленте.

Но существует и другой класс алгоритмов – ***недетерминированные***[1] алгоритмы, которые характеризуются тем, что обработка одних и тех же входных данных может приводить как к одинаковым, так и разным результатам, т.е. недетерминированный алгоритм отличается от более известного детерминированного аналога своей способностью достигать результатов, используя различные пути. Если детерминированный алгоритм представляет единственный путь от входа к результату,

[1] Это понятие ввел Роберт У. Флойд в 1967 году.

недетерминированный алгоритм представляет один путь, ведущий ко многим путям, некоторые из которых могут приводить к одному и тому же результату, а некоторые - к “уникальным” результатам (Рис. 7.8).



а) б)

Рис. 7.8

Детерминированный алгоритм, который выполняет *f(n)* шагов, всегда завершается за *f(n)* шагов и всегда возвращает тот же результат (Рис. 7.8 а). Недетерминированный алгоритм с уровнями *f(n)* может не возвращать одинаковый результат при разных прогонах, а также может никогда не закончиться из-за потенциально бесконечного размера дерева с фиксированной высотой (Рис.7.8 б).

Недетерминированные алгоритмы могут быть также смоделированы машинами Тьюринга, в которых для каждого состояния может быть несколько следующих состояний в соответствии с функцией перехода:

$$g_i a_j = \begin{cases} g_{i_1} a_{j_1} d_{k_1} \\ g_{i_2} a_{j_2} d_{k_2} \\ \cdots \\ g_{i_m} a_{j_m} d_{k_m} \end{cases},$$

где m – это максимальное число вариантов выполнения действия. Если для некоторых внутренних состояний g_i и читаемых символов a_i вариантов меньше, то последний будет дублироваться так, чтобы их стало m.

Перед выполнением очередного действия из одной машины возникает m машин. Записи на их лентах одинаковы – такие, как были у их “предка”,

но с этого момента их пути расходятся: каждая машина выполняет свой вариант из списка. Если процесс работы детерминированной машины Тьюринга может быть изображен в виде линейной последовательности действий, как показано на рис. 7.8а), то процесс работы недетерминированной машины Тьюринга может быть изображен в виде дерева, вершинами которого являются выполняемые действия, а ребрами – переходы от одного действия к другому (рис. 7.8 б). Таким образом, она как бы решает задачу разными способами. Если по некоторой цепи действия заканчиваются, то возможны два исхода:

- задача решена - управляющее устройство перешло в заключительное состояние g_y (от слова "*yes*"-"да");
- решение не найдено – управляющее устройство перешло в состояние g_n (от слова "*no*" – "*нет*").

В первом случае все остальные машины, размножившиеся к этому моменту, также прекращают работу и "превращаются" в одну машину. И в этом случае говорят, что вычисление было *принимающим* (*accept*).

Во втором случае все машины продолжают работу.

Если же по всем цепям произойдет окончание вычисления с отрицательными результатами, то все машины тоже "сливаются" в одну – задача не имеет решения. В этом случае говорят, что вычисление *непринимающее* (*reject*).

Т.е. недетерминированная машина Тьюринга это чисто теоретическое построение, которое по принципам действия аналогично детерминированной машине Тьюринга, с тем отличием, что для каждого из состояний может быть несколько возможных действий. При этом, недетерминированная машина Тьюринга всегда выбирает из возможных действий то, которое приводит к решению за минимально возможное число шагов. Эквивалентно недетерминированная машина Тьюринга производит вычисления всех ветвей и выбирает ту ветвь, которая приводит к решению за минимально возможно число шагов. Только вот главный вопрос и состоит в том – как найти эту "ветвь", не проводя вычислений по другим?

Классы сложности недетерминированных алгоритмов в соответствии с их определением совпадают с вышеприведенными, а названия имеют префикс *N* (например, *NP*), кроме *NTIME* и *NSPACE*, где *D* заменяется на *N*.

Существует теорема доказывающая, что полиномиальные недетерминированные алгоритмы определенно оказываются более

мощными, чем полиномиальные детерминированные, и любая программа, выполненная на недетерминированной машине Тьюринга за t шагов, может быть выполнена на детерминированной машине Тьюринга за число шагов, экспоненциально зависящее от t.

В общем случае известно, что

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq NEXPTIME \subseteq EXPSPACE.$$

Кроме того, $P \not\subseteq EXPTIME$, $NP \not\subseteq NEXPTIME$, $PSPACE \not\subseteq EXPSPACE$.

Так же известно, что если $P = NP$, то $EXPTIME = NEXPTIME$. Но вопрос равенства P и NP является одним из главных нерешенных вопросов современной информатики.

Рассмотрим пример. Пусть мы имеем компьютер, выполняющий 10^7 операций в секунду. Рассмотрим два алгоритма: один выполняется за n^5 операций, а второй за 2^n, где n - длина входных данных. Возьмем, например, исходные данные длиной 50. Первый алгоритм выполнится ~ за 0.5 минуты, а для второго потребуется 3.3 года! Таким же образом соотносятся и алгоритмы из классов P и NP.

Несмотря на то, что задачи из класса NP решаются гораздо дольше, до сих пор не доказано, что эти задачи невозможно решить за полиномиальное время на детерминированной машине Тьюринга. Другими словами, неизвестно, совпадают ли классы P и NP.

Итак, задачи, которые можно решить относительно быстро за полиноминальное время, определяют как ***P класс***. К этому классу относятся задачи линейной сложности, как, например, задача последовательного поиска: при удлинении списка данных вдвое алгоритм работает вдвое дольше. В алгоритмах последовательного поиска нас интересует процесс просмотра списка в поисках некоторого элемента, называемого целевым. При последовательном поиске предполагается, что список не отсортирован. Например, ключевое значение может быть, фамилией, номером, значением или любым другим уникальным идентификатором. Алгоритм последовательного поиска последовательно просматривает по одному элементу списка, начиная с первого, до тех пор пока не найдет нужный элемент. Очевидно, что чем дальше в списке находится конкретное значение ключа, тем больше времени уйдет на его поиск, т.е. временная сложность определяется как $O(n)$.

Другим примером задач этого класса может служить задача сортировки массива. В зависимости от выбранного алгоритма сортировки, для массива длиной n, количество выполненных операций будет от $O(n \cdot log(n))$ до $O(n^2)$.

Таким образом, простота алгоритмов, принадлежащих к классу *P* заключается в том, что при увеличении размера задачи, время выполнения увеличивается относительно незначительно.

Кроме практически разрешимых задач, относящихся к *P* классу, существует и другой класс задач, которые практически неразрешимы и мы не знаем алгоритмов, способных решить их за разумное время. Эти задачи образуют *NP* класс, класс задач недетерминированной полиномиальной сложности. Следует отметить, что сложность всех известных детерминированных алгоритмов, решающих задачи *NP* класса, либо экспоненциальна, либо факториальна. В общем случае ***классом NP*** (от анг. non- deterministic polynomial) называют множество задач, решение которых при наличии некоторых дополнительных сведений можно решить за полиноминальное время.

Для каждого класса существует категория задач, которые являются «самыми сложными». Это означает, что любая задача из класса сводится к такой задаче, и при том сама задача лежит в классе. Такие задачи называются полными задачами данного класса. Наиболее известными являются ***NP –полные задачи,*** которые можно рассматривать как задачи, решение которых мы хотим найти за оптимальное время.

NP –полные задача – задача из класса *NP*, к которой можно свести любую другую задачу из класса *NP* за полиноминальное время. Таким образом, *NP*-полные задачи образуют в некотором смысле подмножество «самых сложных задач» в классе *NP*, и если для какой-то из них будет найден «быстрый» алгоритм решения, то и любая задача из класса *NP* может быть решена так же «быстро».

Как видим, полные задачи являются хорошим инструментом для доказательства равенства классов. Достаточно для одной такой задачи предоставить алгоритм её решения и принадлежащий более «маленькому» классу, и равенство будет доказано.

Тогда равенство ***P*** и ***NP*** означает, что любую поставленную задачу можно быстро решить, и в этом случае почти любой процесс можно будет автоматизировать. Неравенство же ***P*** и ***NP***, в свою очередь, означает, что для некоторых задач быстрое решение не найдется никогда, и отнимает всякую надежду на создание универсального алгоритма. Впрочем, это еще не повод опускать руки: для борьбы с «крепкими орешками» разрабатываются специальные методы, которые во многих случаях работают вполне приемлемо.

Для примера рассмотрим известную задачу быстрого поиска кратчайшего маршрута коммивояжера, которая формулируется следующим образом: есть конечный набор городов $C = \{c_1, c_2, \ldots, c_n\}$ и расстояний между ними $d(c_i, c_j)$. Необходимо найти упорядоченный набор этих городов $(c_{a_1}, c_{a_2}, \cdots, c_{a_n})$ такой, чтобы $\sum_i^n d(c_{a_i}, c_{a_{i+1}}) \rightarrow min$. Иными словами, нужно найти кратчайший путь обхода всех городов без повторного посещения.

Если, например, необходимо решить данную задачу для 5 городов, то надо рассмотреть 5! вариантов. Если мы имеем в распоряжении фантастический компьютер с производительностью 100 эксафлопс (возможное их появление планируется не раньше 2030 года, но поживем-увидим!), то время счета составит порядка 10^{-16}сек. Если же количество городов будет, например, 50, то количество вариантов n будет равно: 30414093201713378043612608166064768844377641568960512000000000000 и требуемое время счета перебором составит ~$3 \cdot 10^{44}$сек. или 10^{36}лет! (в то время как наша вселенная существует примерно 10^{10}лет).

Так неужели для поиска оптимального пути среди всех возможных маршрутов нет способа получше? Можно ли найти кратчайший путь, не просматривая их все? До сих пор никому не удалось придумать алгоритм, который не занимается, по существу, просмотром всех путей. Когда число городов невелико, задача решается быстро, однако это не означает, что так будет всегда, а нас как раз интересует решение общей задачи.

Проиллюстрируем данную задачу конкретным примером. Пусть мы имеем 4 города, соединенные дорогами, как показано на рис. 7.9.

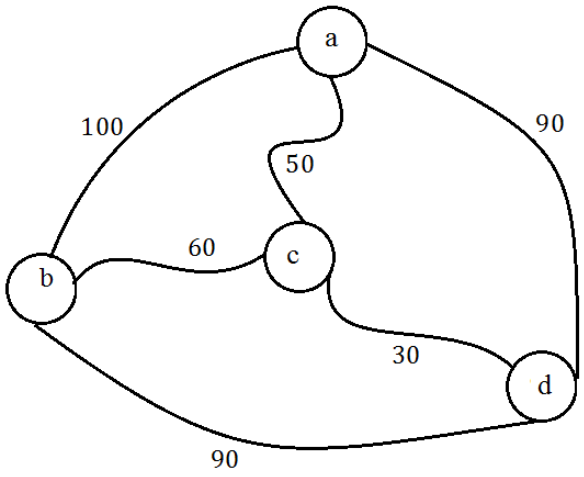


Рис. 7.9

Попробуйте написать алгоритм поиска минимальных путей обхода имеющихся городов, начиная с любого пункта. Надо рассмотреть 24 варианта. Кратчайший путь для данной задачи составляет 270, который

можно построить начиная с любого города. Например, (a-b-d-c-a), (b-a-d-c-b), (c-d-b-a-c), (d-c-a-b-d).

Вот мы и подошли при рассмотрении примера к сути дела. Вопрос о равенстве классов ***P*** и ***NP*** самым непосредственным образом связан с задачей быстрого поиска кратчайшего маршрута коммивояжера (и не только с ней). Названия классов – сокращения от технических терминов, однако будет лучше воспринимать их просто как общие понятия, а не как конкретные математические объекты. Класс ***NP*** – это множество задач, которые мы хотим решить; класс ***P*** – задачи, которые мы умеем решать быстро. Если ***P*** равно ***NP***, мы всегда сможем быстро найти решение любой ***NP***-задачи (например, кратчайший маршрут для коммивояжера). А если не равно, то не сможем.

Задача о коммивояжере, конечно, очень похожа на задачи про графы. Каждый город можно представить вершиной графа, наличие пути между двумя городами - ребром, расстояние между ними - весом этого ребра. Отсюда можно сделать вывод, что алгоритм поиска кратчайшего пути решает и задачу коммивояжера, однако это не так. Какие два условия задачи о коммивояжере отличают ее от задачи о кратчайшем пути? Во-первых, мы должны посетить все города, а алгоритм поиска кратчайшего пути дает лишь путь между двумя заданными городами. Второе отличие состоит в требовании возвращения в исходную точку, которое отсутствует в поиске кратчайшего пути.

Поиск кратчайшего пути не охватывает все аспекты проблемы равенства ***P*** и ***NP***. Задача коммивояжера доказывает, что при наличии огромного числа вариантов совсем не обязательно перебирать их все; главный вопрос, однако, заключается в том, всегда ли можно обойтись без такого перебора?

Рассмотрим еще один пример – задачу о сумме подмножеств, которая формулируется следующим образом: «в заданном множестве целых чисел, имеется ли хотя бы одно непустое подмножество сумма элементов которого равняется нулю?» Например, пусть нам дано множество {5, -2, 17, 10, -4, -8}. Для него ответ на поставленный вопрос является положительным, т.е.-"*да*", поскольку искомое подмножество существует: {-2+10-8=0}. Легко убедиться, что проверка решения осуществляется быстро (нужно лишь просуммировать элементы найденного подмножества). Но на поиск решения необходимо затратить экспоненциальное время, поскольку необходимо проверить все возможные подмножества. Количество возможных комбинаций составляет 2^n для входного слова длины *n*. Это делает решение поставленной задачи

непрактичным для больших значений n. Действительно, даже при нашей гипотетически существующей мощности ЭВМ в 100 эксафлопс и длине входного слова, например, n=100, время счета составить не менее 300 лет!

Итак, проблема ***P*** и ***NP*** касается не только описанных выше задач, но и тысяч других, схожих с ними по сути. Наиболее известными примерами *NP*-задач являются:

1. Задача о выполнимости булевых формул;
2. Кратчайшее решение «пятнашек» размера n × n;
3. Задача коммивояжёра;
4. Проблема раскраски графа;
5. Задача о вершинном покрытии;
6. Задача о покрытии множества;
7. Задача о клике;
8. Задача о независимом множестве;
9. Задача о рюкзаке;
10. Сапер (игра);
11. Тетрис.

Насколько быстро можно перебрать огромное число потенциальных вариантов? Насколько трудно будет отыскать оптимальное решение поставленной задачи?

Немного истории. Впервые проблема равенства классов упоминается еще в 1956 году – в письме, которое один величайший математик XX века, Курт Гёдель, отправил другому величайшему математику XX века, Джону фон Нейману. К сожалению, вплоть до восьмидесятых о письме ничего не было известно, а вот первые официальные публикации появились в начале семидесятых, и к 1971 году эта проблема математически была сформулирована. Авторы – Стивен Кук и Леонид Левин – независимо друг от друга пришли к одному и тому же вопросу. Вслед за этим Ричард Карп опубликовал свой знаменитый список из двадцати одной задачи: все они, включая маршрут для коммивояжера и разбиение на группы, были эквивалентны проблеме «P против NP». Постепенно научное сообщество осознало важность поднятых вопросов, и в развитии информатики наступил поворотный момент. Сейчас проблема равенства классов уже стала основополагающей – причем не только в информатике, но также в биологии, медицине, экономике, физике и многих других областях.

Со временем этот вопрос заработал статус одной из самых трудных задач в истории математики.

В 2000 году был опубликовал список из семи «задач тысячелетия»:

1. Гипотеза Берча–Свиннертон–Дайера.
2. Гипотеза Ходжа.
3. Уравнения Навье–Стокса.
4. Проблема равенства P и NP.
5. Гипотеза Пуанкаре.
6. Гипотеза Римана.
7. Теория Янга–Миллса.

Как мы знаем, гипотезу Пуанкаре в 2003 году доказал наш соотечественник Григорий Перельман, а остальные шесть задач тысячелетия по-прежнему остаются открытыми.

С какой стороны зайти, чтобы вывести неравенство $P \neq NP$? Курт Гёдель показал, что не у всех математических проблем имеется решение; возможно, аналогичным образом удастся доказать тот факт, что не для всех поисковых задач существует быстрый алгоритм. Еще вариант – попытаться разделить вычислительный процесс на более мелкие части, чтобы сложность исходной задачи было легче оценить. Некоторую надежду дает также алгебраическая геометрия – молодой и абсолютно абстрактный раздел математики. Впрочем, до решения проблемы мы в любом случае дойдем еще очень не скоро.

Изменят ли ситуацию компьютеры будущего, основанные на принципах квантовой механики? Снимут ли они проблему «P против NP»? Иногда можно услышать, что квантовые компьютеры являются реализацией недетерминированной машины Тьюринга. Даже предположив, что это может казаться верным в некоторых случаях, но все-таки в общем случае недетерминированная машина Тьюринга является более мощной системой, чем квантовый компьютер.

Но как же решать подобные «трудные задачи» сегодня? Один из способов решения задач состоит в том, чтобы свести, или редуцировать, одну задачу к другой. Тогда алгоритм решения второй задачи можно преобразовать таким образом, чтобы он решал первую. Если преобразование выполняется за полиномиальное время и вторая задача решается за полиномиальное время, то и наша новая задача также решается за полиномиальное время. Но не для всех задач можно использовать метод сведения одной задачи к другой. Конечно, в какой-то степени поиск, совершенствование и развитие вычислительных структур, комплексов, технологий будет способствовать и построению алгоритмов,

обеспечивающих приемлемое время решение «трудных задач» с приемлемой точностью. Но это все вопросы будущего….

Литература

1. В.И. Игошин. Математическая логика и теория алгоритмов. М., Академия, 2008г.
2. Д.Ш. Матрос, Г.Б. Поднебесова. Теория алгоритмов, М., Бином, 2008г.
3. Минский М. Вычисления и автоматы. М., Мир, 1971г.
4. С.Д. Шапорев. Математическая логика. С-т Петербург, БХВ-Петербург, 2007г.
5. Л.А. Гладков, В.В. Курейчик, В.М. Курейчик. Дискретная математика. М., Физматлит, 2014г.
6. Крупский В.П., Плиско В.Е. Теория алгоритмов. М., Академия, 2009г.