Глава 1. Введение в теорию алгоритмов

Алгоритм – точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату.

ГОСТ 19781-74

Всё должно быть изложено так просто, как только возможно, но не проще.

Альберт Эйнштейн

Во все времена деятельность человека была неразрывно связана с решением встающих перед ним разнообразных задач. Узнать встреченного на улице человека, рассчитать длительность предстоящей работы, сходить в магазин за хлебом, построить дом и т.д. Одним из основных способов решения задач и является создание *алгоритма* её решения. Отметим, что почти всегда человек не осознаёт, что он решает задачу в соответствии с некоторым алгоритмом.

Сейчас нам известны такие способы решения задач:

1). алгоритмический (дискретный), цифровые ЭВМ;

2). непрерывный (интуитивный) для человека, аналоговые ЭВМ, нейронные сети;

3). квантово-механический (квантовые ЭВМ);

4). метод само построения решения (метод самоорганизации). Последним способом решается, например, задача развития из зародышевой клетки взрослого организма, используя информацию, записанную в ДНК (так называемый эпигенéз). Более примитивная самоорганизация в химии называется синергетикой.

Как сходить и купить хлеб в магазине? Нужно взять сумку, не забыть взять деньги (ну, или вместо денег взять кредитную карту или смартфон с соответствующим приложением, а скоро достаточно будет при покупке предъявить в объектив на кассе своё лицо). Выглянуть в окно, если идёт дождь, то взять зонтик. Выйти из дома к дороге, если близко автобус, то подъехать одну остановку до магазина, иначе идти пешком и т.д. А если вдруг окажется, что в магазине обеденный перерыв, то это исключительная ситуация .

Конечно, с точки зрения программистов это не совсем «строгий» алгоритм. Например, брать ли зонт, если дождя нет, но он может скоро пойти? Или, наоборот, дождь ещё капает, но скоро кончится и небо ясное? Впрочем, в отличие от компьютера, человек уверенно преодолевает эти трудности. Действительно, когда человек вышел из квартиры, то у него однозначно либо есть с собой зонт, либо нет (и он может об этом вскоре пожалеть).

Так что же такое алгоритм?

1.1. Интуитивное понятие алгоритма

Алгоритм – конечный упорядоченный набор чётко определённых правил для решения проблемы.

ISO 2382/1-93

…человек, что бы он ни делал, почти никогда не знает, что именно он делает, во всяком случае, не знает до конца.

Станúслав Лем. «Сумма технологии»

У каждого из нас есть интуитивное понимание того, что такое алгоритм. Обычно говорят, что это чёткая (строгая) система правил (шагов, этапов, действий), последовательно выполняя которые можно решить поставленную задачу. Многие алгоритмы требуют для своей работы определённых входных данных (или материалов). Например, описанный в женском журнале алгоритм, как связать модный шарф, требует «на входе» спицы подходящего размера и шерсть нужного цвета и толщины.

Далее, после некоторого раздумья, можно понять, что для решения задачи главным является не сам алгоритм, а исполнитель этого алгоритма (algorithm performer), тот, кто собственно и выполняет все шаги алгоритма. Обычно говорят, что некоторый алгоритм получает (вводит) свои входные данные, но на самом деле главным является исполнитель, именно он и вводит данные (правда, подчиня-

ясь заданным а алгоритме указаниям). В информатике, однако, принято (не совсем верно) говорить «алгоритм вводит данные», «алгоритм останавливается» и т.д. и мы тоже будем так говорить, но «в уме» будем знать, что всё это делает именно исполнитель.

Особо подчеркнём тесную (неразрывную) связь между понятиями алгоритма и исполнителя. Не существует алгоритма без исполнителя, который будет его выполнять, и не существует исполнителя без определения типа алгоритмов, которые он сможет выполнить. Одна и та же запись (например, описание в женском журнале, как связать модный шарф), если Вы умеете вязать, будет для Вас алгоритмом, а если не умеете, то не будет! Аналогично, программа на машинном языке некоторой ЭВМ будет для неё алгоритмом, а для другого компьютера, с иным машинным языком, не будет [i] [см. сноску в конце главы]. Исходя из этого, можно построить такую схему решения задачи по заданному алгоритму (рис. 1.1).

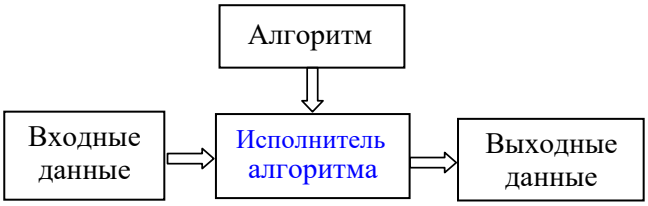


Рис. 1.1. Схема работы алгоритма.

Сначала отметим, что исполнитель алгоритма получает на вход не сам алгоритм (это абстрактное понятие), а запись алгоритма, т.е. конечный текст на некотором языке, описывающий все шаги этого алгоритма.[1] Для исполнителя зафиксированы все действия, которые он может делать, выполняя алгоритм, ясно, что только эти действия и могут входить в запись алгоритма.

Рассмотрим более подробно, как всё это работает. Итак, исполнитель получает запись некоторого алгоритма и начинает последовательно выполнять шаги этого алгоритма, после выполнения каждого шага однозначно определён следующий шаг. При необходимости исполнитель производит ввод входных данных и вывод результатов. Всё это называется вычислительным процессом. Слово «вычислительный» осталось с тех времён, когда компьютерные алгоритмы, в основном, обрабатывали числовые данные. Сейчас, конечно, основной объём обрабатываемых данных представляет из себя всевозможные тексты, базы данных, записи изображения и звука (как говорят, мультимедиа) и т.п., но термин «вычислительный процесс» остался.

Бывают случаи, когда алгоритм очень тесно связан с исполнителем (встроен в него), его нельзя менять. Например, в микропроцессор простых электронных часов встроен алгоритм, который считает входные импульсы тактового генератора и преобразует их в секунды, минуты и т.д. вплоть до определения високосных лет. Такие исполнители нам будут неинтересны, нам важен случай, когда на вход исполнителя можно подавать много разных (записей) алгоритмов.

Вообще говоря, ничего не мешает исполнителю брать на выполнение сразу несколько последовательных шагов алгоритма, если выходные данные одного шага не являются входными данными для следующего шага. Например, для алгоритма похода в магазин шаги а) взять деньги; б) взять сумку; можно делать одновременно (двумя руками). Отметим, что именно так и работают современные компьютеры, они могут брать на выполнение сразу до нескольких десятков команд программы.

Каждый исполнитель знает правила, как запускать вычислительный процесс, как его проводить (шаг за шагом) и как останавливать (завершать) этот процесс с выдачей ответа. Все эти правила, сам исполнитель и способы записи алгоритма определяют, как говорят, конкретную алгоритмическую систему.[2] Итак, исполнитель запускает вычислительный процесс для заданного ему алгоритма с некоторыми входными данными, дальше возможны два случая.

1. После выполнения конечного числа шагов исполнитель остановится и выдаст ответ. В этом случае говорят, что (этот) алгоритм применим (applicable) к (этим) входным данным. Все такие данные образуют *область применимости* алгоритма.

---

[1] Конечно, для некоторых исполнителей (сейчас, в основном, для людей и немного для нейросетей) возможна и «звуковая запись» алгоритма в виде его устного описания или в графическом виде (блок-схемы, наборы картинок и т.д.).

[2] Для алгоритмической системы можно дать строгое (формальное) определение, но оно нам сейчас не важно, достаточно будет и такого объяснения «на пальцах».

2. Когда первый пункт не выполняется, то говорят, что алгоритм не применим к (этим) входным данным. Здесь, в свою очередь, можно выделить два случая.
   1) исполнитель никогда не остановится, как говорят программисты, алгоритм *зациклится*;
   2) исполнитель не сможет выполнить очередной шаг алгоритма, во время его выполнения возникнет аварийная (или исключительная) ситуация (Run Time Error), в первых русских книгах по программированию это называлось просто АВОСТ (АВарийный ОСТанов). При АВОСТе исполнитель тоже останавливает вычислительный процесс, но результата, естественно, нет. Такая ситуация иногда называется *безрезультативным остановом*.

Если подумать, то вся эта классификация выглядит достаточно сомнительно. Вот исполнитель работает над какими-то данными целый год, мы думаем, что он зациклился, а на второй год он вдруг остановится и выдаст ответ. Конечно, хорошо бы создать специальный алгоритм X, который, получая на свой вход запись любого алгоритма A и его входных данных D, говорил нам, остановится ли этот алгоритм A на этих входных данных D, или нет. Забегая немного вперёд, скажем, что, к сожалению, построить один такой универсальный алгоритм X для анализа на остановку всех алгоритмов A *невозможно*. Приходится как-то доказывать, что *данный* алгоритм A применим или не применим к *конкретным* входным данным D.

Рассмотрим теперь, а что именно можно подавать на вход алгоритма? Это, вообще говоря, всё что угодно: спицы и шерсть для вязки носков, ингредиенты для приготовления некоторого блюда, сырьё для выплавки стали и т.д. Важным является случай, когда на вход алгоритма подаётся некоторая информация.

Информация (лат. informatio – разъяснение, осведомление, изложение) относится к основополагающим сущностям нашего мира, таким, как, например, пространство, время и материя. Фактически их невозможно определить через другие, более простые сущности, потому что этих более простых сущностей нет. Обычно говориться, что информация о конкретном объекте – это всё, что уменьшает неопределённость нашего знания об этом объекте. Часто говорят, что это сведения, которые один объект содержит о другом объекте. Ну, тогда, а что такое сведения?. Обычно говолят, что это некоторые данные.

Данные определяются как информация, представленная в формализованном виде и пригодная для хранения, передачи и обработки с помощью некоторого (вычислительного) процесса. А вот алгоритмы, с помощью которых обрабатывают и получают *материальные* предметы и изделия, принято называть *технологиями*, а алгоритмы приготовления блюд и лекарств – *рецептами* 🙂.

Мы будем рассматривать только алгоритмы, которые обрабатывают данные: числа, текст, графики, изображение и т.д. В частности, на вход можно подать и запись некоторого алгоритма. Для программистов случай, когда одна программа получает на вход текст другой программы, самый обычный, так работают все компиляторы. Более «хитрый» случай, когда на вход алгоритма подаётся его собственная запись. В том случае, если алгоритм при этом остановится (всё равно, что от выдаст в качестве результата), говорят, что алгоритм самоприменим, иначе не самоприменим. Например, компилятор языка Free Pascal (или языка C) написан на самом языке Free Pascal (или языке C). Таким образом, на вход компилятора языка можно подать файл с его текстом и он «откомпилирует сам себя».

Описание (запись) алгоритма может поступать на вход исполнителя не целиком до начала работы, а по частям. Возможен случай, когда следующий шаг алгоритма становится доступным для исполнителя только после завершения выполнения текущего шага. Такие исполнители обычно называются интерактивными или интерпретаторами. Таким образом, интерпретатор в процессе выполнения алгоритма как-то взаимодействует с *поставщиком* (источником) шагов этого алгоритма. Таким поставщиком может быть человек, или же шаги выполняемого алгоритма могут быть выходными данными другого, параллельно работающего алгоритма. В частном случае выходные данные исполняемого алгоритма могут поступать на его же вход как описание новых шагов этого же алгоритма. Например, при выполнении программы на языке Lisp, выходными данными этой программы могут быть описание новых функций этого языка, и эти функции могут тут же начать выполняться 🙄.

Обычно предполагается, что при выполнении алгоритма его запись не меняется. Рассмотрим теперь некоторую модификацию схемы работы алгоритма (см. рис. 1.2). В этой схеме добавилась стрелка, ведущая от исполнителя к записи алгоритма. Это означает, что в алгоритме есть такие шаги, которые предписывают исполнителю изменить саму запись этого алгоритма, т.е. одни его шаги могут удаляться, другие добавляться, третьи изменяться и т.д. Такие алгоритмы называются самомодифи-

цирующимися. Разумеется, исполнитель должен уметь выполнять действия по такой модификации своего алгоритма.

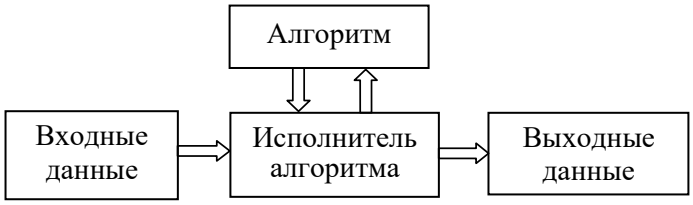


Рис. 1.2. Самомодифицирующийся алгоритм.

На алгоритмических языках, допускающих самомодификацию программ, эти программы сложно писать, отлаживать, понимать и изменять. Сейчас есть много языков, специально придуманных так, чтобы сильно затруднить, и даже сделать невозможным программирование на таких языках для человека. Обычно они называются эзотерическими языками (esoteric programming languages). В качестве примера можно привести язык Malbolge (так называется восьмой круг ада «Злые Щели» (Inferno) из книги «Божественная комедия» итальянского поэта Данте Алигьери). Этот язык имеет короткое описание (можно посмотреть в Википедии), но люди до сих пор не написали на нём ни одной программы. Как говорят, Malbolge специально написан для издевательств над программистами. На двоичной машине он работает с троичными цифрами, в языке всего 8 команд, и после выполнения каждая команда шифруется два раза. Пока получено всего несколько программ на этом языке, две из них, разумеется, выводят "`Hello, world`", вот одна из них:

```
(=<':9876Z4321UT.-Q+*)M'&%$H»!~}|Bzy?=|
{z]KwZY44Eq0/{mlk**hKs_dG5[m_BA{?-Y;;Vb'
rR5431M}/.zHGwEDCBA@98\6543W10/.R,+O<
```

Эта программа не написана человеком, а получена (выведена) другой, специально для этого написанной программой на функциональном языке Lisp, который, кстати, как упоминалось выше, тоже допускает (сильно ограниченное) написание самомодифицирующихся программ.

А вот программа, печатающая знаменитое "`Hello world!`" на языке BrainFuck, в алфавите которого всего 8 символов, одновременно являющихся *операторами* этого языка:

```
++++++++++[>+++++++>++++++++++>+++>+<
<<<-]>++.>+.+++++++..+++.>++.<<++++++
+++++++++.>.+++.------.--------.>+.>.
```

Весьма оригинальным можно считать эзотерический язык Whitespace, в его алфавите всего три символа: пробел, табуляция и перевод строки. Таким образом, любая напечатанная на этом языке программа выглядит как чистый лист бумаги.

У практически всех ЭВМ их машинные языки тоже позволяют писать самомодифицирующиеся программы.[1] Как следствие, самомодифицирующуюся программу (self-modifying code) можно написать на языке машины (или близком к машинному языке Ассемблера). Впрочем, это тема курса по архитектурам ЭВМ.

Сейчас самомодифицирующийся машинный код используется редко, в частности, в так называемых полиморфных вирусах и в некоторых механизмах защиты программ от несанкционированного использования. Кроме того, некоторые «навороченные» компиляторы используют самомодификацию машинных команд для доступа к массивам с динамически изменяющимися границами. Как сказал выдающийся программист Крис Касперски: «Окутанный мраком тайны, окруженный невообразимым количеством мифов, загадок и легенд, самомодифицирующийся код неотвратимо уходит в прошлое, медленно разлагаясь на свалке истории».

Не надо путать эзотерические языки с языками *сверхвысокого уровня*, на которых можно компактно записывать, скажем, действия над векторами и матрицами. Например, вот программа знаме-

---

[1] Такие ЭВМ должны удовлетворять двум так называемым принципам фон Неймана: принципу хранимой прогаммы и принципу неразличимости команд и данных. Подробно об этом говорится в курсе по архитектурам ЭВМ.

нитой игры «Жизнь» [1] на функциональном языке сверхвысокого уровня APL (A Programming Language):

```
life ←{
↑1 ⍵∨.∧3 4=+/,¯1 ⍬ 1∘.⊖¯1 ⍬ 1∘.⌽⊂⍵
}
```

В алфавите языка 58 специфических символов, обратите внимание, как символ нуля ⍬ отличается от символов ⊖, ∘ и ⌽. На русском языке такие символы имеют экзотические названия: «посох», «дно», «сапог», «шапка» и т.д. Впрочем, есть версии этого языка под именем J, где все эти символы набираются в виде комбинаций «обычных» символов клавиатуры). Программист, освоивший язык APL, должен свободно читать такую программу. Этот язык вообще знаменит своими «однострочными программами». Например, вот функция для нахождения наибольшего общего делителя (НОД) между минимальным и максимальным числом вектора (⌈/ – максимум, ⌊/ – минимум, ∨ – НОД):

```
GCDMaxMin ← ⌈/∨⌊/
```

Немного о функциональных языках [ii] [см. сноску в конце главы].

Разумеется, в практическом программировании такая «малопонятная» запись программ оправдана только тогда, когда это позволяет компактно записывать алгоритмы. В других случаях это приводит только к непониманию даже простых алгоритмов.

Вот мнение по поводу «читабельности» программ Майка Тейлора (Mike Taylor), который, между прочим, сам разработал эзотерический язык ETA.[2] Майк Тейлор написал: «Известны 10 преимуществ Паскаля перед Си. Я приведу только одно, но самое важное. На Си Вы можете написать:

```
for(;P("\n"),R--;P("|"))for(e=C;e--;
    P("_"+(*u++/8)%2))P("|"+(*u/4)%2);
```

На Паскале Вы НЕ МОЖЕТЕ такого написать». Любопытно, что сам Майк Тейлор профессиональный программист на языке C; видно у человека наболело…

Следующим шагом, после самомодифицирующихся алгоритмов, являются алгоритмы, модифицирующие (меняющие) свой исполнитель ⚠. Можно сказать, что такой алгоритм меняет синтаксис и семантику языка на котором он написан. Например, программа на таком гипотетическом языке могла бы добавлять в этот язык новые операторы и новые стандартные типы данных. Частично такими свойствами обладают некоторые интерпретируемые языки. С точки зрения ЭВМ-исполнителя в машине «вдруг» появляются новые команды и меняются правила выполнения старых, изменяется объём памяти, появляются новые устройства и исчезают старые и т.д. Другими словами, программа как бы меняет архитектуру ЭВМ, на которой она исполняется 🙄.

Отдалённо похожий случай иногда описывается в художественной литературе, где герои играют между собой в игру, правила которой могут меняться по ходу этой игры. Причём правила могут меняться как самими игроками, так и «сами по себе», независимо от их желания 😜.

Дальнейшее уточнение понятия алгоритма упирается в непреодолимые трудности. Действительно, а что может быть его входными данными? В принципе, что угодно. Какие действия может производить исполнитель алгоритма? Да, вообще говоря, какие угодно (если он понимает, как их выполнять).

Говорят, что роботы очень примитивные существа. Им можно приказать сделать только что-нибудь очень простое, например, «сложить два числа», «включить свет» или «собрать автомобиль».

Итак, дальнейшие уточнения определения алгоритма сделать нельзя. Строгое определение понятия алгоритма невозможно в принципе. Мы не должны этому удивляться, так как уже должны быть знакомы с такими «первичными» понятиями, которые невозможно определить через более простые, так как более простых понятий просто нет. Это, например, «время», «пространство», «информация» и т.д.

---

[1] Игра «Жизнь» (Conway's Game of Life) придумана в 1970 году математиком Джоном Конвеем (John Horton Conway, 1937-2020), к сожалению, он стал жертвой короновируса Covid-19.

[2] В алфавите этого языка всего 8 букв (остальные символы в программе игнорируются), что позволяет программе выглядеть как текст (например, стихи, написанные на естественном языке). Исполнитель ETA работает в стековом режиме и в 7-ой системе счисления 🙄.

Можно, однако, пойти «обходным» путём, сформулировав свойства, которым должен удовлетворять алгоритм. Другими словами, если нечто обладает всеми этими свойствами, то это может быть алгоритмом (а может и не быть!), а вот если не обладает хотя бы одним из этих свойств, то это точно не алгоритм 🙂. Напомним, что в математике это называется необходимыми (но не достаточными) условиями существования. Это как в известном высказывании: «Ести нечто крякает как утка, выглядит как утка и ведёт себя как утка, то, скорее всего, это и есть утка».

1.2. Необходимые свойства алгоритма

Полезно время от времени ставить знак вопроса на вещах, которые тебе давно представляются несомненными.

Бертран Рассел

Существуют следующие (необходтиые) свойства алгоритма.

- Определённость (иногда говорят понятность). Читая запись алгоритма, исполнитель должен абсолютно точно знать (понимать), как выполнять все шаги этого алгоритма. Ясно, что когда описание алгоритма сделано плохо (или вообще на другом языке), исполнитель откажется его выполнять (признает неправильным). Разумеется, и некоторые входные данные могут препятствовать правильному выполнению шага алгоритма. Например, исполнитель чётко знает, как делить друг на друга целые числа, однако делить на ноль не умеет, при этом, как говорят, возникает исключительная ситуация (или просто *исключение*). Современные языки программирования позволяют предусмотреть внутри алгоритма обработку таких исключительных ситуаций (см. главу 17).

Как ни странно, но процессоры фирмы Intel имеют режим работы, при котором они умеют делить на ноль, но только *вещественные* числа. При этом, как и полагается по здравому смыслу, при делении на ноль конечного числа, не равного нулю, выдаётся (в зависимости от знаков) ответ $\pm\infty$.

- Детерминированность. Результат работы алгоритма не зависит от месторасположения исполнителя (в Москве, на Луне, в созвездии Центавра) и времени его работы (сегодня, завтра, через 10^6 лет). Разумеется, надо соблюдать разумные ограничения, если перенести компьютер на Солнце, то он, естественно, работать не будет. Также нельзя, скажем, отодвигать запуск алгоритма за пределы времени существования Вселенной 🙂. Более строго, если алгоритм применим к конкретными входными данным, то он всегда и везде выдаст одинаковый ответ, а если не применим, то всегда и везде зациклится или сломается. Физики называют это свойство симметрией относительно пространства и времени.
- Дискретность или структурность. Каждый достаточно сложный шаг алгоритма тоже является алгоритмом и может быть разложен на более простые шаги. Это же касается и обрабатываемых алгоритмом данных (например, массив и множество состоят из элементов, файл из компонентов и т.д.). Разумеется, делить шаг алгоритма на более простые шаги без конца нельзя. Для современных цифровых ЭВМ, самый элементарный шаг алгоритма это, вероятно, переключение транзистора при смене напряжения на его затворе.

Во многих книгах по программированию (и в Интернете), к необходимым свойствам алгоритма относят также весьма удивительное свойство массовость. Оно предполагает, что для алгоритма *должно* существовать достаточно большое множество разных входных данных. Вскоре после формулировки этого свойства алгоритма, однако, авторы таких учебников гордо приводят первую программу, которая, выводит что-то вроде `"Hello, World!"`. Так как эта программа вообще ничего не вводит, то и не имеет права называться алгоритмом, что, конечно, выглядит весьма странно 🙂.

Ещё более странным выглядит встречающееся во многих книгах по информатике (и в Интернете) свойство результативность или конечность. Оно означает, что для каждого алгоритма возможно получение ответа после конечного числа шагов, а иногда даже говорится, что алгоритм должен выдать результат за конечное число шагов.

Это просто неверно, так как все программисты знают, что программа может «зациклиться». Кроме того, для одних входных данных программа может выдать ответ, а для других – зациклиться, так алгоритм она или нет? Более того, многие программы специально сделаны так, чтобы работать в бесконечном цикле. Например, так работает главная (управляющая) программа операционных систем, всё время ожидая наступление событий (нажатие клавиши, движение мышкой и т.д.) и вызывая

функции для обработки этих событий. И конечно же, существуют алгоритмы, которые не применимы ко всем своим входным данным, например:

```
read(x); repeat until false; write(x+1);
```

Вероятно, источники таких «странных» свойств алгоритма лежат в прошлом, когда в 40-х годах прошлого века только зарождалось современное понятие алгоритма. Тогда компьютеры только разрабатывались, и привычным исполнителем алгоритма был сам человек (такие люди, кстати, и назывались компьютерами 🙂). Ясно, что для людей понятие «зациклиться» было скорее не научным, а литературным («зациклиться на пристрастии к чему-либо»). И, конечно, для человека бессмысленно решать задачи, которые «ничего не вводят» или «ничего не выводят».

Подчеркнём, что мы рассматриваем свойства самого алгоритма, безотносительно к тем данным, которые поступают на его вход. В некоторых книгах пишут: «На хороших входных данных алгоритм должен остановиться и выдать результат». Тогда встаёт вопрос, какие входные данные *хорошие*, а какие нет, и чаще всего определить это нет никакой возможности, так как на *нехороших* данных алгоритм и не остановится… Замкнутый круг 🙁.

Как уже говорилось, «мощные» исполнители могут одновременно выполнять не по одному, а по несколько шагов алгоритма, предугадывая предполагаемый путь вычислительного процесса. Например, современные ЭВМ просматривают машинную программу примерно на 1000 команд вперёд и одновременно выполняют несколько десятков команд.

Любопытно, что можно написать программу, которая ничего не вводит и выводит запись своего собственного текста (само репродуцируется). Вы можете попробовать сделать это на некотором языке программирования или посмотреть, как это делается, в книге [5]. Такие программы называются куайнами (quine) по имени философа, логика и математика Уилларда Ван Орман Куайна (Willard Van Orman Quine). Первая такая программа была написана в 1960 году Хэмишем Дюаром (Hamish Dewar) на языке Atlas Autocode и имела длину в 26 строк. А вот одна из самых коротких таких программ на языке C, написанная В. Таировым и Р. Фахреевым:

```
main(a){
  printf(a,34,a="main(a){printf(a,34,a=%c%s%c,34);}",34);
}
```

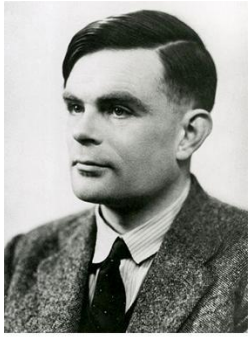
Возможны и более сложные варианты таких программ (куайн n-го порядка, цепной куайн и т.д.).

А вот дальше развивать и изучать понятие алгоритма «в общем виде» уже не представляется возможным, слишком много неопределённостей. В сороковых годах прошлого века, однако, появились первые вычислительные устройства, и возникла настоятельная необходимость сделать определение алгоритма более строгим. Как говорят, надо было **формализовать** понятие алгоритма, приблизив его строгость к определениям и утверждениям математики. Конечно, это можно сделать не «в общем виде», а только *ограничив* возможности алгоритма.

1.3. Формализация понятия алгоритма. Машина Тьюринга

Отыщи всему начало, и ты многое поймёшь.

Козьма Прутков



А.Тьюринг (1912-1954)

Алан Матисон Тьюринг (Alan Mathison Turing) был выдающимся математиком, логиком и криптографом. Он одним из первых в 1936 году формализовал понятие алгоритма, описав свою знаменитую машину Тьюринга (Turing machine).[1] Сначала он ограничил входные и выходные данные алгоритма только словами в некотором алфавите. Здесь два новых понятия – алфавит и слово в этом алфавите. Алфавитом назовём конечный набор различимых друг от друга (для исполнителя алгоритма) значков-символов:

$$A=\{a_1,a_2,\ldots,a_n\}$$

Разумеется, существуют и потенциально *бесконечные* алфавиты (скажем, китайский, где можно придумывать всё новые и новые иероглифы-символы), но

[1] Эту машину назвал так не сам Тьюринг, а его учитель, математик и логик Алонзо Чёрч (Alonzo Church).

мы их рассматривать не будем. Затем определим *слово* в этом алфавите как конечную (возможно и пустую) цепочку символов алфавита, которую будем считать упорядоченной и читать слева направо:

$w=\alpha_1\alpha_2\ldots\alpha_k$

Опять таки, на Земле существуют языки, в которых слова читаются справа налево, или же сверху вниз, но это не наш случай. Итак, теперь алгоритма может вводить не всё, что угодно, а только слова в заданном алфавите. Таким образом мы отбросили все «физические» входные данные (спицы и шерсть для вязания, сахар и фрукты для варки варенья и т.д.). Машина Тьюринга может обрабатывать только данные (что это такое мы говорили выше). Более того, из данных мы оставили только *описания* физических объектов словами в некотором алфавите, отбросив, например, звуковые и графические данные.

Первый вопрос заключается в том, не потеряли ли мы что-нибудь важное при таком ограничении входных данных? Всю ли интересующую нас информацию о физическом мире можно записать в виде слов? Ясно, что, если пробел и знаки препинания считать буквами, то привычные нам предложения в книге становятся словами. Если переход на следующую строку текста задать в виде буквы, то вся страница книги будет словом, а если и переход на следующую страницу сделать буквой, то вся книга станет словом. Далее, словом можно сделать всю библиотеку и т.д. Разумеется, если конец книги не буква, то разные книги будут разными словами.

Мы знаем, что символами можно описать шахматную партию, химическую формулу, записанное нотами музыкальное произведение. Для представления любой аудио информации можно каждую секунду звучания разбить, скажем, на 22000 (а лучше 44000) частей, и в каждой такой части указать силу звука в виде числа. Аналогично, и изображение можно разбить на матрицу точек и для каждой такой точки задать номер (и интенсивность) цвета.

Остались ли какие-либо данные из реального мира, которые мы бы не смогли записать в виде входного слова для машины Тьюринга? Здесь можно вспомнить такой курьёзный случай из истории информатики. В 70-х годах прошлого века стали активно внедрять так называемые АСУП (Автоматизированные Системы Управления Предприятием). Теперь вместо людей компьютер должен был включать и выключать электродвигатели, открывать и закрывать краны в химическом реакторе, охлаждать и нагревать нужные вещества и т.д. И вот, подходят программисты на одном химическом заводе к заслуженному технологу дяде Васе и спрашивают: «Ты, дядя Вася, когда закрываешь вот этот кран на химическом реакторе, чтобы получить хороший продукт? Рассказывай, мы сейчас это запрограммируем». И дядя Вася отвечает: «Здесь всё просто. Окуну палец в продукт и лизну. Если ядрёно, то кран закрываю».

Ясно, что здесь в виде слова надо формально описать вкусовые ощущения. Аналогичные трудности возникают и для запахов. Мы, однако, не теряем надежды, так как знаем, что и вкус и запах определяются химическими молекулами, которые мы вполне может описать символами. Сейчас ситуация исправилась, например, существуют «электронные собаки», программы которых достаточно уверенно распознают наличие в воздухе молекул наркотиков и взрывчатки. Иногда можно увидеть такие приборы, напоминающие маленькие пылесосы, например, у дежурных в метро. Раньше они работали хуже, чем обычные собаки, но с использованием нейронных сетей ситуация сильно улучшилась 🙂.

Входное слово машины Тьюринга располагается на специальной ленте, разбитой на клетки, в каждой из которых записан ровно один символ алфавита. Например, вот в алфавите `A={a,b}` на ленте слово `abbab` длиной пять букв:

Λ	Λ	Λ	a	b	b	a	b	Λ	Λ	Λ	Λ
			↑								

Обратите внимание, что слева и справа от входного слова все клетки ленты заполнены особой пустой буквой `Λ`. Эта буква по определению входит в любой алфавит, но сама *внутрь входного* слова входить не может (это ограничитель слова, как, например, пробел в обычных языках).

Когда на входной ленте находятся *несколько* входных слов, разделённых пустыми символами, то считается не выполненным так называемое предусловие алгоритма, о чём мы будем подробно говорить при описании языка Паскаль. Сейчас лишь скажем, что в этом случае действие алгоритма не определено. Отметим, что аналогичная ситуация наблюдается и во всех языках программирования высокого уровня (HLL – High Level Language). Например, в языке C++ описано около 200 случаев, когда у программы будет так называемое неопределённое поведение (undefined behavior). Это означает, что «как это будет работать толком неизвестно, но ничего хорошего не будет» 🐷.

В каждый момент времени на одну из клеток установлена читающая и пишущая головка, обозначенная у нас знаком ↑. По командам исполнителя алгоритма эта головка может двигаться по клеткам ленты налево и направо, каждый раз сдвигаясь на одну клетку, или оставаться не месте. Лента считается потенциально бесконечной, то есть головку можно двигать налево и направо сколько угодно раз. Таким образом, записанное на ленте слово имеет конечную, но как угодно большу́ю длину.

Далее, надо строго описать исполнитель машины Тьюринга, сказать, какие действия он может выполнять. Основная идея здесь была в следующем: исполнитель должен быть очень простым, оставаясь при этом *универсальным* (что это такое, мы вскоре расскажем). Простота исполнителя позволяет всем людям понимать его действия однозначно, без всяких двусмысленностей.

В каждый момент времени машина Тьюринга находится в одном из конечного множества своих состояний, которые будем обозначать значками q_1, q_2,…,q_n. Выполняя шаги алгоритма, машина Тьюринга может *переключаться* из одного состояния в другое. Проще всего представлять себе это как переключение скорости в автомобиле, смена режимов работы в стиральной машине и т.д.

Итак, исполнитель машины Тьюринга может делать только следующее:

1. Читать символ из клетки, на которую сейчас установлена головка.
2. Писать символ в клетку ленты, на которую сейчас установлена головка, старый символ стирается и на его место пишется новый символ. Впрочем, старый и новый символы могут и совпадать.
3. Двигать головку налево или направо на одну клетку, в частном случае головка никуда не двигается (остаётся на месте). Будем обозначать эти команды исполнителя русскими буквами Л (наЛево), П (наПраво) и Н (Никуда не двигаться).
4. Переключаться из одного состояния в другое (в частном случае в то же самое) состояние.

Ничего другого исполнитель делать **не умеет**! Как видно, за один шаг работы в слове на ленте может измениться только одна буква.

Существуют различные формы представления алгоритма для машины Тьюринга, самая наглядная из них – записать алгоритм в виде прямоугольной таблицы (далее будет рассказано и о других способах записи алгоритма для машины Тьюринга). Строки этой таблицы «пронумерованы» состояниями q_i, в которых может находится машина Тьюринга, а столбцы «озаглавлены» символами алфавита (включая пустой символ).[1] В каждой клетке на пересечении строки и столбца записаны ровно три операции для исполнителя машины Тьюринга

<символ><движение><переключение>

Первая операция задаётся символом, который надо записать в текущую позицию ленты, вторая определяет движение головки {Л,П,Н}, а третья – состояние q_i, в которое надо переключиться. Например, $bПq_2$, т.е. записать в клетку ленты символ b, сдвинуть головку направо и переключиться во второе состояние. Обратите внимание, что внешний вид команд выбран так, чтобы между ними можно было не ставить никаких разделителей, скажем, пробелов или запятых.[2]

Поговорим теперь об алфавите машины Тьюринга. В общем случае, результат решения задачи (выходное слово) может записываться не в том же алфавите, в котором записано входное слово. Будем называть $A_{вх}$ (входной) и $A_{вых}$ (выходной) алфавиты машины Тьюринга. Кроме того, в процессе решения задачи на ленту могут записываться и дополнительные (служебные) символы, которых нет ни во входном, ни в выходном алфавите. Таким образом, появляется внутренний алфавит $A_{вн}$, в который уже входят все символы, с которыми работает данная машина Тьюринга. Символами именно этого внутреннего алфавита и «озаглавлены» все столбцы таблицы, в виде которой записан алгоритм для машины Тьюринга. Отметим, однако, что во внутренний алфавит могут не входить те символы выходного алфавита, которые машина записывает на ленту, но никогда не читает оттуда, следовательно, эти символы и не будут искаться над столбцами таблицы.

Далее опишем один шаг работы машины Тьюринга. Сначала читается символ из клетки, на которой установлена головка. Затем ищется столбец таблицы, озаглавленный этим символом. Если такого

---

[1] В некоторых книгах, наоборот, строки озаглавлены символами, а столбцы – состояниями. Это неудобно, так как программист привык писать новые команды в программу сверху вниз, а не слева направо.

[2] Когда в алфавит необходимо включить сами буквы Л, П или Н, то можно, например, перейти на английские названия этих команд L (Left), R (Right) и N (Not move) или придумать что-нибудь ещё.

столбца нет, то это «чужой» символ, который не входит во внутренний алфавит машины Тьюринга. В этом случае производится аварийный (безрезультативный) останов алгоритма.

Когда символ найден, то берётся клетка таблицы, расположенная на пересечении этого столбца и строки, «пронумерованной» *текущим* состоянием q_i машины Тьюринга. Затем последовательно выполнятся все три команды, которые расположены в этой клетке. Далее по точно таким же правилам выполняется следующий шаг работы и т.д. Обратите внимание, что после выполнения очередного шага алгоритма, исполнитель начисто забывает, что он только что сделал, и каждый следующий шаг начинает с «чистого листа».

Определим теперь, как наша алгоритмическая система начинает свою работу. Договоримся, что в начале работы головка стоит на первом (слева) символе входного слова и исполнитель находится в первом состоянии q_1. В качестве примера возьмём простой алфавит из двух букв `A={a,b}` (так как пустой символ Λ входит в любой алфавит, то не будем указывать его явно). На рис. 1.3. слева показана таблица машины Тьюринга, а справа – последовательные (на каждом шаге работы) состояния слова и головки на ленте, внизу у стрелки указано текущее состояние).

	a	b	Λ
q_1	Λ П q_2	Λ П q_2	Λ Н q_1
q_2	Λ П q_3	Λ П q_3	Λ Н q_2
q_3	a Н q_3	b Н q_3	Λ Н q_3

Λ	a	b	b	a	b	Λ
	↑q_1					
Λ	Λ	b	b	a	b	Λ
		↑q_2				
Λ	Λ	Λ	b	a	b	Λ
			↑q_3			

Рис. 1.3. Таблица машины Тьюринга и трассировка её работы.

На первом шаге головка читает с ленты букву a и, находясь в состоянии q_1, выбирает для выполнения клетку таблицы `ΛΠq2`, которая находится на пересечении первой строки и первого столбца. Выполнив команды из этой клетки, исполнитель пишет на место буквы a пустой символ Λ, сдвигает головку направо и переключается в состояние q_2. На следующем шаге работы с ленты читается символ b и для исполнения выбирается клетка `ΛΠq3` на пересечении второй строки и второго столбца нашей таблицы. Выполнив команды из этой клетки, машина стирает букву b, двигается вправо и переключается в состояние q_3. А вот на следующем шаге работы на выполнение выбирается клетка `bHq3`, расположенная на пересечении столбца, озаглавленного буквой b и третьей строки таблицы, соответствующей состоянию q_3.

Мы можем заметить, что это очень странная клетка, попав в которую машина Тьюринга будет далее выполнять её бесконечно: символы на ленте не меняются, головка не двигается и в другие состояния машина не переключается. Такие клетки называются клетками останова, попав в которые, машина Тьюринга считает, что алгоритм завершён, и останавливается.[1] Внимательно посмотрите нашу таблицу, и найдете в ней пять клеток останова. Ясно, что, если в таблице нет ни одной клетки останова, то машина никогда не остановится, и, следовательно, алгоритм не применим ни к одному входному слову (вот и первая теорема 🙂). Что же делает этот алгоритм? Как можно понять, если во входном слове более двух символов, то наш алгоритм стирает ровно две первые буквы, иначе стирает все буквы, выдавая в качестве ответа пустое слово.

По определению, машина Тьюринга в качестве ответа всегда выдаёт только одно (возможно и пустое) слово. Будем считать, что, если головка после останова алгоритма стоит не на пустой клетке (т.е. в пределах некоторого слова), то это слово и есть ответ, когда головка стоит на пустой клетке, то ответом считается пустое слово.[2]

Возможны и другие правила определения начала и конца работы машины Тьюринга. Например, можно договориться, что как в начале, так и в конце работы головка стоит на первой пустой клетке за входным и, соответственно, выходным словом. Ясно, что если теперь перед головкой тоже пустая

---

[1] Первоначально в машине Тьюринга предполагалость целое состояние останова q_s, попав в которое исполнитель останавливается. Для программиста, однако, более экономно сделать именно клетку останова.

[2] Для случая когда машина Тьюринга после останова оставляет на ленте не одно, а, скажем, два слова, будем предполагать, что всегда можно изменить алгоритм работы этой машины так, чтобы перед остановом она стирала это «лишнее» слово, оставляя только один ответ. Вообще то это теорема, которую надо доказать ⚠.

клетка, но входное и, соответственно выходное, слово пустое. Иногда такую машину Тьюринга называют *нормальной* (интересно, почему?).

Для более компактной записи таблицы для машины Тьюринга можно использовать некоторые условные обозначения. Во-первых, можно не писать команду отсутствия движения [H]. Во-вторых, можно не указывать записываемый в клетку ленты символ, который там уже находится тогда, например, вместо [bHq3] можно писать просто [q3]. В-третьих, можно опустить команду переключения, если машина остаётся в прежнем состоянии, например, вместо [bПq3] писать просто [П].

	a	b	Λ
q1	ΛПq2	ΛПq2	!
q2	ΛП!	ΛП!	!

Заметим, что когда опущены все три команды, то это клетка останова, чтобы такие клетки «бросались в глаза», будем вместо неё писать [!]. Таким образом, знак [!] будет новой командой для исполнителя «перейти в какую-нибудь (любую) клетку останова». С этими условными обозначениями наша программа запишется более компактно и наглядно, как показано слева, при этом третье состояние уже не нужно.

Таким образом, мы полностью определили алгоритмическую систему машины Тьюринга. В начале работы на ленте ровно одно (возможно пустое) входное слово, слева и справа от которого потенциально бесконечное число пустых клеток, головка стоит на первом слева символе слова и машина находится в первом состоянии [q1]. Строго определён шаг работы исполнителя машины Тьюринга, все шаги выполняются одинаково, вычислительный процесс заканчивается, когда машина попадает в клетку останова. Определены два аварийных (без результативных) останова:

1. С ленты считан «чужой» символ, которого нет в заголовке ни одной колонки таблицы. Обратите внимание, что на ленту можно писать «чужие» символы, но их нельзя читать.
2. В выполняемой клетке стоит команда [qi] переключения в несуществующее состояние.

Итак, машина Тьюринга устроена так просто и выполняемые ею действия (шаги алгоритма) настолько элементарны, что не должны вызывать у людей неоднозначного толкования. В то же время этот исполнитель в определённом смысле универсален, так как для него был сформулирован так называемый тезис Тьюринга.

> Тезис Тьюринга Для любого алгоритма обработки слов в некотором алфавите можно построить машину Тьюринга, которая делает то же самое. Более точно, она имеет ту же область применимости, и для любого слова из этой области даёт тот же результат. Другими словами, машина Тьюринга может решать все задачи, которые сможет решить любая другая алгоритмическая система обработки слов.

Ясно, что строго доказать этот тезис невозможно, так как в нём присутствует неформализованное понятие «любая система обработки слов». Единственное, что остаётся, это опровергнуть тезис Тьюринга. Для этого надо придумать формальный исполнитель алгоритмов обработки слов, который может решать задачи, недоступные машине Тьюринга.

Каково максимальное количество шагов, которое может выполнить машина Тьюринга, перед тем, как остановиться? Эта задача известна в информатике под названием «задача усердного бобра» (busy beaver). Это частный случай задачи поиска алгоритмов, которые работают максимально долго, впервые она была сформулирована венгерским математиком Тибором Радо в 1962 году. Ясно, что для машины Тьюринга ответ зависит от числа состояний в таблице. Формально задача ставится так: в алфавите из двух символов [{1,Λ}], первоначально пустой ленте и таблице из N состояний (и, возможно, с дополнительным состоянием останова) написать программу, которая остановится, выполнив максимальное число шагов, обозначим эту функцию как BB(N).[1]

	1	Λ
q1		1Пq2
q2	1П!	1Лq1

Легко доказать, что BB(2)=6, слева показана эта машина Тьюринга. В 1965 году Радо Шэнь Линь показал, что BB(3)=14 и BB(4)=107. В 1965 году показано, что BB(5)>=47176870. На сегодняшний день для BB(6) найдена оценка 7.4 10^{36534}, но не доказано, что оно максимальное. Для BB(7) пока есть только оценка BB(7)>=10^10^10^10^18705352 и мнение, что эта задача решена быть не может 🙁. [iii] [см. сноску в конце главы]

---

[1] Есть модификация этой задачи с максимальное число "1", которые остаются на ленте после останова.

1.4. Решение задач в машине Тьюринга

Программирование начиналось как искусство; даже сейчас большинство учится ему, наблюдая, как работают другие (например, преподаватель или более опытный коллега), постигая приёмы и мало задумываясь над принципами, которые лежат в их основе. Однако в результате научных исследований последнего десятилетия найдены некоторые полезные теоретические положения и общие принципы, так что наступает время, когда можно начинать учить принципам и их осознанному применению.

Дэвид Грис
«Наука программирования», 1984 г.

Когда надо составить алгоритм решения конкретной задачи для машины Тьюринга, то говорят не «написать (составить) программу для машины Тьюринга», а «построить машину Тьюринга для решения задачи». Действительно, при решении каждой задачи мы строим *новую* машину Тьюринга, у которой свой алфавит и своё число состояний (определяемые условием задачи).[iv] [см. сноску в конце главы]

Итак, давайте приступим к разработке алгоритмов для машины Тьюринга. Сначала, однако, рассмотрим вопрос, а как вообще строить алгоритм для решения задачи? Обычно такое умение человека называется алгоритмическим мышлением. Умения человека, как правило, имеют как формальную (действовать по определённым правилам), так и творческую составляющую (действовать по вдохновению 🙂). Так, многим знаменитым скульпторам приписывается такое высказывание. В ответ на вопрос: «Как Вам удаётся создавать такие прекрасные скульптуры?» они отвечали: «Просто я беру камень и отсекаю от него всё лишнее».

Конечно, творческая компонента есть и в разработке алгоритма (написании программ), этому можно научиться только разрабатывая свои собственные алгоритмы и анализируя чужие (хорошие). В то же время и формальная составляющая очень важна. Как бы не был талантлив скульптор, но для каждой твёрдости камня он должен брать особое зубило, приставлять его к камню под строго определённым углом и бить молотком с нужной силой. Иначе камень будет просто как-то откалываться, и никакой скульптуры не получится.

Так и в программировании необходимо придерживаться некоторых формальных правил, иначе хорошей программы не получится. Вот с этими правилами мы сейчас и познакомимся, обычно они формулируются в виде шагов (этапов), которые необходимо выполнить при написании программы. Часто это называется жизненным циклом программы (Software Development Life Cycle). Перечислим эти этапы в порядке их выполнения.

1. Осознание (понимание) задачи. Странно, но часто программисты плохо понимают, какую задачу им надо решить, что дано на входе, и каким должен быть результат. Преподавателям многократно приходилось сталкиваться с фактом, что учащийся решил не ту или не совсем ту задачу, которая перед ним ставилась. Например, рассмотрим такую простую задачу: «Ввести одно целое число и вывести сумму всех значащих десятичных цифр в этом числе». Здесь можно не осознавать, что число может быть отрицательным, а цифры всегда неотрицательные. [1] Так что не надо спешить и пропускать этот этап.
2. Спецификация задачи (Requirements Specifications) На этом этапе надо внимательно проанализировать все особые случаи, возможные во входных данных задачи. Для каждого такого случая надо чётко определить, как при этом должен себя вести алгоритм, и какой результат он выдаст. Заметим, что обычно здесь различают заказчика (он формулирует задачу), и разработчика (он реализует алгоритм решения этой задачи). Ясно, что принятая спецификация должна быть одобрена заказчиком. К сожалению, заказчик часто и сам не знает, чего он точно хочет, и задача программиста – помочь ему осознать требование к программному продукту. Итак, для всех входных данных (даже на первый взгляд невозможных) программа должна да-

[1] Современные нейронные сети (большие языковые модели), если попросить их написать такую программу, тоже делают аналогичную ошибку, так как большинство примеров решения этой задачи в Интернете именно такие.

вать предусмотренный Вами результат. В некоторых книгах об этом говорят как о защитном (или безопасном) программировании (defensive programming), а иногда и просто как о защите от дурака 🙂).

Допущенные на этом этапе ошибки обычно потом очень тяжело исправить. В качестве примеров расскажем о двух таких ошибках. Старшее поколение помнит, что сначала плата в метро составляла одну пятикопеечную монету, затем появились пластиковые жетоны, и, наконец, карты с магнитной полосой. Программа турникета для магнитных карт имела такую спецификацию.

Для повышение надёжности, магнитная полоса делилась на две равные части с идентичными данными. Магнитная полоса считывалась турникетом, и сначала программа анализировала первую часть полосы. Если она считалась правильно (контрольные суммы совпали), то из оставшегося числа поездок вычиталась единица, и результат записывался и на первую, и на вторую часть полосы. Если же правильно считать первую половину не удалось (например, она поцарапана), то анализировалась вторая половина магнитной полосы. Если она правильная, то из её оставшегося числа поездок вычиталась единица, и результат записывался и на первую, и на вторую часть полосы (была вероятность, что после записи на поврежденную часть полосы данные на ней восстановятся и будут потом правильно читаться).

Как Вы догадываетесь, спецификация содержала грубую ошибку, которой тут же воспользовались несознательные пассажиры. У новой карты, рассчитанной на `N` поездок, заклеивалась скотчем первая половина магнитной полосы, таким образом она закрывалась на чтение и запись, т.е. работала только вторая часть полосы. После исчерпания числа поездок во второй части, на один следующий раз снимался скотч с первой половины, а потом она снова заклеивалась на `N−1` поездку. Легко подсчитать, что общее число разрешённых поездок будет не `N`, а уже `N*(N−1)/2`.

Отметим, что схожая задача стоит при защите данных от повреждений (царапин) на поверхности лазерных дисков (ранее они широко использовались). Там задача сохранности данных решается так. Биты каждого байта данных записываются на дорожку такого диска не подряд, а с шагом в два-три сантиметра. Кроме того, на диск добавляются избыточные байты. В этом случае царапина портит но одному биту в каждом байте, и особые средства коррекции (так называемые коды Рида-Соломона), позволяют восстановить испорченные биты. Вероятно, похожую спецификация надо было сделать и для магнитных карт метро.

Второй пример связан с разработкой программы для бортового компьютера тяжёлого американского истребителя. В 80-х годах прошлого века эти компьютеры стали такими мощными, что могли в автоматическом режиме выводить самолёт на цель, атаковать, возвращаться на свой аэродром и даже садиться без помощи пилота (конечно, только на специальные аэродромы). Испытания этой системы успешно подходили к концу, и вот последний полёт. Самолет взлетел из Флориды и взял курс на юг. Сначала всё шло хорошо, но вдруг в центр управления пришёл сигнал бедствия: «Самолет перевернулся вверх шасси и не слушается управления!». Конечно, тут же приказ: «Перейти на ручное управление, срочно возвращаться на базу!».

Как потом выяснилось, была составлена следующая спецификация на хранения координат самолёта: долгота от 0 до 360 градусов, широта от +90 градусов (северный полюс) до –90 градусов (южный полюс). Третьей координатой была высота над уровнем мирового океана. При пересечении экватора одна из координат сменила знак, и программа решила, что самолёт летит вверх шасси.

Ошибка была исправлена, самолёт стал выпускаться серийно, и часть таких самолётов купила Саудовская Аравия. И вот, лётчики отрабатывают пилотирование на предельно низких высотах (бреющем полёте), и летят над Мёртвым морем. А, надо сказать, что уровень Мёртвого моря много ниже уровня мирового океана (а уровень Каспийского моря вообще на 29 метров ниже)… Судя по недавним крушениям самолётов фирмы Боинг, дела со спецификацией в этой фирме, вероятно, до сих пор не в порядке. ˅ [см. сноску в конце главы]

3. Разбиение на подзадачи. Этот этап имеет много и других названий: «Программирование сверху вниз» (top-down programming), «Пошаговая детализация» (stepwise refinement), «Модульный анализ» и другие. Суть здесь проста: когда задача слишком сложна, чтобы сразу записать её решение на нужном языке программирования, то она разбивается на более мелкие подзадачи. Как можно догадаться, здесь мы пользуемся свойством структурности алгоритма. Далее, если какая-нибудь подзадача опять оказывается слишком сложной, мы тоже разбиваем её на подзадачи, и т.д.

В языках программирования для реализации самых крупных подзадач используются так называемые модули, а для более мелких – процедуры и функции. Главная трудность здесь в том, что выходные данные одной подзадачи чаще всего являются входными данными для другой. В этом случает говорят, что подзадачи образуют *суперпозицию*. Ошибки в согласовании форматов выходных и входных данных очень трудны для обнаружения и исправления.

В качестве шутки иногда говорят о нашем отечественном способе разработки программ «Программирование снизу вверх наискосок»: «Многие программисты утверждают, что прежде чем начинать писать программу, необходимо время на обдумывание алгоритма, а некоторые даже призывают вникнуть в суть задачи, которую предстоит решать. Категорически не следует интересоваться постановкой задачи до момента получения готовой программы».

4. Кодирование (coding). На этом этапе для всех полученных подзадач разрабатываются алгоритмы, которые записываются на языке программирования. Возможна запись подзадач как на одном, так и на нескольких (чаще всего на двух) языках программирования, такие системы программирования называются *многоязыковыми*. Это самый понятный этап решения задачи, многие начинающие программисты считают его главным, но мы скоро поймём, что это не так.[vi] [см. сноску в конце главы]

На этапе записи алгоритма на языке программирования возможно проникновение в программный текст различных *семантических* ошибок. Это чаще происходит для языков программирования, которые не обладают хорошим (надежным) синтаксисом. Пожалуй, самым известным примером такой ошибки была запись символа точки вместо запятой в программе на Фортране. Эта программа управляла 27 августа 1962 года ракетой-носителем Атлас для запуска на Венеру космического корабля Маринер-2. Вместо заголовка оператора цикла `DO 50 I = 12,525` после замены запятой на точку и учитывая, что в Фортране пробелы внутри имён допускаются и игнорируются, получился оператор присваивания для неописанной вещественной переменной `DO50I=12.525` [17]. В результате через 5 минут после старта ракету пришлось взорвать. Ущерб составил 18,5 миллионов долларов. Видно, что язык Фортран игнорирует пробелы в записи имён и чисел, позволяет не описывать переменные, а также допускает использование служебных слов не по их прямому назначению. Всё это свидетельствует о ненадёжности его синтаксиса, поэтому в последних версиях языка Фортран такое, как горорят *неявное* (implicit) *объявление* типа программист может отключить директивой `IMPLICIT NONE`.

5. Отладка программы (debuging). Необходимо подготовить достаточное число наборов входных данных с заранее известными результатами работы программы. Далее все такие наборы подаются на вход программы и проверяется, что ответы совпадают с ожидаемыми. Этот этап по праву считается одним из самых трудных. Как с юмором писал известный Брайан Керниган, автор языка C: «Отладка программы вдвое сложнее, чем её написание. Так что если вы пишете программы настолько умно, насколько можете, то вы по определению недостаточно сообразительны, чтобы их отлаживать».

   Вместе с отладкой часто производится тестирование (testing) написанной программы. Разница между отладкой и тестированием заключается в том, что отладку производит тот, кто кодировал программу, а тестирование – «посторонний» программист. У них разные цели. Автор программы пытается убедить (прежде всего самого себя), что в его программе нет ошибок, а тот, кто тестирует программу, наоборот, пытается найти в ней как можно больше ошибок. К сожалению, учащиеся при отладке очень быстро убеждают себя, что в их программе ошибок нет, а преподаватель тестирует программу и быстро находит в ней ошибку (и, как правило, не одну 🙂).

Любопытно, что существует хорошая методика написания программ под названием *разработка через тестирование* (TDD – Test-Driven Development), её создал Кент Бек (Kent Beck), сотрудник кампании Facebook. По этой методике (принятой во многих серьёзных программистских фирмах) для поставленной задачи программист сначала разрабатывает полный набор тестов, а лишь потом пишет программу, которая должна показать свою правильность на этом наборе. Аналогично, когда в программу надо добавить новые возможности, сначала пишутся тесты, покрывающие эти возможности, а лишь затем модифицируется сама программа. Эта методика считает, что главным назначением тестов является вовсе не выявление ошибок в программе, а более глубокое осмысление человеком поставленной задачи в процессе написания тестов 🙄. Вот цитата из книги Мартина Роберта «Идеальный программист»: «Я не знаю ни одной методологии, которая бы сравнилась по эффективности с TDD».

Часто даже большие программистские фирмы не имеют достаточно средств и сотрудников, чтобы качественно протестировать новый программный продукт.[1] Тогда они выпускают «бета версию» своей системы, отдают (часто даже бесплатно) её достаточно большой группе пользователей и просят их сообщать о всех замеченных ошибках. И только после некоторого времени такого «общественного» тестирования фирма выпускает «итоговую версию (релиз)» своей программы.

И, наконец, существует ещё так называемая верификация (verification) программного продукта. При верификации проверяются заявленные авторами характеристики программ. Например, если было заявлено, что программа может за один запуск обрабатывать до пяти входных файлов по 4 Гб каждый, а проверка показала, что на пятом файле программа «виснет», то верификация не пройдена. Особенно тщательно проходят верификацию популярные компиляторы, проверяется, что они правильно обрабатывают все тонкости входного языка программирования. Когда верификация сопровождается повторной тщательной отладкой программного продукта, то иногда говорят о валидации (validation) программы.

6. Оптимизация. Этот этап проводится для уже работающих программ, когда они не удовлетворяют авторов и/или пользователей по быстродействию или потребляемым ресурсам, в основном размеру требуемой памяти (оперативной или внешней).

Вот что говорит о способностях, которыми должен обладать программист, Чарльз Уэзерелл в книге «Этюды для программистов» [5]:

«Перечислим те способности, которые жизненно необходимы всякому программисту (и очеркисту тоже).

1. Способность читать и понимать описание поставленной задачи, улавливать пожелания того, кто её ставит (что не всегда легко, так как и задачи, и те, кто их ставит, часто отличаются именно неуловимостью).
2. Способность чётко видеть действительные трудности и отбрасывать всё, не относящееся к делу.
3. Способность выявлять все случаи, где можно применить теорию, самостоятельно решиться на её применение или обратиться за советом к специалисту.
4. Способность разбить задачу на ряд обозримых независимых частей и понять взаимосвязи этих частей.
5. Способность оценивать эффективность предлагаемых решений с точки зрения затрат на программирование, машинных ресурсов и удовлетворения потребностей пользователя и находить приемлемый компромисс между этими видами эффективности.
6. Способность объединять множество частных решений воедино, получая при этом чёткое и изящное решение всей задачи.
7. Способность выражать решения на простом и понятном языке. Естественный это язык или искусственный – роли не играет, важно лишь, чтобы правильность решения была ясна и людям, и машине.
8. И, наконец, способность при неудаче подавить самолюбие и поискать другой подход (или даже другую задачу)».

После всех этих этапов программа начинает эксплуатироваться пользователями. Разумеется, и после этого в программе будут находиться ошибки, которые необходимо исправлять. Обычно для найденной ошибки сначала проверяется, не возникла ли она на этапе кодирования. Например, где-нибудь набили знак "–" вместо знака "+", использовали не ту константу и т.д. Если место ошибки найдено и она исправлена, то придётся снова отлаживать программу.

Когда таким образом найти ошибку не удаётся, то проверяется этап разбиения на подзадачи. Например, все ли функции получают свои параметры в правильном формате, хорошо ли построена цепочка вызываемых процедур и функций и т.д. Если найдена именно такая ошибка, то приходится повторять этап разбиения на подзадачи, а следом и этап отладки.

Бывает, однако, что и здесь найти ошибку не удаётся, тогда приходится возвращаться на этап спецификации. Например, вдруг не учтён какой-нибудь «хитрый» случай во входных данным. Но здесь, как сказал Алан Перлис (первый лауреат премии Тьюринга) часто «Легче изменить специфи-

[1] Обычно программным продуктом называют такие программы, которые могут эффективно использоваться без участия авторов этих программ.

кацию, чтобы она соответствовала программе, но не наоборот». После нахождения такой ошибки приходится повторять все следующие этапы (разбиение на подзадачи, кодирование и отладку).

Самый тяжёлый случай, когда ошибка возникла на этапе осознания алгоритма, когда программист не совсем правильно понял, что его программа должна делать. После исправления такой ошибки приходится заново проводить все последующие этапы. Итак, чем на более раннем этапе возникла ошибка, тем тяжелее её исправить, именно поэтому первым этапам надо уделять повышенное внимание.

Принято считать, что этап кодирования, который многие программисты отождествляют с созданием программы, занимает 10-15% всего времени разработки программы. Большие программы создаются коллективом разработчиков, которые специализируются на разных этапах работы, причём «кодировщики» или «кодеры» относятся к самой низко квалифицированной части команды разработчиков. Часто поступившие на факультет студенты начинают хвастаться, как много языков программирования они знают: «Я могу программировать и на языке C, и на Python, и на Java…». Это напоминает пришедшего на завод специалиста, который хвастается, что умеет работать и на сверлильном станке, и на фрезерном, и на шлифовальном… Тогда его спрашивают: «А ты расчёт и чертёж детали сделать сможешь?» Да, в университетах готовят не программистов, а разработчиков программного обеспечения, а это, как говорится «две большие разницы». Соответственно, есть и две специальности: Computer Engineering (для программистов) и Computer Science (для разработчиков).

Итак, рассмотрим первую задачу. Построить машину Тьюринга, которая для каждого слова в алфавите `{a,b}` удаляет из этого слова последний символ. Сначала поймём, что на ленте может быть слово любой длины, и для непустого слова его длина уменьшится ровно на один символ. Далее выполняем спецификацию задачи и отмечаем особый случай, когда на ленте пустое слово. Что в этом случае делать? В этом случае можно предложить два типичных способа поведения алгоритма.

Во-первых, можно считать, что в пустом слове нет последнего символа, ничего делать не надо и нужно просто остановиться. Это «математический» подход, ведь для математика пустое множество ничем не хуже всех остальных. При «программистском» подходе можно считать это ошибкой во входных данных и остановиться с выдачей диагностики (например, оставить на ленте символ `'*'`). Необходимо понять, что это два разных алгоритма решения одной задачи, для выбора нужного можно проконсультироваться у того, кто поставил задачу. К сожалению, часто в задачниках такой спецификации не делается, т.е. задача не доопределена, и программисту приходится это делать самому. Например, мы примем решение для пустого слова ничего не делать и сразу остановиться.

Далее, в нашей задаче просматриваются две подзадачи.

1. Поставить головку на последний символ слова (для пустого слова сразу перейти ко второй подзадаче).
2. Стереть символ, на который указывает головка, сдвинуться на предыдущий символ и остановиться.

	a	b	Λ
q_1	П	П	Лq_2
q_2	ΛЛ!	ΛЛ!	!

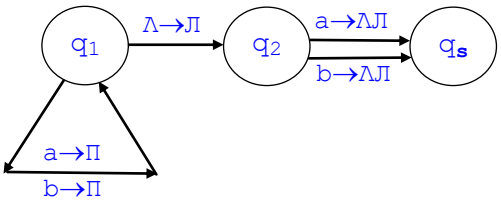
Приступим теперь к этапу кодирования. Первую подзадачу реализуем в первом состоянии q_1 машины Тьюринга, в вторую подзадачу – во втором состоянии q_2. При записи программы (показана слева) для команд в клетках сразу будем использовать сокращённые условные обозначения. Задача такая простая, что отладку здесь можно провести «в уме». Отметим, что для второй спецификации, когда для пустого входного слова на ленте остаётся диагностика-символ `'*'`, клетку `!` надо просто заменить на клетку `*!`.

$aq_1 \to$ П
$bq_1 \to$ П
$\Lambda q_1 \to$ Лq_2
$aq_2 \to$ Λ!
$\Lambda q_2 \to$!

Запись алгоритма машины Тьюринга в виде таблицы удобна для программиста и наглядна. Отметим, однако, что таблицу трудно использовать в качестве входного слова для другого алгоритма, и сложно подать на свой вход (например, для проверки самоприменимости). В связи с этим существует и альтернативные формы записи алгоритма для машины Тьюринга. Например, это привычная нам форма записи в виде обычного текста, состоящего из строк. Число строк в этом тексте равно числу клеток в таблице, например, слева показана программа нашего последнего примера.

Знак $\to$ будем считать служебным символом-разделителем, если не использовать сокращений и в каждой клетке писать все три команды, то знак $\to$ можно не указывать Постарайтесь понять, как исполнитель будет обрабатывать входное слово по такой программе.

Другой наглядный (хотя и громоздкий) способ записи программы для машины Тьюринга в виде так называемой диаграммы переходов (между состояниями). В этом случае программа записывается в виде ориентированного графа, узлы которого являются состояниями, а дуги задают переходы при обработки символов с ленты. Например, приведённая выше программа будет при этом записываться в таком виде (приходится через q_s обозначать состояние останова):



Рассмотрим теперь новую задачу. В алфавите `A={0,1,2,3,4,5,6,7,8,9}` любое непустое слово на ленте будет просто записью (неотрицательного) целого числа. Требуется составить алгоритм, который прибавляет к этому числу единицу. Мы осознаём задачу так: исходное число стереть и оставить вместо него число, на единицу большее.[1] Особым случаем, естественно, будет пустое слово. Пустых чисел в арифметике не бывает, и можно принять, например, одно из двух решений. Во-первых, можно считать пустое слово по определению нулём, и выдать в качестве ответа единицу. Во-вторых, можно считать это ошибкой и выдать диагностику (например в виде символа `'*'`). Мы примем первое решение.

Разобьём нашу задачу на такие подзадачи.

1. Поставить головку на последнюю цифру.
2. Если последняя цифра меньше 9, то прибавить к ней единицу и остановиться (пустую клетку мы приравниваем к цифре 0), иначе сдвинуться на предыдущую цифру и повторить пункт 2.

Теперь напишем саму программу.

	0	1	2	3	4	5	6	7	8	9	Λ
q_1	П	П	П	П	П	П	П	П	П	П	Лq_2
q_2	1!	2!	3!	4!	5!	6!	7!	8!	9!	0Л	1!

Сами отладьте эту программу на нескольких тестах.

> Задача для самостоятельной работы. Модифицируйте программу, когда в числе может находиться точка, отделяющая целую часть от дробной, а перед числом может стоять (а может и не стоять) знак + или –.

В качестве последнего примера рассмотрим такую задачу. В алфавите `A={a,b}` удвоить входное слово, т.е. приписать к нему (в начале или в конце) точно такое же слово. Пустое слово, естественно, будем оставлять пустым. Сделаем такое разбиение на подзадачи.

1. Двигаясь по слову слева направо, находим первую букву `a` или `b`, а если достигли пустой клетки (т.е. буквы `a` и `b` не найдены), то сдвигаемся на одну клетку влево и переходим на пункт 6.
2. Запоминаем найденную букву `a` или `b` путём переключения в состояния q_2 или q_3, одновременно заменяя её на (служебные) символы внутреннего алфавита `A` и `B` соответственно.
3. Ставим головку на первую пустую клетку за концом слова.
4. Пишем символ `A`, если находимся в состоянии q_2, или `B`, находясь в состоянии q_3.
5. Сдвигаем головку на первый символ слова и переходим к пункту 1.
6. Движемся по слову справа налево, заменяя букву `A` на букву `a`, а букву `B` на букву `b`, а когда достигнем пустой клетки, то сдвинемся влево и остановимся.

---

[1] Так работает обычный калькулятор. Отметим, что можно осознать задачу и по другому (более сложно), например: справа от исходного числа (через одну пустую клетку) записать новое число, на единицу больше исходного и сделать его ответом, оставив на нём головку. Это, однако, нарушает правило, что в качестве ответа машина должна оставлять на ленте только одно слово.

	a	b	A	B	Λ
q_1	A П q_2	B П q_3	П	П	Л q_5
q_2	П	П	П	П	A Л q_4
q_3	П	П	П	П	B Л q_4
q_4	Л	Л	Л	Л	П q_1
q_5			a Л	b Л	П !

Теперь напишем саму программу (см. слева). Отметим, что машина Тьюринга может запоминать события, происходящие при работе алгоритма, двумя способами. Во-первых, можно записывать на ленту различные служебные символы из внутреннего алфавита. Во-вторых, можем запоминать события, переключая машину в разные состояния. Например, в первом состоянии мы заменяем букву a на букву A, а букву b на букву B и переключаемся в состояние q_2 или q_3, чтобы запомнить, какую букву мы заменили. Придя в конец слова, мы записываем туда букву A или B, чтобы запомнить, какую букву мы только что удвоили.

Заметим теперь, что в состоянии q_5 мы никогда не встретим на ленте букв a и b (все они уже заменены на A и B), чтобы отметить это, мы помечаем эти клетки серым цветом (иногда их перечёркивают ⊠). Это важно, когда мы хотим подсчитать, сколько клеток занимает реализованный алгоритм, в нашем примере это 23, а не 25 клеток. Таким образом мы можем сравнивать между собой два алгоритма, решающих одну и ту же задачу: лучшим будем считать тот, в котором меньше (не зачёркнутых) клеток.

Это оценка сложности алгоритма по занимаемой им памяти (в нашем случае в клетках, в настоящих ЭВМ в байтах). Оценим теперь сложность алгоритма по времени выполнения, за единицу примем выполнение одной клетки, чаще всего это совпадает с числом движений головки по клеткам ленты. Пусть длина слова N символов. На первом шаге, двигаясь «туда и обратно» мы прошли $2*N$ клеток, затем $2*N+2$, $2*N+4$ и так далее до $2*N+2*N$ клеток. Итого (арифметическая прогрессия) $(2*N+4*N)*N/2=3*N^2$ клеток.

	a	b	A	B	Λ
q_1	A П q_2	B П q_3			!
q_2	П	П	П	П	A Л q_4
q_3	П	П	П	П	B Л q_4
q_4	Л q_5	Л q_5	Л	Л	П q_6
q_5	Л	Л	П q_1	П q_1	
q_6			a Л	b Л	Л !

Когда скорость работы алгоритма кажется программисту недостаточной, он может выполнить этап *оптимизации* с целью уменьшения времени работы. Например, для нашего последнего алгоритма количество символов, проходимых «туда и обратно», всё время возрастает (от $2*N$ до $4*N$). Этого можно избежать, если при движении назад мы сначала определим, что буквы a и b для замены на A и B в слове ещё есть. Тогда можно двигаться назад не до начала слова, а до первой буквы A или B, при этом длина прохода по слову увеличиваться не будет! Слева показан модифицированный алгоритм, время работы снизилось до $2*N^2$, т.е. в полтора раза! Правда, добавилось одно новое состояние, а общее число «рабочих» клеток увеличилось с 23 до 25, но, может быть, оно того стоит.

Отметим такое удивительное свойство машин Тьюринга. Если поменять местами любые две строки или любые два столбца в машине Тьюринга, то алгоритм не изменится ⚠. В одном из первых языков программирования Basic все строки программы были пронумерованы, поэтому эти строки можно было располагать во внешнем файле в любом порядке, исполнитель всё равно сортировал их по возрастанию номеров строк. Но я не знаю ни одного «настоящего» языка программирования, в программе которого можно было бы менять местами столбце текста программы.

1.5. Композиция алгоритмов

Природа научных знаний такова, что малопонятные и совершенно бесполезные приобретения сегодняшнего дня становятся популярной пищей для будущих поколений.

Чарльз Бэббидж

Свойство структурности алгоритма позволяет не только разбивать шаг алгоритма на более мелкие шаги, но и наоборот, объединять два алгоритма в один. Более строго, можно сформулировать такую теорему.

Теорема. Пусть существуют два алгоритма A и B таких, что выходные данные A можно подать как входные данные для B. Последовательное применение этих алгоритмов назовём их композицией

(или суперпозицией) и обозначим $A \circ B$. Тогда существует алгоритм C, который эквивалентен композиции A и B, т.е. $\exists\ C=A \circ B$.

Слова «выходные данные A можно подать как входные данные для B» обычно предполагают, что выходной алфавит алгоритма A входит во входной алфавит алгоритма B. Кроме того отметим, что для машины Тьюринга алгоритм A (первый в суперпозиции) должен в качестве ответа оставлять на ленте одно слово, а если ответ является пустым словом, то и вся лента должна состоять из клеток с пустыми символами.

Термин «суперпозиция» пришёл из математики, где так называется применение одной функции к результату, выданному другой функцией, т.е. $f(g(x))$. Для математиков существование функции $h(x)=f(g(x))$ очевидно из самого определения функции, другое дело, что это *абстрактное* существование, не говорится как построить такую функцию $h(x)$.

Программист может легко сделать, чтобы две программы (скажем, `A.exe` и `B.exe`) работали как композиция, для этого он пишет в командной строке команду `C:>A.exe | B.exe`, при этом стандартный поток вывода программы `A.exe` направляется в качестве стандартного потока ввода программы `B.exe`. И, естественно, любой «настоящий» программист уверен, что он может написать новую программу `C.exe`, которая одна работает как композиция двух предыдущих программ (хотя сделать это иногда бывает и не так просто).[1]

Доказательство. Проведём доказательство для алгоритмов в машине Тьюринга. Пусть есть две таблицы для алгоритмов A и B. Их внутренние алфавиты обязательно имеют непустое пересечение (хотя бы по одному пустому символу Λ). Переставим столбцы в алгоритме A так, чтобы общие символы были в последних столбцах, а в алгоритме B так, чтобы общие символы были в первых столбцах, суть алгоритмов от этого не изменится. Пусть в алгоритме A есть состояния $q_1,q_2,\ldots,q_n$, а в алгоритме B состояния $q_1,q_2,\ldots,q_k$. Переименуем во всех клетках алгоритма B состояние q_1 на q_{n+2}, q_2 на q_{n+3} и т.д. до q_k на q_{n+k+2}. Суть алгоритма B при этом не измениться.

И, наконец, все клетки останова в алгоритме A изменим так, чтобы вместо останова они переключали машину Тьюринга в состояние q_{n+1}. Для случая, когда в алгоритме A нет ни одной клетки останова, то он не применим ни к одному входному слову (зацикливается, а может и ломается). В этом случае очевидно, что C≡A, так как до алгоритма B дело никогда не дойдёт, и теорема доказана.

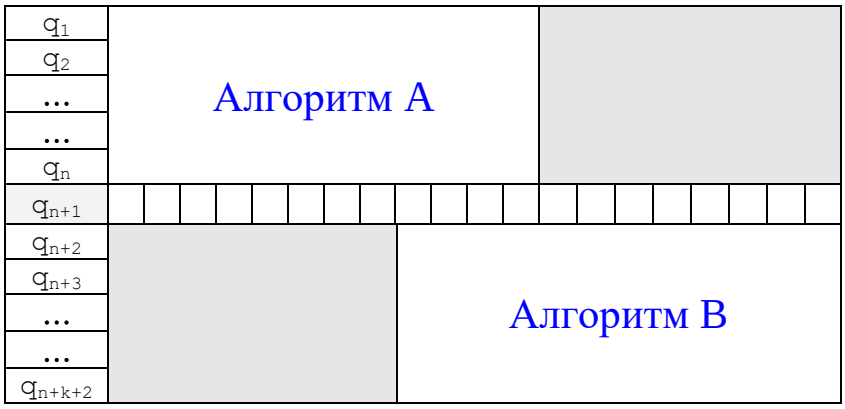


Рис. 1.4. Построение алгоритма C как композиции алгоритмов A и B.

Далее разместим таблицы A и B в общей таблице C, как показано на рис. 1.4. Те части алгоритма C, которые никогда не будут использоваться, выделены серым цветом. Как видим, в объединённой таблице алгоритма C между алгоритмами A и B мы оставили одно (служебное) состояние q_{n+1}.

Назначение этого служебного состояния в том, чтобы перевести головку, которая после работы алгоритма A находится внутри слова-ответа, на первый символ этого слова, так как потом это слово станет входным для алгоритма B. Вид этого состояния

---

[1] Не будем обращать внимание на то, что для математиков суперпозиция функций правоассоциативна, т.е. для $f(g(x))$ сначала $y=g(x)$, а потом $z=f(y)$, а для программистов $A \circ B$ левоассоциативна, т.е. для $(x)A \circ B$ будет $(x)A \to y$, а потом $(y)B \to z$.

					Λ						
q_{n+1}	Л	Л	Л	Л	Пq_{n+2}	Л	Л	Л	Л	Л	Л

Таким образом, как нам и нужно, после работы алгоритма А и выполнения состояния q_{n+1} начнёт работать алгоритм В и он «не заметит», что работает не первым, а после алгоритма А. Итак, мы доказали существование алгоритма С, просто построив его таблицу (он существует по построению).

К сожалению, наше простое доказательство работает только тогда, когда машина Тьюринга оставляет на ленте ответ, слева и справа от которого только пустые клетки. Попытайтесь понять, что алгоритм В в суперпозиции может работать неправильно, если алгоритм А оставит на ленте несколько слов, считая ответом только одно из них (в котором стоит головка).

1.6. Универсальная машина Тьюринга

Главной задачей … является поиск таких в высшей степени универсальных законов… из которых с помощью чистой дедукции можно получить картину мира.
Альберт Эйнштейн

Ранее мы уже говорили о том, что для решения каждой задачи приходится строить свою машину Тьюринга, со своим алфавитом и своим числом состояний. Это не очень удобно, и возникла идея построить такую УМТ (универсальную машину Тьюринга, UTM – universal Turing machine), которая могла бы моделировать работу всех остальных таких машин. Более строго: если на ленту УМТ поместить запись алгоритма для некоторой (моделируемой) машины Тьюринга и входное слово для этой машины, то УМТ должна оставить на ленте ответ, который должен быть выдан этой моделируемой машиной Тьюринга:

<Запись алгоритма машины М> <Входное слово для М> УМТ <Результат М>

Здесь мы не будем проводить полного доказательства существования УМТ, а лишь набросаем общую схему такого доказательства. На пути построения УМТ есть много трудностей. Во-первых, в каком алфавите должна работать УМТ, ведь у каждой машины Тьюринга алфавит свой? Здесь нам поможет приём, называемый кодированием. Пусть УМТ работает в алфавите только из двух символов $A=\{0,1\}$ (ну, конечно, там есть и обязательный пустой символ).

Символы алфавита $A=\{a_1,a_2,\ldots,a_n\}$ моделируемой машины М закодируем таким образом:

a_1 -> 101
a_2 -> 1001
…
a_n -> 100…01 (между единицами n нулей).

Видно, что мы может как закодировать, так и однозначно декодировать любое слово в нашем алфавите. После кодирования входное слово и будет записываться на ленту УМТ, а выходное слово УМТ будет декодироваться. Таким же образом закодируем и состояния:

q_1 → 101
q_2 → 1001
…
q_k → 100…01 (между единицами k нулей).

Для команд движения примем такую кодировку:

Н → 101	
П → 1001	
Л → 10001	

Для записи программы для машины Тьюринга будем использовать не таблицу, а приведённую ранее эквивалентную форму записи по одной строке на клетку таблицы:

<старый символ><текущее состояние> →
<новый символ><движение><новое состояние>

Таким образом, каждая клетка будет закодирована, например, клетка

a_2q_3 → a_3Лq_1

будет закодирована такой последовательностью (пробелы вставлены для улучшения читаемости человеком, служебная стрелка → опускается):

1001 10001 10001 10001 101

Итак, и программа конкретной машины Тьюринга, и её входное слово будут последовательностями нулей и единиц. Для того, чтобы разделить на ленте УМТ программу моделируемой машины и её входное слово, будем использовать одну пустую клетку.[1]

Λ	<программа M>	Λ	<входное слово M>	Λ

Ясно, что УМТ должна последовательно читать символы входного слова моделируемой машины, искать запись клетки для обработки этого символа, и выполнять все три команды из записи в этой клетке.

Второй серьёзной проблемой является потенциально бесконечный рост обрабатываемого слова. Ясно, что при этом символы слова вполне могут перейти на запись самой программы моделируемой машины и начать её затирать (портить).

Аналогичная проблема затирания программы была и на старых 16-битных процессорах фирмы Intel. Например, в системе программирования с языком Турбо-Паскаль 3.0 можно было описать массив `x:array[1..10] of integer;` из 10 чисел, а присаивать `x[1000]:=0` (это называется выходом индекса за границы массива). При этом какая-то команда программы затиралась нулём, и далее могли наблюдаться интересные эффекты 🙂. Современные процессоры позволяют закрывать область памяти программы на запись, и при этом будет просто исключительная (аварийная) ситуация.

Чтобы справиться с этой проблемой в УМТ, сделаем из нашей одной бесконечной ленты две аналогичные ленты. Под первую из этих лент отведём все клетки исходной ленты, следующие через одну друг за другом, а под вторую ленту – остальные клетки (они выделены серым цветом):

На первую ленту поместим запись программы, а на вторую – входное слово моделируемой машины Тьюринга. Ясно, что теперь при движении налево и направо УМТ будет двигаться не на одну, а сразу на две клетки. Очевидно, что в этом случае обрабатываемое слово уже никак не сможет испортить программу моделируемой машины (их миры не пересекаются 🙂).

Впервые теорему о том, что УМТ существует, сформулировал и доказал сам А. Тьюринг в 1947 году. Интересно, что построенная им УМТ моделировала другие машины не более, чем с квадратичным замедлением, т.е. если обычная машина Тьюринга обрабатывает входное слово длины N за t шагов, то УМТ за $O(t^2)$ шагов.

УМТ имеет очень широкое применение в теории алгоритмов. Как известно, тезис Тьюринга гласит, что для любого алгоритма обработки слов можно построить эквивалентную машину Тьюринга. Как уже говорилось, доказать этот тезис нельзя, так как неясно, что такое «любой алгоритм». Можно, однако, попытаться опровергнуть этот тезис, построив алгоритмическую систему, способную решать задачи, недоступные машине Тьюринга. И понятно, что поиск таких систем стал активно вестись сразу после провозглашения тезиса Тьюринга.

Сначала попытались модифицировать машину Тьюринга. Почему по ленте должна двигаться только одна головка? Сделаем сразу N головок, независимо друг от друга обрабатывающих символы на ленте (многоголовочная машина Тьюринга). Эти головки могут работать либо по очереди, либо независимо друг от друга (параллельно, асинхронно). Надо только определить, что делать, если две и более головки будут одновременно писать символы в одну и ту же клетку ленты. Например, можно перенумеровать головки и выполнять такую запись последовательно, по возрастанию номеров головок (при записи в одну клетку головки выстраиваются в очередь). Может быть такая машина сможет решать задачи, которые «обычная» машина Тьюринга решить не в состоянии? Как Вы догадываетесь, вскоре была предложена модификация УМТ, которая моделировала работу такой многоголовочной машины.

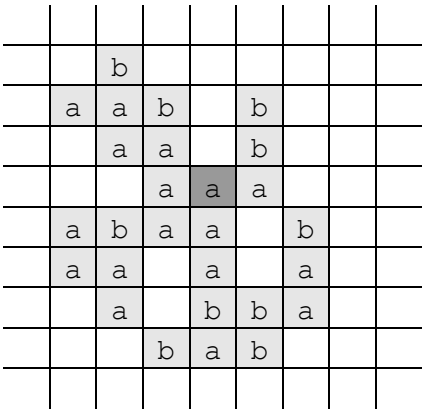
Затем была предложена машина Тьюринга, у которой не одна, а N лент, и на каждой своя головка. Однако, и для этого случая можно построить модификацию УМТ. Мы уже умеем делить одну по-

---

[1] Как видим, в алгоритмической системе для УМТ её входными данными являются два слова на ленте. Впрочем, вместо пустой клутки между этими словами можно поставить и какой-нибудь закодированный спецсимвол.

тенциально бесконечную ленту на две, выбирая клетки через одну. Ясно, что можно разделить одну ленту и на N+1 бесконечных лент, выделяя для них клетки с шагом N+1. Ничего принципиально не изменится и в случае, если дать машине Тьюринга $N_т$ независимых таблиц, к каждой таблице по $N_{тл}$ лент и по каждой ленте пустить $N_{тлг}$ головок. УМТ по прежнему работает (конечно, всё медленнее и медленнее 🙁).

Дальше фантазия учёных разыгралась. Давайте заставим головку двигаться не по ленте, а по разбитой на клетки с символами бесконечной плоскости, добавив команды движения В (Вверх) и З (вниЗ). Ответом такой машины будем считать связное множество клеток с непустыми символами. Для большей выразительности пустой символ изображён как пустая клетка, положение головки отмечено более тёмной клеткой. Ниже показан пример работы такой машины в алфавите `A={a,b}`.



Может быть такая машина будет решать новые задачи, недоступные обычной машине Тьюринга? Как Вы уже поняли, нам надо взаимно однозначно отобразить клетки ленты и клетки плоскости. Из курса по математическому анализу известно о таком отображении (так называемом диагональном методе), так что модифицированная УМТ снова работает. Ничего в принципе не изменится, если от плоскости перейти к трёхмерному пространству из кубиков с символами, и далее к пространствам любого числа измерений [1].

Следующий шаг на этом пути – так называемая недетерминированная машина Тьюринга. У такой машины в таблице на пересечении столбца, озаглавленного символом алфавита, и строки с текущим состоянием, может находиться не одна клетка программы, а целый набор («стопка») таких клеток. Каждая клетка, как известно, задаёт три операции: запись символа в текущую клетку ленты, движение головки по ленте на одну позицию и переключение в новое состояние. Так вот, в недетерминированной машине Тьюринга все эти клетки такой «стопки» начинают выполняться одновременно. В этот момент для единой таблицы происходит порождение нескольких копий текущей ленты, у каждой своя головка. Далее все головки параллельно работают со своими лентами (и, вообще говоря, с разной скоростью). Следует понять, что это происходит при *каждом* выполнении любой клетки, и, таким образом, число лент и головок может быстро возрастать [vii] [см. сноску в конце главы].

Работа недетерминированной машины Тьюринга завершается, когда любая из параллельно работающих таких вычислительных потоков попадает в клетку останова, при этом позиция головки на соответствующей ленте определяет ответ. Понятно что при работе такой машины порождается дерево вычислительных потоков, число которых может быстро и неограниченно возрастать. Ясно, что при запусках с одинаковыми входными данными, такая машина может давать *разные* ответы, что и означает, что она *недетерминированная* (и, вообще говоря, не задаёт алгоритм в традиционном смысле). Можно, однако, сделать такую машину детерминированной, если случайные величины, управляющей работой всех головок, сделать дополнительными входными данными такой машины.

Рассмотрим, например, использование этой машины для решения задачи поиска выхода из лабиринта. В каждой точке, где лабиринт разветвляется на несколько путей, мы ставим в программе соответствующее число клеток, и начинаем параллельно двигаться по всем возможным путям в лабиринте, при этом первая же задача, которая найдёт выход из лабиринта, останавливает работу машины.

---

[1] Это наглядно показывает, как слабо мы представляем, что такое бесконечное (даже счётное) множество, оно может содержать в себе и бесконечное (счётное) число счётных подмножеств. Учим матан 🙂.

Следующим шагом в наших «ухищрениях» будет квантовая машина Тьюринга, которая является дальнейшей модификацией недетерминированной машины, её мы рассматривать не будем. Скажем лишь, что можно построить УМТ, которая эмулирует работу и этой машины, так что даже «модная» квантовая ЭВМ не может решать задачи, недоступные нашей «обычной» машине Тьюринга. Как видим, машина Тьюринга такая мощная, что может моделировать работу даже неалгоритмических вычислительных систем. Для интереса отметим, что сейчас уже есть и нейронная машина Тьюринга (Neural Turing Machine), основанная на объединении идей машины Тьюринга и машины фон Неймана.

Итак, (пока?) не удаётся построить более мощную систему обработки слов, чем машина Тьюринга. Заметим, что построить менее мощную систему легко. Например, в такой системе можно запретить головке двигаться, скажем, налево. В качестве упражнения придумайте задачу, которую обычная машина Тьюринга решает, а такая «ущербная» нет.

Итак, тезис Тьюринга утверждает, что любой алгоритм обработки слов может быть реализован в виде эквивалентной машины Тьюринга. В частности, в виде машины Тьюринга может быть записана любая программа (на любом языке программирования). [viii] [см. сноску в конце главы]

1.7. Машина Тьюринга и вычислимые функции

В сущности, самым плодотворным следствием появления машин Тьюринга стало построение, изучение и вычисление так называемых вычислимых функций, т.е. программирование для компьютера.

Алан Перлис,
первый лауреат премии Тьюринга

Как известно, входными и выходными данными машины Тьюринга выступают слова из символов в некотором алфавите. Пусть в алфавите машины Тьюринга N «обычных» символов и один пустой, перенумеруем их всех целыми числами от 0 (для пустого) до N. Тогда можно рассматривать входное и выходное слово как целые числа в позиционной системе счисления с основанием N+1. Например, для входного слова $a_k a_{k-1} \dots a_1 a_0$ (нам сейчас удобно нумеровать символы слова именно в таком порядке, справа налево) это будет число

$$ord(a_k)(N+1)^k+ord(a_{k-1})(N+1)^{k-1}+ \dots ord(a_1)(N+1)+ord(a_0),$$

где $ord(a_i)$ обозначает порядковый номер символа a_i в алфавите (начиная с нуля).

Таким образом, можно считать, что каждая машина Тьюринга определяет целочисленную функцию с одним аргументом. Множество всех входных слов, к которым машина Тьюринга применима, будет, естественно, областью определения нашей функции (в математике это называется ОДЗ – область допустимых значений). Получается, что эта функция для любого значения из области определения может (за конечное число шагов) вычислить свой результат. Такие функции, естественно, называются вычислимыми, для вычислимой функции всегда существует вычисляющий её алгоритм.

Более строго, в математике их принято называть частично вычислимыми функциями. Здесь имеется в виду, что на аргументах вне ОДЗ функцию, естественно, вычислить невозможно. Кроме того, легко написать машину Тьюринга, которая не применима ни к одному входному слову, а в математике не принято рассматривать функции с пустой областью определения.

Множество целых чисел называется перечислимым, если существует алгоритм, который ничего не вводит и выдаёт (в произвольном порядке) по одному разу все элементы такого перечислимого множества. Это «плохой» алгоритм, для конечного перечислимого множества он выводит все его элементы, а потом не останавливается, а «зависает».

> Доказана теорема, что множество чисел, входящих в область определения каждой вычислимой функции, перечислимо.[1]

Множество целых чисел называется разрешимым, если существует алгоритм, который для каждого целого числа выдаёт, принадлежит ли оно этому множеству, или нет. Например, для каждого

[1] Одна из первых работ Тьюринга на эту тему так и называлась: «О вычислимых числах в приложении к проблеме разрешения».

числа этот алгоритм будет выдавать букву Д, если число принадлежит искомому множеству, или Н, если не принадлежит.

Доказана теорема, что каждое разрешимое множество целых чисел перечислимо.

Обратное неверно, существуют перечислимые, но не разрешимые множества (вспомним о «плохом» алгоритме в определении перечислимого множества). К сожалению, подмножество разрешимого множества может не быть разрешимым (разрешающий алгоритм сможет определить, что некоторое значение принадлежит всему множеству, но не сможет, что принадлежит ещё и подмножеству).

Совокупность всех разрешимых множеств (а, следовательно, и совокупность всех вычислимых функций, а значит и совокупность всех алгоритмов) является счётным множеством. А вот совокупность всех не перечислимых множеств имеет мощность континуум, а так как на каждом таком множестве может быть определена своя функция, то число всех функций *несчётно*. Печально, что почти все функции (от одной целой переменной!) «запрограммировать» не получится 😕.

Более подробно эти темы изучаются в курсе по дискретной математике.

1.8. Нормальные алгоритмы Маркова

Три пути ведут к знанию: путь размышления – это путь самый благородный, путь подражания – это путь самый лёгкий, и путь опыта – это путь самый горький.

Конфуций, V век до н.э.

А.А. Марков (1903-79)

Когда атака на тезис Тьюринга путём модификаций самой машины Тьюринга провалилась, то стали создаваться принципиально другие алгоритмические системы. Самой известной из них являются Нормальные Алгоритмы Маркова (НАМ). Эту интересную формализацию понятия алгоритма сделал в 1954 году отечественный математик Андрей Андреевич Марков (младший). Его исполнитель алгоритмов базируется на совершенно других элементарных действиях по преобразованию слов, и совсем не походит на машину Тьюринга. В отличие от Тьюринга, Марков был «чистым» математиком. Ему были чужды такие «механические» понятия, как читающая и пишущая головка, что-то, похожее на телеграфную ленту, переключатель состояний, как в стиральной машине и т.д. Можно сказать, что его машина описана на чисто математическом языке.

Как и у Тьюринга, входные и выходные данные НАМ являются словами в некотором алфавите, однако никакой «пустой» буквы там нет, вместо этого есть *пустое слово*, не содержащее ни одной буквы. Для математика и программиста более близки и очевидны понятия пустого множества, пустого файла, пустой строки и т.д., чем какого-то «пустого символа», который не является даже пробелом (мы знаем, что пробел входит в «нормальные» алфавиты и разделяет слова в предложении). Слова НАМ, конечно же, не располагаются ни на какой ленте. Как мы вскоре увидим, более всего они походят на обычные слова, расположенные на экране какого-нибудь текстового редактора.

Исполнитель НАМ может выполнять над словами всего два действия, будем иллюстрировать эти действия в алфавите из двух символов `A={a,b}`.

- Искать вхождение одного слово в состав другого. Например, слово `ab` входит в слова `abb`, `bab` и `babab`, но не входит в слово `baaa`. То, что слово может входить более одного раза, НАМ не волнует, он всегда находит только первое вхождение. Заметим, что пустое слово стоит перед и после любого символа, однако, так как ищется только первое вхождение, то пустое слово всегда найдётся перед первым символом. Кроме того, по определению, пустое слово входит в другое пустое слово.
- Производить замену найденного вхождения (только первого!) слова на любое другое заданное слово. Например, если для слова `babab` заменить первое вхождение `ab` на `b`, то получится слово `bbab`. Как видим, получилось слово на одну букву короче, причём остальные буквы «сомкнули ряды» и никаких пустых мест между ними не осталось. Наоборот, если вхождение `ab` заменить на слово `bab`, то получится слово на одну букву длиннее, остальные буквы раздвинутся.

Для современных пользователей ЭВМ это самое обычное дело, эти два действия часто выполняются для операции поиска и замены в текстовых процессорах. Ясно, что при замене на пустое слово найденное вхождение просто удалялось. А вот при замене пустого вхождения (как мы помним, оно всегда стоит в самом начале), заданное слово просто приписывалось в начало. Например, если для слова `bababab` заменить пустое вхождение на `aa`, то получится слово `aababab`.

Во времена А.А. Маркова экранных редакторов текста, конечно, не было, да и сами ЭВМ были в диковинку, а ввод данных для них чаще всего производился с картонных перфокарт. Однако для образованных людей того времени такое поведение слов текста при операциях вставки и замены было хорошо известно. Таким образом исправлялся текст в печатных вёрстках статей и книг. Каждая строка вёрстки состояла из отдельных свинцовых буковок-символов, поджатых в строке пружинкой. Оттягивая пружинку, типографский рабочий удалял или вставлял нужное количество символов.[1]

Алгоритм для исполнителя НАМ записывается в виде непустого набора так называемых правил подстановки. Эти правила, в отличие от состояний машины Тьюринга, считаются *упорядоченными* (пронумерованными), хотя в явном виде эти номера ставить не принято. Запишем правила в виде:

$$\begin{aligned} \alpha_1 &\to \beta_1 \\ \alpha_2 &\to \beta_2 \\ &\dots \\ \alpha_n &\to \beta_n \end{aligned}$$

Здесь α_i и β_i называются соответственно левыми и правыми частями правил подстановки, они могут быть любыми (в том числе и пустыми) словами в алфавите НАМ. Стрелка между ними является служебным символом и не может входить в α_i и β_i, но, как ни странно, может входить во входное слово. Как мы вскоре увидим, работа исполнителя НАМ от этого не изменится, так как стрелка не будет участвовать в операциях поиска и замены. На самом деле служебных стрелок два вида: обычная $\to$ (она называется нетерминальной стрелкой) и «стрелка с хвостиком» $\mapsto$ (терминальная стрелка). Смысл этих названий вскоре станет понятен.

Опишем теперь один шаг работы НАМ, все такие шаги будут одинаковы. Обозначим как W входное слово НАМ. Сначала исполнитель последовательно проверяет, входят ли левые части правил α_1, α_2 и т.д. до α_n во входное слово W. Если W не содержит ни одного α_i, то НАМ останавливается, и объявляет W ответом. В противном случае найдётся (первое) правило, в котором α_i входит в W, т.е. W имеет вид $W_1\alpha_i W_2$. В частном случае W_1 и W_2 могут быть и пустыми словами.

Найдя вхождение, исполнитель выполняет второе действие: заменяет в слове W (первое) вхождение α_i на β_i, получая слово $W' = W_1\beta_i W_2$. Далее проверяется, какая стрелка стояла в найденном правиле подстановки. Если это нетерминальная стрелка $\to$, то шаг работы считается завершённым, слово W' объявляется новым *входным* словом, и исполнитель переходит к следующему шагу своей работы, который полностью идентичен предыдущему. Обратите внимание, что на следующем шаге правила снова начинают просматриваться, начиная с первого. Когда же в найденном правиле стоит терминальная стрелка $\mapsto$, то исполнитель останавливается и объявляет слово W' ответом. Таким образом, в отличие от машины Тьюринга, в НАМ есть два различных способа остановки работы.

Рассмотрим несколько задач для НАМ. Сначала рассмотрим ту же задачу, которую мы делали первой в машине Тьюринга: удалить последний символ слова в алфавите $A=\{a,b\}$. В результате непустое слово должно становиться на один символ короче. Как и прежде, сделаем спецификацию, что пустое слово остаётся пустым.

Первой проблемой для нас станет нахождение последнего символа слова, ведь никакой «головки», которую можно двигать вдоль слова, теперь нет. Мы, однако, может сделать *модель* такой головки с помощью служебного символа, например, звёздочки `*`. Разобьём задачу на такие подзадачи:

1) припишем звёздочку в начало слова;
2) сдвинем звёздочку за последний символ слова (т.е. поставим её в конец слова);
3) удалим звёздочку вместе с одним стоящим перед ней символом и остановимся.

---

[1] После внесения всех исправлений каждая страница книги отливалась в отдельный лист, с которого и печаталось много копий.

Проще всего выполнить первую подзадачу, для этого нам подойдёт правило подстановки `→*`, которое заменяет пустое вхождение на символ `*`. Для движения звёздочки в конец слова нам подойдут два правила:

`*a → a*`

`*b → b*`

И, наконец, третья подзадача тоже решается двумя правилами:

`a* ↦`

`b* ↦`

В результате получается такая программа для НАМ.

```
1).  *a → a*
2).  *b → b*
3).  a* ↦
4).  b* ↦
5).     → *
```

В дальнейших примерах ставить номера у правил больше не будем. Для отладки, возьмём, например, слово `baba`, тогда получим такую цепочку преобразований (в программировании это называется трассировкой программы):

`baba ⇒ *baba ⇒ b*aba ⇒ ba*ba ⇒ bab*a ⇒ baba* ⇒ bab`

Вроде, всё хорошо, но если взять для отладки особый случай – пустое входное слово, то получим бесконечную цепочку подстановок

`⇒ * ⇒ ** ⇒ ***` …

Зациклились 🙁. Чтобы исправить ошибку, вставим в программу между 4-м и 5-м новое правило

4а). `* ↦`

Оно останавливает работу для пустого входного слова. Вот теперь, вроде, всё работает. Обратим здесь внимание на типичную ошибку при работе с НАМ. Правило `5). → *` (с пустой левой частью) может быть только одно и стоять в программе только последним. Действительно, такое правило всегда применимо, и все стоящие позже правила никогда просматриваться не будут.[1]

```
*a → +
*b → -
+a → a+
-a → a-
+b → b+
-b → b-
 + ↦ a
 - ↦ b
 * ↦
   → *
```

Решим теперь следующую задачу: первый символ слова в алфавите `A={a,b}` перенести из начала слова в его конец. Ясно, что длина слова при этом не изменится. Пустое слово, как и прежде, будем оставлять без изменений. Первое, что приходит в голову, это разбить задачу на такие подзадачи:

1). припишем звёздочку в начало слова;
2). удалим звёздочку вместе с первой буквой и запомним, какую букву удалили;
3). идём в конец слова;
4). пишем в конце слова запомненную букву и останавливаемся.

Теперь у нас нет переключателя состояний, как в машине Тьюринга, и единственный способ что-то запомнить, это писать внутрь слова служебные символы. Разумеется, как и в машине Тьюринга, в конце работы все служебные символы должны быть из слова удалены. Например, вместо стёртой первой буквы `a` запишем символ `+`, а вместо `b` символ `-`. Идти символами (в нашем случае это `+` и `-`) в конец слова мы уже умеем. Тогда получается показанная слева программа, в ней 10 правил подстановки. Самостоятельно сделайте отладку этой программы и убедитесь, что в ней нет ошибок.

Теперь поймём, что у задачи может быть несколько алгоритмов её решения. Для этого вернёмся на этап пошаговой детализации и попытаемся сделать другое разбиение на подзадачи:

---

[1] Для программистов аналогичная ошибка выглядит так:
goto `L; X:=1;` { оператор `X:=1` НИКОГДА выполняться не будет }

```
*aa → a*a
*ab → b*a
*ba → a*b
*bb → b*b
   * ↦
     → *
```

1). припишем звёздочку в начало слова;
2). заставим звёздочку вместе со следующей за ней буквой «перепрыгивать» в слове через одну букву, пока прыгать будет уже некуда;
3). удаляем из слова звёздочку и останавливаемся.

Получается показанная слева новая версия программы для решения нашей задачи. Удивительно, но при таком разбиении на подзадачи длина программы сократилась с 10 всего до 6 правил подстановки, т.е. на 40% ⚠. Этот пример показывает нам важность этого шага решения задачи, здесь не надо спешить и следует всё тщательно продумать.

В качестве последнего примера решим одну и ту же задачу и для машины Тьюринга, и для НАМ. Пусть надо для слова в алфавите `A={a,b}` удалить из входного слова все буквы `b` (никаких «пробелов» вместо `b` оставлять нельзя). Для Тьюринга составим такой план действий.

1) В первом состоянии будем двигаться по слову слева направо, буквы `b` будем просто стирать, первую встреченную букву `a` тоже сотрём и перейдём во второе состояние.
2) Во втором состоянии продолжаем движение по слову, буквы `a` пропускаем, а первую букву `b` заменим на `a` и перейдём в третье состояние. Если во втором состоянии букв `b` не найдено, то на место первого пустого символа ставим букву `a` и останавливаемся.
3) И, наконец, третье состояние будем использовать для того, чтобы снова поставить головку на первый символ и затем переключиться опять в первое состояние.

	a	b	Λ
q_1	Λ П q_2	Λ П	!
q_2	П	a Л q_3	a !
q_3	Л	Л	П q_1

Получается показанная слева программа, самостоятельно проведите её отладку.

А теперь поймём, что НАМ для решения этой же задачи будет состоять всего из одного простого правила `b →`. Как видим задача может сложно решаться для машины Тьюринга, но просто для НАМ. Разумеется можно придумать задачу, которая, наоборот, просто решается для Тьюринга, но сложно для НАМ.

А теперь решим «теоретическую» задачу. Пусть надо написать НАМ, который *применим* только к словам чётной длины. При осознании задачи мы должны понять, что на словах чётной длины наш алгоритм должен останавливаться (и всё равно, что он выдаст в качестве ответа!), а на словах нечётной длины – зацикливаться. Отметим, что «сломаться», как машина Тьюринга, НАМ не может. Особым случаем является пустое слово. Как математики, мы понимаем, что длина такого слова равна нулю, а так как ноль – чётное число, то мы решаем в этом случае остановиться. Слева показана возможная программа решения этой задачи.

```
b  → a
aa →
a  → a
```

Ранее мы уже говорили, что следует различать сам алгоритм и его *запись* в виде слова в некотором алфавите. Запись алгоритма можно подать на вход какого-нибудь другого алгоритма, или же на свой собственный вход. Далее мы определили, что алгоритм самоприменим, если он применим к собственной записи. Для программиста это обычное дело, надо в качестве стандартного входного потока программы использовать текстовый файл самой этой программы.[1] Если программа при этом остановится (неважно, что она выдаст), то такая программа самоприменима, иначе не самоприменима.

Легко сделать запись алгоритма НАМ, например, ниже показан один алгоритм из двух правил в алфавите `A={a,b}` и его слово-запись.

Алгоритм	Запись алгоритма в виде строки
`ba → ab` `ab → b`	`ba → ab,ab → b`

Как видим, мы просто записали все правила в одну строку друг за другом, разделяя их новым служебным символом – запятой. Легко видеть, что наш алгоритм самоприменим, на своей записи он остановится и выдаст ответ `b → b,b → b`. В качестве упражнения придумайте самоприменимый и не самоприменимый НАМ, которые состоят всего из одного правила.

А.А. Марков тоже высказал свой принцип, который гласит, что любой алгоритм обработки слов в некотором алфавите может быть нормализован, т.е. записан как алгоритм для исполнителя НАМ. Как

---

[1] Например, запустить паскалевскую программу так:
```
C:>MyProg.exe <MyProg.pas
```

Вы можете догадаться, вскоре была доказана теорема об эквивалентности машины Тьюринга и Нормальных алгоритмов Маркова.

Скорее всего, многие из Вас уже знакомы с какими-то языками программирования и смогли заметить, как алгоритм для машины Маркова не похож на привычные нам программы. В «обычных» языках алгоритм записывается в виде последовательности выполняющихся команд, каждая из которых «знает», где расположены обрабатываемые ею данные. Машина Маркова работает совсем не так. В этой машине обрабатываемые данные (входное слово) сами указывали те правила подстановки, которые должны обрабатывать входное слово ⚠. Как и для машины Тьюринга, не поток команд обрабатывает поток данных (SISD – Single Instruction Single Data), а, наоборот, поток данных (изменяемых слов) обрабатывает себя с помощью потока команд подстановкок (SDSI – Single Data Single Instruction).

Сам Марков называл выполнение своего нормального алгоритма ассоциативным исчислением. Дело в том, что с математической точки зрения НАМ задаёт так называемые конечно определенные ассоциативные системы (полугруппы), мы в эти сложности вдаваться не будем.

Машина Маркова является основой для особых, так называемых продукционных (или логических) языков программировыания, первым из которых был язык Prolog, разработанный в 1972 году.

1.9. Алгоритмически неразрешимые задачи

Перед нами открываются потрясающие возможности, замаскиро́ванные под неразрешимые проблемы.

Джон Гарднер

Ранее уже говорилось, что между алгоритмами и (частично) вычислимыми функциями может быть установлено взаимно-однозначное соответствие. Все алгоритмы образуют счётное множество, а вот все задачи – множество мощности континуум. Задачи, для которых в принципе невозможно построить алгоритм их решения и называются алгоритмически неразрешимыми.

Сформулируем одну из таких задач. Пусть надо построить такой алгоритм X, который, получая на вход запись любого алгоритма A, определяет, самоприменим ли этот A, или нет.

Теорема: алгоритм X не существует.

Доказательство (от противного). Пусть алгоритм X существует, и, получив на вход запись алгоритма A, он вырабатывает однобуквенный ответ `Д` (Да), если A самоприменим, и ответ `Н` (Нет), если не самоприменим.

Построим вспомогательный алгоритм Y, вот его запись в форме НАМ:

```
Д → Д
Н ↦ Н
```

Как видно, мы специально сделали так, чтобы выходные данные алгоритма X можно было подать на вход алгоритма Y. Тогда, как мы ранее доказали, обязательно существует алгоритм Z, который работает как композиция (суперпозиция) X◦Y, те Z=X◦Y. Исследуем, самоприменим ли алгоритм Z.

1). Пусть Z самоприменим, тогда
<запись Z> Z ⇒ <запись Z> X◦Y ⇒ <Д> Y ⇒ Зациклились, предположение 1). неверно.
2). Пусть Z не самоприменим, тогда
<запись Z> Z ⇒ <запись Z> X◦Y ⇒ <Н> Y ⇒ Останов, предположение 2). неверно.

Как видно, оба предположения неверны, поэтому делаем вывод, что алгоритм Z не существует. Однако алгоритм Y существует (мы построили его в виде НАМ), поэтому остаётся единственная возможность, что не существует алгоритм X. Теорема доказана.

Вот ещё интересная алгоритмически неразрешимая проблема для машины Тьюринга. Невозможно построить алгоритм X, который для записи произвольного алгоритма A определяет, применим ли этот A к пустой ленте, или нет. Интересно, что впервые существование алгоритмически неразрешимых проблем доказал сам Тьюринг ещё в 1936 году, когда не было ни компьютеров (по крайней мере в современной понимании), ни программ для них ⚠. Его доказательство аналогично нашему.

Впервые проблема алгоритмически неразрешимых задач возникла в 30-годах прошлого века. Тогда знаменитый математик Дэвид Гильберт описал так называемую проблему разрешения (по немецки Entscheidungsproblem). Пусть есть некоторая система аксиом, в рамках которой мы можем формулировать множество теорем. Для любой ли из этих теорем можно доказать, что она либо истинная,

либо ложная? Было доказано, что в рамках этой системы аксиом существуют теоремы, которые невозможно ни доказать, ни опровергнуть. В «программистском» понимании: всегда ли существует алгоритм, который, получив на вход описание некоторой теоремы, определяет, истинная она или ложная? Как Вы понимаете, было доказано, что такой алгоритм существует не всегда.

Отсюда следует печальный вывод, что рамках заданной системы аксиом существуют теоремы, которые мы никогда не сможет ни доказать, ни опровергнуть. Другими словами, существуют утверждения, о которых мы никогда не сможем узнать, истинные они ли ложные. Впервые эту «теорему о неполноте» доказал в 1931 году австрийский ученый Курт Гёдель, его работа по праву считается величайшим достижением современной науки: «Для любой непротиворечивой системы аксиом существует утверждение, которое в рамках принятой аксиоматической системы не может быть ни доказано, ни опровергнуто». К сожалению, это можно рассматривать и как предел развития математики, и как предел могущества человеческого разума 😕.

1.10. Машина Поста

Знание только тогда знание, когда оно приобретено усилиями своей мысли, а не памятью.

Л.Н. Толстой. *«Круг чтения»*

Эмиль Леон Пост
(1894-1954)

Абстрактная машина Поста была изобретена Эмилем Леоном Постом практически одновременно с машиной Тьюринга в 1936 году, но статья о ней была опубликована на несколько месяцев позже. Как алгоритмическая система машина Поста эквивалентна машине Тьюринга, но устроена проще и, главное, легче понимается теми, кто уже умеет программировать. Так происходит потому, что её исполнитель построен по схеме знакомой программистам машины фон Неймана, которая работает по схеме ОКОД (Один поток Команд обрабатывает Один поток Данных), по английски SISD (Single Instruction Single Data). А вот машина Тьюринга работает по совершенно другой схеме ОДОК (Один поток Данных Один поток Команд), по английски SDSI (Single Data Single Instruction) [см. сноску [ii] в конце этой главы]. Отметим, что всё это темы курса второго семестра «Архитектура ЭВМ и язык Ассемблера», здесь мы только немного познакомимся с машиной Поста.

Как и в машине Тьюринга, все данные машины Поста расположены на потенциально бесконечной ленте, разбитой на клетки, правда, в алфавите машины всего два символа: пустой (или пустая клетка, обычно для наглядности в книгах она изображается в виде □) и символ «метки», который будем изображать как V. Как и в машите Тьюринга, на одну из клеток ленты указывает головка ↑, ниже приведён пример:

□	□	□	□	□	V	V	V	V	□	□	□	□	□
				↑									

Перед началом работы головка стоит перед первой меткой (как мы помним, головка машины Тьюринга стояла на первом непустом символе ленты). Как и для машины ТЬюринга, будем считать, что входным является слово из меток, расположенной сразу справа после головки, как показано на рисунке выше.[1]

Программа для машины Поста похожа на программу для первых версий знаменитого языка Basic, созданного ещё в 1964 году. Это последовательность команд, пронумерованных возрастающими натуральными числами (номерами), обычно это последовательные натуральные числа 1, 2, 3 и т.д. Каждая команда имеет вид `n. K [m1[;m2]]`, где `n`, m_1 и m_2 – номера команд, а `K` – сама команда. Квадратные скобки показывают, что для некоторых команд один или оба номера команд после самой команды могут отсутствовать. Всего существует 6 команд K:

1. ← – движение головки влево; **goto** m_1
2. → – движение головки вправо; **goto** m_1

[1] Наверное было бы удобнее программировать на этой машине, если бы в начальный момент головка, как и в машине Тьюринга, стояла на первой метке входного слова, и только для пустого входного слова головка стояла бы на пустой клетке.

3. V – поставить метку в клетку; **goto** m_1
4. □ – стереть метку в клетке; **goto** m_1
5. ? – команда условного перехода:
   if □ **then goto** m_1 **else goto** m_2
6. ! – СТОП.

Обычный (результативный) останов производится по команде ! (СТОП), возможны аварийные (безрезультативные) остановы:

1. запись метки V в клетку, где она уже есть,
2. запись пустого символа □ в клетку, где он уже есть,
3. переход на команду с номером, которого нет в программе. [1]

1.	→ 2
2.	? 3;1
3.	V 4
4.	!

В качестве примера рассмотрим программу (показана слева), которая дописывает к (возможно пустому) слову одну метку справа (делая слово на одну метку длиннее). На языке высокого уровня эту программу можно записать в виде

repeat → **until** □; V; **halt**

1.	→ 2
2.	? 6;3
3.	→ 4
4.	? 5;1
5.	V 6
6.	!

В качестве другого примера рассмотрим программу (показана справа), которая дописывает к слову метку слева, если в исходном слове было нечётное число меток, иначе «ничего не делает». На команде №1 эта программа находится, если головка уже прошла чётное число меток, а на команде №3, когда нечётное. На языке высокого уровня получается нечто такое:

```
1: → ; if □ then goto 6;
   → ; if not □ then goto 1;
   V  ;
6: !
```

Итак, клетки ленты можно рассматривать как битовые переменные, тогда в языке нашей машины будут два оператора присваивания, две команды движения по массиву битовых переменных, команда условного перехода и команда СТОП.

Вопросы и упражнения

Скажи мне – и я забуду,
покажи мне – и я запомню,
дай мне сделать – и я пойму.
Конфуций, V век до н.э.

1. Почему нельзя дать строгое определение понятия любого алгоритма?
2. Объясните, почему так важно́ понятие исполнителя алгоритма?
3. Какие свойства обязательны (необходимы) для алгоритма?
4. В чём разница между алгоритмом и записью этого алгоритма?
5. Для чего было необходимо формализовать понятие алгоритма?
6. Чем данные отличаются от информации?
7. Какие свойства должен иметь алфавит машины Тьюринга?
8. Что такое вычислительный процесс?
9. Опишите один шаг (такт) работы машины Тьюринга.
10. Что такое клетка останова и для чего она нужна?
11. Как найти слово-ответ, если машина Тьюринга оставила на ленте несколько слов, разделённых пустыми клетками?
12. Какие условные обозначения используются при записи программы для машины Тьюринга?
13. Что такое тезис Тьюринга и почему его нельзя доказать?
14. Какие два алгоритма называются эквивалентными?
15. Какой алгоритм называется самомодифицирующимся?

---

[1] Отсутствующая метка в программе может быть, нельзя только делать на неё переход, например:
3 ? 4;5 – делается переход на номер 4, а номера 5 в программе нет.
4 !

16. В чём смысл тезиса Тьюринга?
17. Как связаны между собой входной, выходной и внутренний алфавиты машины Тьюринга?
18. Что означает клетка таблицы машины Тьюринга, которая выделена серым цветом или перечёркнута?
19. Какие свойства должен иметь любой алгоритм?
20. Опишите своими словами, что такое алгоритмическая система.
21. Из каких этапов состоит решение задачи в алгоритмической системе?
22. Чем отличается программа от программного продукта?
23. Дайте определения композиции (суперпозиции) двух алгоритмов.
24. Как можно закодировать и декодировать любой алфавит в алфавит, состоящий всего из двух символов?
25. Как доказать, что множество всех алгоритмов счётное?
26. Для чего нужна Универсальная машина Тьюринга?
27. Чем исполнитель НАМ отличается от исполнителя машины Тьюринга?
28. Когда происходит останов НАМ?
29. Чем машина Поста отличается от машины Тьюринга?

---

[i] Для продвинутых читателей. Алгоритм и его исполнитель друг без друга бесполезны. Соответственно, всё должно быть известно не только об алгоритме, но и об исполнителе надо знать все тонкости его работы. Например, запись в алгоритме `X+Y` только тогда имеет смысл, когда точно известно, как это предписание будет выполнять исполнитель (целые или вещественные числа, символьная обработка, объединение множеств и т.д.). Мы часто не понимаем этого, например, обсуждая особенности некоторой программы, мы не осознаём, что сами являемся исполнителями этого алгоритма. В частности, два программиста являются двумя *разными* исполнителями одного и того же алгоритма 🙂.

В качестве более серьёзного примера можно привести ДНК, содержащую закодированную информацию о фенотипе организма, который надо вырастить по этой информации. «Исполнителем» здесь является сложная биохимическая «фабрика» живой клетки, с которой будет взаимодействовать ДНК (например, куриное яйцо 🙂). Следовательно, разные фабрики-исполнители могут вырастить несколько разные организмы.

[ii] ***Функциональные языки программирования***

Одним из важных видов данных в функциональных языках программирования являются монады. Монада – это моноид в категории эндофункторов 🙄.

… всякая монада – это функтор, но не всякий функтор – это монада… 🅱.

Для понимания этой сноски надо достаточно хорошо знать хотя бы один «традиционный» язык программирования, здесь для примера используем Free Pascal. Лучше вернуться к этой сноске в конце данного курса, после изучения языка программирования.

Рассмотрим простую программу скалярного произведения двух векторов (иногда его ещё называют *внутренним* произведением):

```
type Mas=array[1..n] of integer;
var a,b: Mas;
function SP(var x,y: Mas; n: integer): integer;
  var i: integer;
begin SP:=0;
  for i:=1 to n do SP:=SP+x[i]*y[i]
end;
```

Сначала заметим, что в «чистых» функциональных языках нет (изменяемых) переменных, нет оператора присваивания и процедур (только функции), параметры передаются только по значению, а вместо циклов используется рекурсия. Обычно функциональный язык содержит набор элементарных функций, а также так называемые *функциональные формы*, они в чём-то похожи на рекурсивные функции обычных языков. Рассмотрим примеры функциональных форм для очень простого (учебного) функционального языка. Так, функциональная форма, имеющая «имя» `||` задаёт (рекурсивную) вставку заданной операции в последовательность данных (последовательность данных будем записывать в круглых скобках). Например, для операции «плюс»:

$\|+(1,2,3) \Rightarrow (1+\|+(2,3)) \Rightarrow (1+(2+3)) \Rightarrow (1+5) \Rightarrow 6$

Как видим, вставка левоассоциативная (производится слева направо). Видно, что функциональная форма $\|$ имеет в качестве параметра функцию, в последнем примере это элементарная операция сложения $+$.

Функциональная форма с «именем» $\circ$ определяет композицию (funcnion composition) или суперпозицию (последовательное выполнение действий):

$f \circ g \circ q(x) \Rightarrow f \circ (g \circ q)(x) \Rightarrow f \circ (g(q(x)) \Rightarrow f(g(q(x)))$

Функциональная форма с «именем» $\oplus$ порождает новую последовательность, с элементами попарной комбинации элементов двух входных последовательностей (равной длины), например:

$\oplus(1,2,3)(4,5,6) \Rightarrow ((1,4),(2,5),(3,6))$

С помощью этих функциональных форм наша функция вычисления скалярного произведения будет выглядеть так ⚠:

$\|+\|*\oplus(x,y)$

Программист понимает выполнение этого «однострочного» алгоритма таким образом:

$\|+\|*\oplus(x,y) \Rightarrow \|+\circ\|*\circ\oplus(x,y) \Rightarrow$

$\|+\circ\|*((x_1,y_1),(x_2,y_2), \dots ,(x_n,y_n)) \Rightarrow$

$\|+((x_1*y_1),\|*((x_2,y_2), \dots ,(x_n,y_n))) \Rightarrow$

$\|+((x_1*y_1),(x_2*y_2),\|*((x_3,y_3), \dots ,(x_n,y_n))) \Rightarrow \dots \Rightarrow$

$\|+((x_1*y_1),(x_2*y_2),(x_3*y_3), \dots ,(x_n*y_n)) \Rightarrow$

$((x_1*y_1)+\|+((x_2*y_2),(x_3*y_3), \dots ,(x_n*y_n)) \Rightarrow \dots \Rightarrow$

$((x_1*y_1)+(x_2*y_2)+(x_3*y_3)+ \dots +(x_n*y_n)) \Rightarrow$

$x[1]*y[1]+x[2]*y[2]+x[3]*y[3]+ \dots +x[n]*y[n]$

Надеюсь, что этот простой пример позволит Вам почувствовать всю мощь функциональных языков программирования. Разумеется «настоящие» функциональныя языки весьма сложны для изучения (см. эпиграф), но общая идея остаётся. Например, на языке APL наша программа $\|+\|*\oplus(x,y)$ запишется в весьма похожем виде как $x+.\times y$.

Больным для функциональных языков является ввод и вывод данных. Так как «чистая» функция для одинаковых аргументов всегда возвращает одно и то же значение, то ясно, что просто так написать функцию ввода не получится 🙁. Приходится идти на «хитрости», частности, для этого используются пресловутые монады, мы в эту сложную тему вдаваться не будем.

[iii] Для продвинутых читателей. Задача об усердном бобре неожиданно обнаружила сильную связь со знаменитой гипотезой Гольдбаха. Эта гипотеза утверждает, что любое чётное число больше двух является суммой двух простых чисел. Доказательство или опровержение этой гипотезы может оказаться важным событием, в нашем понимании распределения простых чисел.

В 2015 году анонимный пользователь Code Golf Addict опубликовал в системе GitHub машину Тьюринга с 27 состояниями, которая останавливается, если гипотеза Гольдбаха ложная. Таким образом, вычисление BB(27) опровергло бы эту гипотезу!

Дальше больше. В 2016 году была построена машина Тьюринга с 744 состояниями, которая останавливается только тогда, когда не выполняется знаменитая гипотеза Римана о распределении простых чисел. За решение этой проблемы объявлена награда в 1000000$ (не будем говорить, в чём заключается эта гипотеза, Интернет Вам в помощь 🙂).

Таким образом за вопросом «Остановится ли машина Тьюринга?» скрываются многие принципиальные вопросы «чистой» математики. Например, в 2016 году построена машина Тьюринга с 748 состояниями, которая остановиться только тогда, когда «теория множеств ZF противоречит самой себе» 🙄. Интересно, что сначала эта задача была решена с помошью программы с 7910 состояниями, но потом был найден более эффективный алгоритм с 748 состояниями. Ожидаем талантливого программиста на машине Тьюринга 🙂, который построит ещё более эффективный алгоритм (подозревают, что можно снизит сложнось этого алгоритма до нескольких десятков состояний).

[iv] Для продвинутых читателей. Наши «настоящие» ЭВМ работают по-другому: каждая задача, а конечном счёте преобразуется (разлагается на машинные команды) так, чтобы соответствовать архитектуре компьютера (быть написанной на языке именно этой ЭВМ). А вот машина Тьюринга, наоборот, изменяет свою структуру, чтобы соответствовать решаемой задаче! Информация к размышлению: а как решает задачи человек? 🙄.

Исполнитель алгоритма машина Тьюринга резко отличается от исполнителей привычных нам языков программирования. Действительно, в обычных языках алгоритм состоит из шагов (операторов языка программирования, машинных команд и т.д.). Каждый шаг знает, где находятся его подлежащие обработке данные и после выполнения однозначно определяет следующий шаг алгоритма. Итак, шаги алгоритма (операторы языка, команды ЭВМ) являются «главными», а обрабатываемые ими данные – второстепенными (подчинёнными). Для компьютеров такая схема обработки алгоритма называется ОКОД (Один поток Команд обрабатывает Один поток Данных), по английски SISD (Single Instruction Single Data). Так вот, в машине Тьюринга всё наоборот. Там именно данное (считанный с ленты символ) определяет клетку с командами (которые надо выполнить при обработке этого символа). Можно сказать, что это исполнитель ОДОК (Один поток Данных Один поток Команд), по английски SDSI (Single Data Single Instruction). Компьютеры, работающие по этой схеме, называются потоковыми ЭВМ (по-английски DFC – Data Flow Computers), конструирование таких ЭВМ сопряжено с очень большими трудностями, сейчас есть только экспериментальные образцы.

Впрочем, это темы курса по архитектурам ЭВМ второго семестра.

---

[v] Для продвинутых читателей. Можно сказать, что сутью спецификации является четкая постановка задачи. В некоторых книгах по программированию говорится, что необходимо построить *математическую модель* решаемой задачи, однако очевидно, что это относится в основном к физическим и техническим задачам.

В качестве примера построения математической модели рассмотрим такую простую задачу из школьного учебника физики. Необходимо вычислить, на какую максимальную высоту `h` поднимется тело, если его бросить вертикально вверх с начальной скоростью `v`. При осознании задачи сначала надо определить, что будет пониматься под термином «тело». Вероятно, это не может быть что-то очень легкое, вроде пушинки или песчинки, так как сопротивление воздуха не позволит нам бросить его достаточно высоко ни при какой начальной скорости. Итак, или тело бросается в пустоте, или сопротивление воздуха не учитывается, т.е. это должно быть достаточно плотное и увесистое тело, вроде камня, пули или металлического ядра.

С другой стороны, тело не должно быть и уж слишком массивным, например, весить миллиарды тонн, т.к. тогда оно само будет ощутимо притягивать Землю, а это уже совсем другая задача (по крайней мере, интуитивно понятно, что такое тело при заданной скорости будет подниматься на *меньшую* высоту). При таком осознании задачи, используя знания из школьного учебника физики, математическая модель представима в виде формулы (`g` – ускорение свободного падения на поверхности Земли):

$$h = v^2/2 \cdot g$$

Далее, важно понять, что была сделана и ещё одна спецификация задачи, которая заключается в том, что «`v` не слишком велико». Действительно, если начальная скорость тела превысит первую космическую скорость, то это тело (если не учитывать сопротивление воздуха) уже не упадет назад на Землю. Величина первой космической скорости около 8 километров в секунду, так что уже при стрельбе вверх из современного артиллерийского орудия приведенная выше математическая модель становится малопригодной.

Построение новой математической модели для достаточно больших скоростей `v` уже несколько выходит за рамки школьной математики и приводит к следующей формуле (`R` – радиус Земли):

$$h = R \cdot v^2/(2 \cdot g \cdot R - v^2)$$

Кроме того, здесь предполагается, что Земля не вращается, иначе сила Кориолиса внесёт свои поправки в результат, а притяжение Солнца и Луны не учитывается.

[vi] Для продвинутых читателей. До недавнего времени кодирование (т.е. запись алгоритма на языке программирования) было исключительно прерогативой человека. В последнее время, однако, появляются интеллектуальные системы, призванные помочь человеку-программисту в этой деятельности. Данные системы строятся на базе нейросетей, обученных на огромном объёме программного кода (на всех языках программирования). Далее, человек-программст может сформулировать, что должен делать очередной фрагмент программы, а система предложит кусок текста, решающий данную задачу. Интересно, что допускается и настройка на конкретный стиль программирования, присущий именно данному программисту (если нейросеть обучена именно на его программах).

В качестве примера можно привести системы с названиями CodeWhisperer (фирма Амазон, работает с языками Java, Javascript и Python), Copilot (фирма Microsoft, работает с языками JavaScript, Python, TypeScript, Ruby, Go, C#, C++) и Tabnine (фирма Tabnine, поддерживается более 20 языков), OpenAI Codex (фирма OpenAI Codex, работает с языками JavaScript, Go, Perl, PHP, Ruby, Swift, TypeScript, Shell).

Например, если в системе Copilot (уже предустановоен в системе Visual Studio 2022) программист введёт заголовок функции (на языке Java)

```
int calculateDaysBetweenDays
```

и обратится за помощью к системе, то она завершит текст данной функции:

```
int calculateDaysBetweenDays(Date startDate,Date endDate){
  return (int)((endDate.getTime()-startDate.getTime())/(1000*60*60*24));
```

```
}
```

Ещё пример, если для системы OpenAI Codex ввести запрос "Say Hellow, Word 5 times" получим такую программу на Питоне:

```
for i in range(5):
    print("Hellow, Word")
```

В этом году фирма Alphabet (дочка Google) выпустила систему AlphaCode, которая, по утверждению авторов, способна писать простые полные программы на уровне среднего программиста! В системе задействовано глубокое распознавания текстов запроса на естественном (английском) языке. При тестировании система соревновалась в написании простых программ по их описанию на естественном языке примерно с 5000 программистами-людьми и показала по оценке экспертов 54.3% эффективности!

В 2021 году реализован проект Doesnotexist (фирма Doesnotexist.codes), система может генерировать код на языках C и C++, система обучена на нескольких миллионах строк кода. Каждый желающий может зайти на сайт `https://doesnotexist.codes/` и попробовать угадать, написан ли показанный код системой или человеком.

В качестве ещё одного примера приведём нейросеть ChatGPT (есть доступ из интернета!), она пишет как фрагменты кода, так и полные программы на многих языках программирования, оптимизирует код, конвертирует программы с одного языка на другой, создаёт сайты по их словесному описанию и т.д. Кроме того, этот же, как говорят, чат-бот может поддерживать разговор на отвлечённые (в том числе филосовские) темы, отвечать на «каверзные» вопросы и т.д.

Похоже, что монополия программистов-людей приближается к концу 😕.

[vii] Для продвинутых читателей. С точки зрения программиста при выполнении такой клетки порождается несколько вычислительных потоков Windows, у всех таких потоков общая секция кода, но свои секции данных и стека (в начальный момент при порождении это копии из родительской задачи). Более всего этому соответствует системный вызов `fork()` в языке C, который «раздваивает» текущий поток (в ОС семейства Unix – нить). Это тема курса по архитектурам ЭВМ.

[viii] Для продвинутых читателей. После изучения машины Тьюринга у читателей, уже знакомых с программированием, может возникнуть сомнение в правильности тезиса Тьюринга. Действительно, а как быть с программами, входные данные которых определяются, например, в процессе диалога с пользователем? Ведь такие данные нельзя заранее поместить на ленту машины Тьюринга в виде входного слова. Например:

```
Write('X='); Read(X); Write('X+1=',X+1);
```

Здесь приходится делать такую модификацию машины Тьюринга, у которой в начале работы входное слово присутствует на ленте не целиком, начальное слово ограничено, скажем, служебными символами `*`. Символы могут добавляться к этому слову (скажем, слева) в процессе работы, причём подчиняясь появлению на ленте каких то (управляющих) символов, которые записываются при работе алгоритма на ленту, например, справа:

Λ	Λ	Λ	Λ	*	α_1	α_2	α_3	…	α_n	*	Λ	Λ	Λ	Λ

Пусть сформированное сейчас слово на ленте закрашено, а слева и справа служебными символами `*` помечены зоны ввода и вывода соответсвенно. Выполнив оператор `Write('X=')` машина получит такую ленту

Λ	Λ	Λ	Λ	*	α_1	α_2	α_3	…	α_n	*	X	=	Λ	Λ

После этого правая `*` перемещается в конец слова. Далее, при выполнении оператора `Read(X)` головка перемещается к левой звёздочке и машина начинает выполнять бесконечный цикл

*	Λ
Л	П

Ясно, что это будет продолжаться до тех пор, пока из «внешнего мира» на ленту перед звёздочкой не будут помещены (непустые) входные данные, например

Λ	Λ	1	2	*	α_1	α_2	α_3	…	α_n	X	=	*	Λ	Λ

Вот теперь машина может читать введённые данные и обрабатывать их. Отметим, что так же ведёт себя и «настоящая» процедура ввода `Read(X)`: при её выполнении программа переходит в состояние ожидания, пока из потока ввода не поступит порция введённых данных. После обработки будет сформирована такая лента

Λ	Λ	*	α_1	α_2	α_3	…	α_n	X	=	X	+	1	=	1	3	*	Λ	Λ

В «настоящих» программах после ввода порция данных пропадает (из буфера ввода входного потока), а наша машина Тьюринга, соответственно, просто стирает введённые слева данные.

Продвинутые читатели уже слышали, что современные программы могут выполняться в виде двух и более параллельных вычислительных потоков, обмениваясь между собой данными. Такие программы могут моделироваться машиной Тьюринга, по ленте которой перемещаются две и более независимые головки и т.д. Итак, тезис Тьюринга опровергнуть не удаётся.