

Алгоритмы и алгоритмизация

Понятие алгоритма и алгоритмического процесса

В своей повседневной деятельности каждому из нас постоянно приходится сталкиваться с разнообразными правилами, предписывающим последовательность действий, цель которых состоит в достижении некоторого необходимого результата. Подобные правила очень многочисленны. Например, мы должны следовать вполне определенной системе правил, чтобы вычислить произведение двух многозначных чисел или найти корни квадратного уравнения. Примеры такого рода можно продолжать неограниченно. Такие системы правил часто называют **алгоритмами**.

Алгоритм или алгори́зм – одно из основных понятий (категорий) математики, не обладающих формальным определением в терминах более простых понятий, а абстрагируемых непосредственно из опыта. Понятие алгоритма занимает одно из центральных мест в современной математике, прежде всего вычислительной.

Само слово “алгоритм” происходит от *algorithmi*, являющегося, в свою очередь, латинской транслитерацией арабского имени выдающегося персидского математика Абу-Джафара Мохамеда ибн-Мусы аль-Хорезми (IX век н.э.). Он изложил общие правила выполнения действий над числами, представленными в десятичной форме, которыми мы пользуемся до сих пор. Существенным было то, что эти правила могли быть применены к любым числам.

Действия согласно той или иной инструкции, являющейся алгоритмом, для получения определенного результата предполагают наличие некоторых исходных данных. Например, такими исходными данными при выполнении арифметической операции над двумя числами является эта пара чисел, а результатом – одно число. Таким образом, под алгоритмом понимается всякое точное предписание, которое задает процесс, начинающийся с произвольных исходных данных и направленный на получение полностью определенного этими исходными данными результата.

Первым свойством алгоритма является **дискретный**, т.е. пошаговый характер определяемого им процесса. Кроме исходных данных для алгоритма также должно быть определено правило, по которому процесс признается закончившимся ввиду достижения результата. При этом вовсе не предполагается, что результат будет обязательно получен, т.е. процесс применения алгоритма к конкретным исходным данным может также оборваться безрезультатно или не закончиться вовсе. Если процесс заканчивается (соответственно не заканчивается) получением результата, то в этом случае говорят, что алгоритм применим (соответственно не применим) к рассматриваемым исходным данным.

Другим важнейшим свойством алгоритма является **массовость**. Смысл данного понятия заключается в том, что существует некоторое множество объектов, которые могут служить возможными исходными данными для рассматриваемого алгоритма. Например, для алгоритмов выполнения арифметических операций – сложения, вычитания, умножения и деления – такими данными являются все действительные

или целые числа. Смысл массовости таких алгоритмов как раз и состоит в том, что они одинаково пригодны для всех случаев, т.е. они требуют лишь механического выполнения цепочки некоторых простых действий.

Отметим еще одну важную особенность, присущую каждому алгоритму. Предполагается, что алгоритм *понятен* для исполнителя, т.е. исполнитель алгоритма знает, как его выполнять. При этом исполнитель алгоритма, выполняя его, действует “механически”. Очевидно, что формулировка алгоритма должна быть настолько точна и однозначна, чтобы могла полностью определять все действия исполнителя.

Анализ алгоритмов показывает, что если применять алгоритмы повторно к одним и тем же исходным данным, то мы всегда будем получать один и тот же результат. Например, если применять алгоритм деления отрезка пополам с помощью циркуля и неразмеченной линейки к одному и тому же отрезку, то каждый раз будем получать одну и ту же точку – середину отрезка. Таким образом, можно говорить об *определенности* и *однозначности* алгоритмов.

Итак, алгоритм можно определить как систему правил, сформулированную на языке, понятном исполнителю, и определяющую последовательность действий, в результате выполнения которых мы приходим от исходных данных к искомому результату. Такая последовательность действий называется **алгоритмическим процессом**, а каждое действие – его **шагом**. Число шагов для достижения результата обязательно должно быть конечным. Кроме того, алгоритм должен обладать свойствами массовости, определенности и однозначности.

Отметим здесь, что наряду с совокупностями возможных исходных данных и возможных искомых результатов для каждого алгоритма имеется еще совокупность промежуточных результатов, представляющих собой ту рабочую среду, в которой развивается алгоритмический процесс, каждый дискретный шаг которого состоит в целенаправленной смене одного конструктивного объекта другим. Например, при применении алгоритма вычитания столбиком к паре целых чисел (307, 49) последовательно возникнут такие конструктивные объекты:

<u>307</u>	<u>307</u>	<u>307</u>	<u>307</u>	<u>307</u>
49	49	49	49	49
		8	58	258

Неправильно было бы думать, что для любой задачи существует лишь один алгоритм ее решения. Например, для подсчета количества зрителей на трибунах стадиона можно предложить несколько алгоритмов, применяя которые, будем получать один и тот же результат.

Однако далеко не всегда для задачи удается найти хоть какой-нибудь алгоритм ее решения. Так, на протяжении тысячелетий математики пытались решить задачу о квадратуре круга: с помощью циркуля и неразмеченной линейки построить квадрат, равновеликий кругу с заданным радиусом r , т.е. так, чтобы $\pi r^2 = x^2$, где x – сторона искомого квадрата. Лишь в XIX веке усилиями выдающихся математиков А.М.Лежандра и К.Ф.Линдемана была строго доказана неразрешимость этой задачи. В самом деле, из условия задачи следует, что $x = \sqrt{\pi} \cdot r$, поэтому надо осуществить

построение так, чтобы получить отрезок, длина которого в $\sqrt{\pi}$ раз больше длины данного отрезка. Оказывается “умножить” графически с помощью циркуля и неразмеченной линейки отрезок на число можно лишь тогда, когда это число является корнем алгебраического уравнения с целыми коэффициентами. Лежандр установил, что π – число иррациональное (т.е. не представимо в виде $\frac{p}{q}$, где p и q –

взаимно простые числа), а Линдемман – что оно, к тому же, и трансцендентное, т.е. не удовлетворяет никакому алгебраическому уравнению с целыми коэффициентами.

Столь же известна и другая задача древнегреческой математики – задача о трисекции угла, т.е. о делении угла на три равные части с помощью циркуля и неразмеченной линейки. Как и в случае задачи о квадратуре круга, неразрешимость этой задачи была доказана лишь в XIX веке и тоже алгебраическими методами. Заметим, что для этих знаменитых задач (и это строго доказано) не существует алгоритмов решения (т.е. задачи неразрешимы) в рамках фиксированного набора допустимых действий, в данном случае – построения только с помощью циркуля и неразмеченной линейки.

Есть также множество проблем, о которых мы и сейчас не можем сказать, разрешимы они или нет, и если разрешимы, то каким образом. Например, известна гипотеза о том, что любое четное число представимо в виде суммы двух простых чисел. Но утверждение это пока не доказано, и неясно, удастся ли его когда-либо доказать или опровергнуть.

Итак, для некоторых задач существует несколько алгоритмов их решения, для некоторых таких алгоритмов вообще не существует, и, наконец, есть задачи, для которых мы не знаем, существуют или нет алгоритмы их решения.

Алгоритмические системы

Когда речь идет о построении алгоритма решения какой-либо задачи, то мы явно или неявно предполагаем известными те объекты, которые будут исходными данными для нашего алгоритма. Например, в задаче деления отрезка пополам в качестве таких объектов выступают отрезки произвольной длины. При построении алгоритма нахождения корней алгебраического уравнения $ax^2 + bx + c = 0$ такими объектами являются тройки любых чисел (a, b, c) , определяющие коэффициенты этого уравнения.

Второе, что мы предполагаем известным, – это те действия, с помощью которых строятся шаги алгоритма. Например, в той же задаче о делении отрезка мы разрешаем использовать только действия с циркулем и неразмеченной линейкой, а в задачах решения алгебраического уравнения разрешается выполнять действия сложения, вычитания, умножения, деления и извлечения квадратного корня. Иными словами, мы предполагаем известными все возможности исполнителя алгоритма.

Мы также должны знать, как формулировать шаги заданий, чтобы их понял исполнитель, т.е. знать, какой язык понятен исполнителю алгоритма. Наконец, при построении алгоритма решения какой-либо конкретной задачи мы должны знать, что собой будет представлять результат: число, точку на прямой и т.д., т.е. мы должны знать множество объектов, к которым принадлежит результат. Однако с помощью циркуля и линейки решается не только задача деления отрезка пополам, но и целый ряд геометрических построений. С помощью арифметических операций также строятся алгоритмы решения самых различных задач.

Набор средств и понятий, позволяющих строить не один алгоритм, а множество алгоритмов, решающих различные задачи, будем называть **алгоритмической системой**. Алгоритмических систем может быть много и каждая из них определяется:

- множеством входных объектов, подлежащих обработке алгоритмами данной системы;
- свойствами исполнителя алгоритмов, т.е. набором тех действий, которые может выполнять исполнитель;
- множеством выходных объектов, представляющих собой искомые результаты выполнения алгоритмов данной системы;
- языком, на котором формулируются алгоритмы, адресованные исполнителю.

Заметим, что множество входных объектов (исходных данных) и множество выходных объектов (результатов) часто совпадают. Например, в алгоритмах вычислений арифметических выражений и исходные данные, и результаты – числа. Но в ряде случаев исходные данные и результаты могут оказаться совершенно разной природы. Например, исходными данными алгоритма поиска номера телефона в справочнике служат фамилия абонента и его адрес места жительства.

Что касается действий, которые может выполнять исполнитель, то они также могут резко отличаться в различных алгоритмических системах. Язык алгоритмической системы тесно связан с исполнителем и не должен включать в

свой состав указаний на недопустимые или невозможные для исполнителя действия, а также обращения к входным объектам, если они не принадлежат алгоритмической системе (это же касается и результатов). Язык должен быть точно понят исполнителем.

Если исполнитель алгоритма – человек, то в качестве языка для формулировки алгоритмов вполне можно использовать естественный язык. Зачастую естественный язык служит основой для другого формального языка – метаязыка или псевдокода.

Если исполнитель алгоритма – компьютер, набор действий которого весьма ограничен, то соответственно язык, на котором задаются шаги заданий, значительно беднее, чем язык, адресованный человеку. В то же время такой язык более точен, его предложения не допускают различных толкований и алгоритм, написанный на таком языке, представляет собой последовательность команд (инструкций), адресованных различным устройствам компьютера, совсем не похожую на привычные для нас фразы естественного языка. По своей сути язык компьютера, называемый также машинным языком, формален, а то, что он все-таки назван языком, имеет под собой довольно глубокое основание. Несмотря на несхожесть его с естественным языком, он имеет свой алфавит, свою грамматику.

Алгоритм, адресованный компьютеру как исполнителю, представляет собой некий текст, т.е. некоторую последовательность символов. Этот текст называют программой, а язык, с помощью которого записывают программу, соответственно называют языком программирования. В связи с этим отметим здесь одно очень важное обстоятельство – если мы имеем дело с алгоритмической системой, предназначенной для составления алгоритмов обработки данных, т.е. любых символьных последовательностей, то открывается принципиально важная возможность составлять алгоритмы, которые в свою очередь будут преобразовывать алгоритмы, обрабатывая тексты, реализующие эти алгоритмы. В частности, именно таким образом системные программы – трансляторы – преобразуют программу, написанную на одном языке программирования, в программу, написанную на другом языке. При решении задач с помощью компьютера автоматическое преобразование алгоритмов из одной формы в другую (как делают, например, трансляторы и компиляторы) или автоматическое изменение алгоритмов в ходе вычислений (как делают, например, интерпретаторы) представляется чрезвычайно важным. Это как раз и создает ту удивительную логическую гибкость, которая превратила компьютер в принципиально новый инструмент обработки данных.

Понятие о математической модели

Мощным инструментом познания мира является математическое моделирование. **Математическая модель** – приближенное описание какого-либо класса явлений внешнего мира, выраженное с помощью математической символики. Чтобы описать явление, необходимо выявить самые существенные его свойства, закономерности, внутренние связи, роль отдельных характеристик явления. Выделив наиболее важные факторы, влияющие на происходящие процессы, можно пренебречь менее существенными. Тем самым несколько упрощается и огрубляется модель явления.

Учет важнейших факторов, влияющих на изучаемое явление, и взаимосвязей этих факторов производится с помощью математической символики, как правило, в виде уравнений и неравенств. Получившиеся математические соотношения в совокупности с некоторыми известными исходными данными и образуют математическую модель явления.

Итак, математическая модель представляет собой аналог явления, сохраняющий его существенные черты и служащий для его изучения.

Процесс построения математической модели обычно состоит из нескольких взаимосвязанных этапов, каждый из которых, как в целом и сам процесс, по своей сути являются итеративными.

Первый этап состоит в том, что выделяются основные закономерности, характеризующие моделируемое явление. Эти закономерности зачастую носят гипотетический характер. Затем закономерности облакаются в математическую форму. Однако на этом этапе возникает целый ряд сложностей. С одной стороны, хотелось бы выявить и формализовать все факторы, влияющие на изучаемое явление. С другой стороны, запись их в виде уравнений и неравенств обычно сопряжена с большими сложностями. Дело в том, что информацию для составления математических соотношений, как правило, приходится черпать из наблюдений. Результаты же наблюдений всегда сопровождаются некоторыми погрешностями, которые тем самым привносятся и в математические соотношения. Как следствие, может оказаться, что решение одних уравнений или неравенств, характеризующих данное явление, не удовлетворяет другим соотношениям той же самой модели. Конечно же, такая модель должна быть изменена, т.е. на этом этапе главное – выявить все основные закономерности явления, максимально точно замерить все его параметры, записать все это в математической форме. При этом необходимо следить за тем, чтобы получившиеся соотношения были совместными, т.е. чтобы существовало решение задачи.

Следующий этап состоит в решении поставленной математической задачи. Решив задачу, необходимо проанализировать результаты ее решения. Если окажется, что результаты вычислений не соответствуют наблюдениям, то такая модель требует изменений. Придется воспользоваться какими-то другими гипотезами, написать соответствующие математические соотношения, несколько иначе описывающие исследуемое явление, т.е. на самом деле построить новую математическую модель. Может случиться, что и на этот раз результаты окажутся неудовлетворительными. Тогда опять надо поправить модель – и так далее. Разумеется, все это делается в

предположении, что сама математическая задача, возникающая для каждого варианта модели, решается достаточно точно и если возникают расхождения с ожидаемыми результатами, то только из-за недостаточно удачно выбранных гипотез, положенных в основу модели.

В действительности, и само решение математической задачи зачастую вызывает значительные трудности. Поэтому при построении модели необходимо учитывать, к какой математической задаче она сведется, – имеет ли эта задача решение, легко ли получить это решение. Заметим при этом, что решение таких задач, как правило, возможно только с помощью компьютера.

Алгоритмизация

Под *алгоритмизацией* понимают процесс разработки алгоритма решения какой-либо задачи. Переход от содержательной постановки задачи к разработке алгоритма ее решения иногда вызывает большие трудности. Разработчик алгоритма всегда должен четко представлять себе ту алгоритмическую систему, в рамках которой составляется алгоритм. Если речь идет о составлении алгоритма для компьютера, необходимо хорошо представлять возможности исполнителя алгоритма и тщательно их учитывать.

Процесс алгоритмизации по своей методике очень близок к процессу построения математических моделей. Мало того, при построении собственно формального математического описания явления и задач, которые нужно решать в связи с анализом такого явления, необходимо прямо учитывать возможность построения такого алгоритма решения, который мог бы выполнить компьютер. В этом смысле математическое моделирование и процесс алгоритмизации теснейшим и неразрывным образом связаны между собой.

Несмотря на то, что разработка каждого нового алгоритма требует своего собственного подхода, тем не менее, есть некоторые общие приемы и этапы этого рода деятельности.

Первый необходимый этап разработки алгоритма может быть охарактеризован как содержательный анализ задачи. На этом этапе следует вникнуть в суть задачи, выяснить, что дано и что требуется получить. Пожалуй, самое важное – это оценить множество исходных данных, ведь речь, как правило, идет о разработке алгоритма, обладающего свойством массовости. Первый этап алгоритмизации, состоящий в анализе задачи, отвечает нам на вопрос: может ли быть задача решена вообще и при каких исходных данных мы можем получить имеющий смысл результат.

Второй этап алгоритмизации состоит в точной постановке задачи или в построении математической модели исходной задачи.

Третий этап алгоритмизации можно назвать анализом возможностей исполнителя или более точно – анализом алгоритмической системы, в которой мы будем строить алгоритм.

Следующим этапом алгоритмизации является разработка идеи алгоритмического процесса и анализа этой идеи. И, наконец, на завершающем этапе осуществляется кодирование каждого шага алгоритмического процесса с помощью языка, понятном исполнителю алгоритма.

Типы алгоритмов

Рассмотрим классические типы алгоритмов:

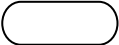


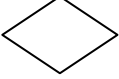


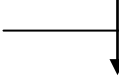

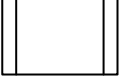
- *линейный*,
- *условный* (или *разветвляющийся*),
- *циклический*,
- *рекурсивный*,
- *эвристический*.

Для каждого типа алгоритма представим его описание как на естественном языке, так и на языке блок-схем. При этом описание алгоритма на естественном языке представим как в традиционном, так и в структурированном видах.

При описании алгоритмов на естественном языке рекомендуется придерживаться следующих принципов:

- порядковая нумерация действий алгоритма;
- структуризация номера для структурированного описания алгоритма;
- объявление константы с помощью глаголов **“определить”** и **“положить”**;
- инициализация переменной с помощью глаголов **“ввести”** и **“положить”**;
- визуализация значений выражений с помощью глагола **“вывести”**.

При описании алгоритмов на языке блок-схем рекомендуется придерживаться следующих графических символов:

	Пограничные точки – начало и конец исполнения
	Операция общего вида
	Ввод и Вывод
	Разветвление
	Линия передачи управления – сверху вниз и слева направо
	Линия передачи управления – снизу вверх и справа налево
	Схождение
	Точка перехода
	Вызов подпрограммы

Линейный алгоритм

Линейный алгоритм – это линейная (однонаправленная) последовательность действий, т.е. в линейных алгоритмах допускается только последовательная передача управления.

В качестве примера разработки линейного алгоритма рассмотрим задачу вычисления значения арифметического выражения $ax^2 + bx + c$.

Прежде, чем приступать к разработке какого-либо алгоритма, обсудим некоторые характерные особенности этапа его проектирования.

В общем случае для решения этой задачи можно разработать, например, три различных алгоритма, которые в зависимости от способа объявления исходных данных и способа представления результата вычисления по-разному определяют интерфейс, связанный с объявлением и инициализацией этих объектов:

- для первого алгоритма объекты a , b , c и x пусть все будут константами;
- для второго – только объекты a , b и c пусть будут константами, а объект x пусть будет переменной;
- для третьего – объекты a , b , c и x пусть все будут переменными.

В первом случае результат вычисления может быть значением как константы, так и переменной.

Использование такого алгоритма приведет к вычислению только одного значения арифметического выражения, чтобы получить другое значение этого выражения, необходимо осуществить переопределение значений констант.

Если результат вычисления арифметического выражения должен быть значением константы, то при разработке алгоритма кроме арифметических операций могут использоваться либо операции объявления и вывода, либо – только вывода.

Если этот результат должен быть значением переменной, то теперь к уже обязательным операциям объявления и вывода необходимо добавить еще и операцию присваивания.

Во втором и в третьем случаях результат вычисления может быть только значением переменной, при этом использование этих алгоритмов может привести к вычислению совокупности значений – либо функции $f(x)$, либо функции $f(a,b,c,x)$.

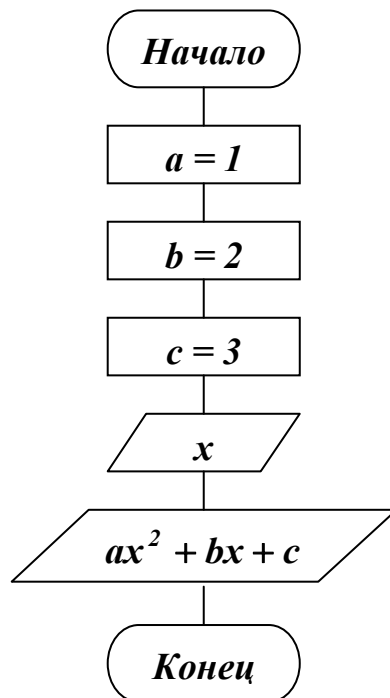
Остановим свой выбор, например, на втором алгоритме, когда объекты a , b и c являются константами, а объект x является переменной. При разработке этого алгоритма, как и в дальнейшем всех последующих, будем использовать только необходимый минимум операций. Например, здесь можно воспользоваться только операциями объявления и ввода-вывода.

Чтобы ответить на вопрос, почему даже при таком выборе операций результат вычисления может быть только значением переменной, следует напомнить, что в программной реализации этого алгоритма при выполнении операции вывода для вычисления значения арифметического выражения будет использоваться временный объект, обладающий свойствами переменной и хранящийся во вспомогательной памяти, называемой стеком времени выполнения программы.

Представим вначале традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $a = 1$*
3. *Положить $b = 2$*
4. *Положить $c = 3$*
5. *Ввести значение x*
6. *Вывести значение $ax^2 + bx + c$*
7. *Конец*

А теперь представим описание алгоритма на языке блок-схем:



Условные алгоритмы

Условные алгоритмы (алгоритмы выбора или алгоритмы ветвления) позволяют организовать выполнение альтернативных действий. В общем случае существуют две разновидности условных алгоритмов, которые отличаются друг от друга как количеством, так и механизмом исполнения альтернативных действий.

Если выполнение действий алгоритма зависит от одного из двух возможных значений некоторого логического выражения, иногда называемого условием, то возможны две ветви алгоритма – одна для значения “истина”, а другая для значения “ложь”. Каждая из двух ветвей такого алгоритма в общем случае обладает свойством независимости и в свою очередь может состоять из последовательности действий. В зависимости от значения логического выражения управление в таком условном алгоритме передается одной из двух ветвей.

Если выполнение действий алгоритма зависит от одного из множества заданных значений некоторого выражения, обладающего свойством упорядочивания конечного числа всех своих возможных значений и называемого иногда переключателем, то возможно множество ветвей алгоритма – одни для одного значения переключателя, а другие для нескольких его значений. Каждая из множества ветвей такого алгоритма является независимой и в свою очередь может состоять из последовательности действий. В зависимости от значения переключателя управление в таком условном алгоритме в общем случае передается какой-либо одной из множества ветвей. При этом может выполняться либо только одно альтернативное действие, связанное с соответствующей ему ветвью, либо несколько альтернативных действий подряд друг за другом, начиная с того, на чью соответствующую ему ветвь передается управление. Как исключение, возможен и такой случай, когда значение переключателя не позволяет передать управление ни одной из множества ветвей.

В качестве примера разработки условного алгоритма с двумя ветвями рассмотрим задачу вычисления значения модуля числа $|x| = \begin{cases} x, & \text{если } x \geq 0 \\ -x, & \text{если } x < 0 \end{cases}$

При разработке этого алгоритма рассмотрим случай, когда объект x является переменной. При этом будем придерживаться принятого соглашения о необходимом минимуме операций.

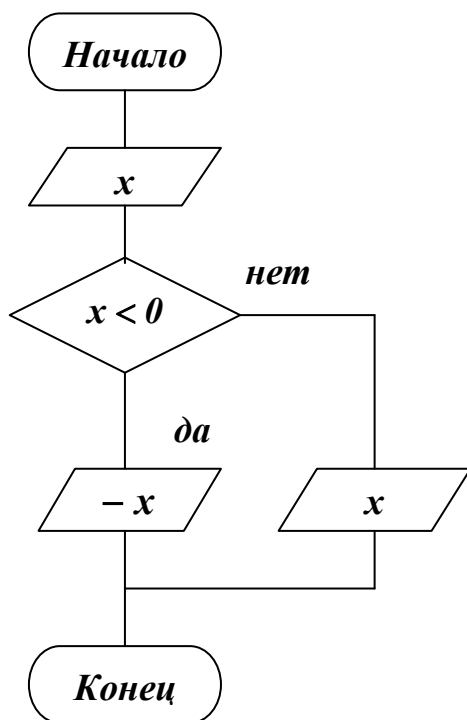
Представим вначале традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение x*
3. *Если $x < 0$, то перейти к пункту 4, иначе перейти к пункту 6*
4. *Вывести значение $-x$*
5. *Конец*
6. *Вывести значение x*
7. *Перейти к пункту 5*

Представим теперь структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Ввести значение x*
3. *Если $x < 0$*
то
 - 3.1. *Вывести значение $-x$*
 - иначе*
 - 3.2. *Вывести значение x*
4. *Конец*

А теперь представим описание алгоритма на языке блок-схем:



В качестве примера разработки условного алгоритма с множеством ветвей рассмотрим задачу вычисления значения функции $y = \begin{cases} a + x & \text{при } a = 1 \\ (a + x)^2 & \text{при } a = 2 \\ (a + x)^3 & \text{при } a = 3 \end{cases}$

При разработке этого алгоритма рассмотрим случай, когда объект a , называемый переключателем, и объект x оба являются переменными. При этом по-прежнему будем придерживаться принятого соглашения о необходимом минимуме операций.

Заметим, что как при традиционном описании алгоритмов на естественном языке, так и при описании их на языке блок-схем нет средств, чтобы представить этот алгоритм именно как условный алгоритм с множеством ветвей. Однако это становится возможным в случае структурированного описания алгоритма на естественном языке.

1. *Начало*
2. *Ввести значение x*
3. *Ввести значение a*
4. *По значению переключателя a выбрать*
Ветвь 1 :
 - 4.1. *Вывести значение $a + x$**Ветвь 2 :*
 - 4.2. *Вывести значение $(a + x)^2$**Ветвь 3 :*
 - 4.3. *Вывести значение $(a + x)^3$**иначе*
 - 4.4. *Вывести значение 'Ошибка!'*
5. *Конец*

Циклические алгоритмы

Циклические алгоритмы позволяют организовать повторное, т.е. неоднократное выполнение какой-либо последовательности действий алгоритма. Заметим при этом, что понятие повторности выполнения последовательности действий алгоритма здесь трактуется в широком смысле – от одного раза или фиксированного числа раз и многократно до тех пор, пока не будет выполнено предписанное условие. Мало того, в ряде случаев эта последовательность действий алгоритма может либо вообще не выполняться, либо выполняться бесконечно много раз.

Циклом принято называть неоднократное выполнение последовательности действий алгоритма. Последовательность действий, которая однократно выполняется в цикле, принято называть *телом цикла*. Переменную, значение которой определяет, выполнять или не выполнять тело цикла, принято называть *переменной цикла*. Однократное выполнение тела цикла принято называть *итерацией*.

В общем случае рассматривают две разновидности циклов – *цикл с условием продолжения* или так называемый цикл с предусловием и *цикл с условием завершения* или так называемый цикл с постусловием.

Цикл с условием продолжения

Первым действием тела цикла с условием продолжения является проверка условия продолжения цикла – вычисление значения логического выражения. Выполнение тела цикла осуществляется всякий раз, если значение логического выражения есть “истина”. Чтобы выполнить конечное число итераций, в теле цикла необходимо обеспечить возможность смены значения логического выражения на “ложь”, а до тела цикла – инициализацию переменной цикла. Тело цикла с условием продолжения может либо вообще не выполниться, либо выполниться требуемое число раз, либо выполняться бесконечно много раз.

В качестве примера разработки циклического алгоритма, реализованного в виде цикла с условием продолжения, рассмотрим задачу табулирования функции одного переменного, например, $y = x^2$ для пяти узлов. Здесь, как и в дальнейшем, будем придерживаться принятого соглашения о необходимом минимуме операций.

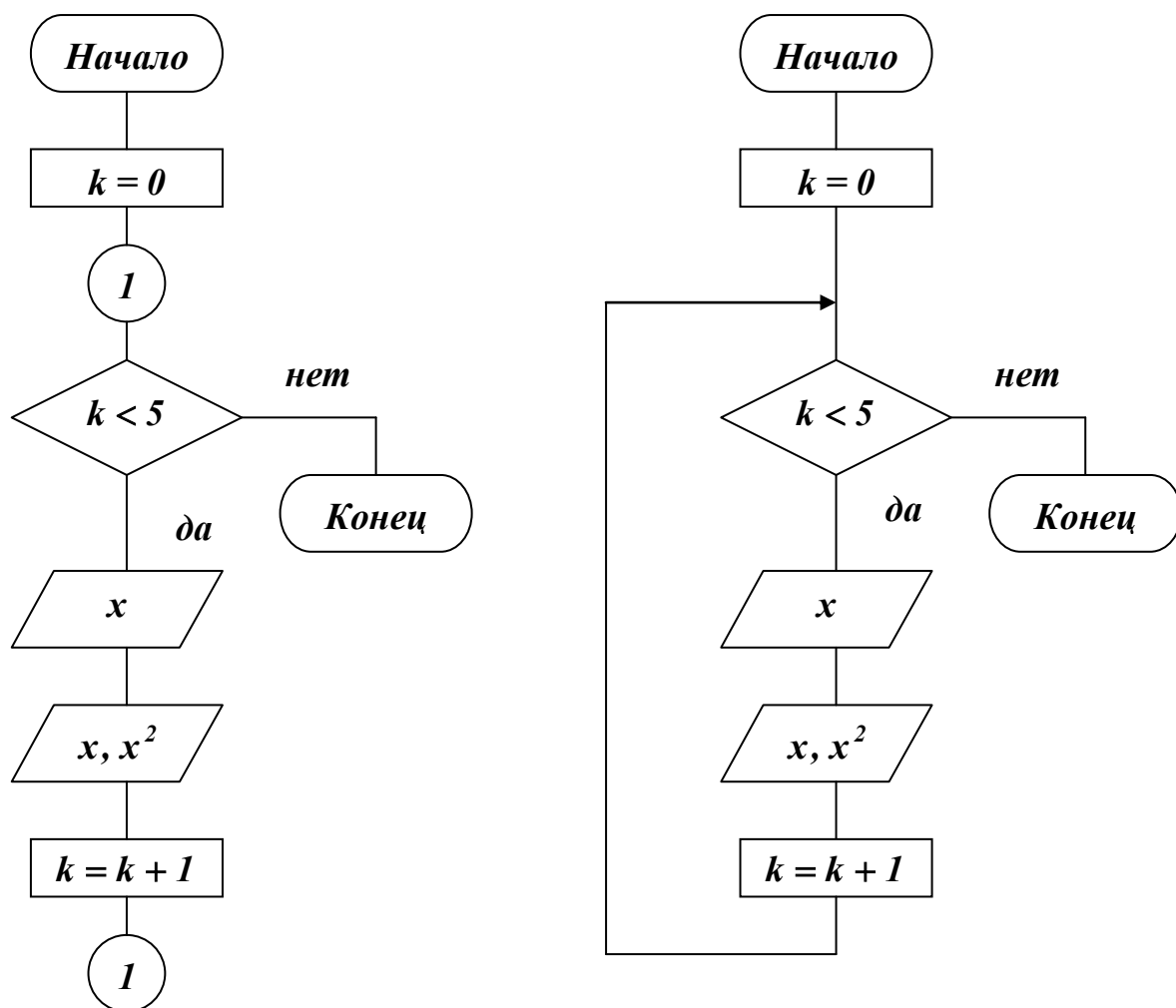
Представим вначале традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $k = 0$*
3. *Если $k < 5$, то перейти к пункту 4, иначе перейти к пункту 8*
4. *Ввести значение x*
5. *Вывести значения x и x^2*
6. *Положить $k = k + 1$*
7. *Перейти к пункту 3*
8. *Конец*

Представим теперь структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $k = 0$*
3. *Пока $k < 5$ повторить*
 - 3.1. *Ввести значение x*
 - 3.2. *Вывести значения x и x^2*
 - 3.3. *Положить $k = k + 1$*
4. *Конец*

А теперь представим два варианта описания алгоритма на языке блок-схем:



В первом варианте описания алгоритма для организации цикла используется точка перехода, помеченная, например, номером *1*, а во втором – только линии передачи управления.

Частным случаем цикла с условием продолжения является так называемый **счетный цикл**, тело которого выполняется фиксированное число раз благодаря явному или неявному изменению переменной цикла с постоянным шагом в заданном диапазоне значений либо от начального к конечному, либо от конечного к начальному. Значение шага при этом может быть задано как в явной, так и в неявной форме (так называемое стандартное значение или значение по умолчанию).

В качестве примера разработки циклического алгоритма, реализованного в виде счетного цикла, обратимся к уже известной задаче табулирования функции одного переменного $y = x^2$ для пяти узлов. Заметим, что как при традиционном описании алгоритмов на естественном языке, так и при описании их на языке блок-схем нет средств, чтобы представить этот алгоритм именно в виде счетного цикла. Однако это становится возможным в случае структурированного описания алгоритма на естественном языке.

Представим общую форму структурированного описания алгоритма на естественном языке:

1. *Начало*
2. *От $k = 1$ до $k = 5$ с шагом 1 повторить*
 - 2.1. *Ввести значение x*
 - 2.2. *Вывести значения x и x^2*
3. *Конец*

Как видим, переменная цикла здесь изменяется неявно с шагом, который явно задан единицей. Можно предложить и краткую форму описания этого алгоритма:

1. *Начало*
2. *От $k = 1$ до $k = 5$ повторить*
 - 2.1. *Ввести значение x*
 - 2.2. *Вывести значения x и x^2*
3. *Конец*

А здесь, как видим, переменная цикла изменяется неявно с шагом, который неявно задан единицей.

Теперь следует обратить внимание на одно важное обстоятельство, связанное с указанием свойств объекта, являющегося переменной цикла. С одной стороны, для реализации счетного цикла переменная цикла обязательно должна обладать свойством упорядочивания конечного числа всех своих возможных значений. Например, переменная цикла может относиться к объектам целого типа, что означает, что ее значения принадлежат диапазону целых чисел.

С другой стороны, для реализации цикла с условием продолжения, как и цикла с условием завершения, переменная цикла не обязательно должна обладать свойством упорядочивания конечного числа всех своих возможных значений. Например, переменная цикла может относиться как к объектам целого типа, так и к объектам вещественного типа, что означает, что ее значения могут принадлежать диапазону как целых, так и действительных чисел. При этом следует помнить, что выполнение вычислений в арифметике действительных чисел всегда сопряжено с ошибками округления, что в принципе может привести к потере одной итерации.

Цикл с условием завершения

Последним действием тела цикла с условием завершения является проверка условия завершения цикла – вычисление значения логического выражения. Выполнение тела цикла осуществляется всякий раз, если значение логического выражения есть “ложь”. Чтобы реализовать конечное число итераций, в теле цикла необходимо обеспечить возможность смены значения логического выражения на “истину”, а до тела цикла – инициализацию переменной цикла. Тело цикла с условием завершения может либо выполниться только один раз, либо выполниться требуемое число раз, либо выполняться бесконечно много раз.

В качестве примера разработки циклического алгоритма, реализованного в виде цикла с условием завершения, рассмотрим уже известную задачу табулирования функции одного переменного $y = x^2$ для пяти узлов.

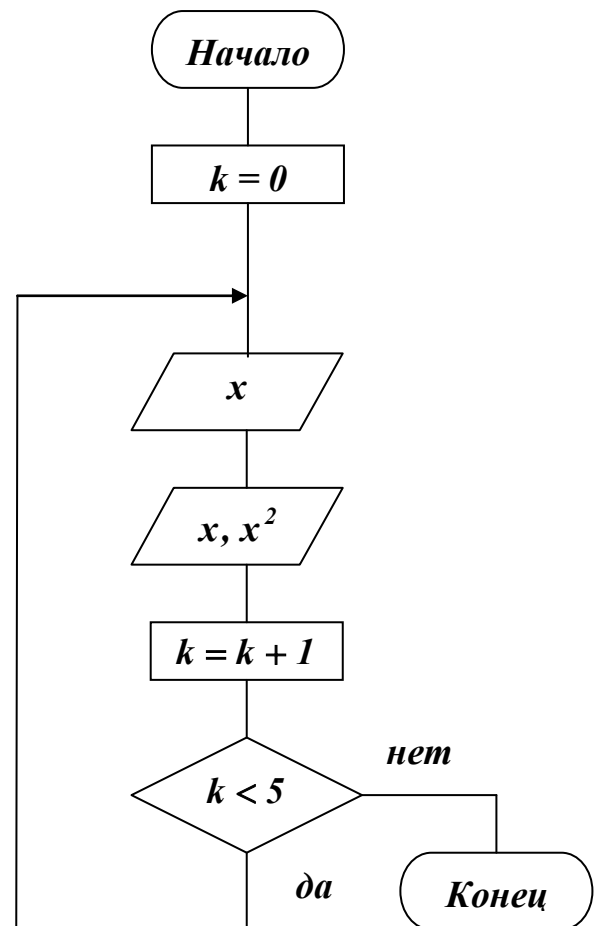
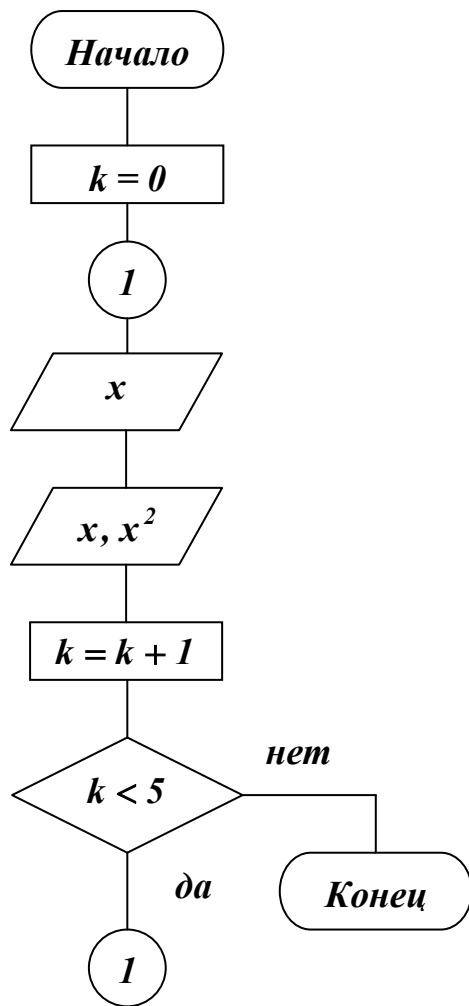
Представим вначале традиционное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $k = 0$*
3. *Ввести значение x*
4. *Вывести значения x и x^2*
5. *Положить $k = k + 1$*
6. *Если $k < 5$, то перейти к пункту 3, иначе перейти к пункту 7*
7. *Конец*

Представим теперь структурированное описание алгоритма на естественном языке:

1. *Начало*
2. *Положить $k = 0$*
3. *Повторить*
 - 3.1. *Ввести значение x*
 - 3.2. *Вывести значения x и x^2*
 - 3.3. *Положить $k = k + 1$**пока не будет $k = 5$*
4. *Конец*

А теперь представим два варианта описания алгоритма на языке блок-схем:



И в заключение обсудим весьма важную проблему, связанную с так называемым “зацикливанием” алгоритмов, реализованных в виде циклов с условием продолжения или завершения. В предложенных реализациях этих алгоритмов условием выхода из цикла является значение переменной цикла k , равное 5 . Переменная цикла изменяется от 0 до 5 с шагом, равным 1 , что позволяет выполнить требуемое число итераций, именно 5 , начиная с первой при $k = 0$ и заканчивая последней при $k = 4$. В общем случае это означает, что если требуется выполнить n итераций, то переменная цикла должна изменяться от 0 до n с шагом, равным 1 , чтобы условием выхода из цикла всегда было значение $k = n$.

Рассмотрим теперь случай, когда переменная цикла изменяется от 1 до n с шагом, равным 1 . Чтобы выполнить n итераций, условием выхода из цикла должно быть значение переменной цикла, равное $n + 1$. Однако, теперь операция изменения значения переменной цикла $k = k + 1$ становится в некотором роде спусковым механизмом “зацикливания” алгоритма, если значение $n + 1$ не будет следующим по порядку за значением n . Такое бывает возможным, например, при программировании такого алгоритма, если значение n является верхней границей некоторого выбранного диапазона целых чисел.

Если значение шага по изменению переменной цикла будет отличным от 1 , то необходимо заранее до реализации алгоритма вычислить такое допустимое верхнее значение этой переменной, чтобы оно обязательно было следующим по порядку за своим предыдущим значением. Именно это верхнее значение переменной цикла и должно проверяться в условии выхода из цикла.

Рекурсивные алгоритмы

Рекурсия – это такой способ организации алгоритма, когда он явно или неявно вызывает сам себя. Напомним, что какой-либо алгоритм может быть вызван лишь в том случае, если он организован либо как процедура, либо как функция. Способ организации алгоритма в виде процедуры или функции предполагает наличие у него заголовка, состоящего из имени, за которым в круглых скобках может следовать список параметров. Вызвать алгоритм – означает обратиться к нему по его имени с указанием, если это необходимо, списка аргументов вызова.

Явный вызов алгоритмом самого себя принято называть *прямой рекурсией*. Одним из классических примеров прямой рекурсии является функция для вычисления факториала неотрицательного целого числа:

$$n! = \begin{cases} 1 & \text{при } n = 0, \\ n \cdot (n-1)! & \text{при } n > 0. \end{cases}$$

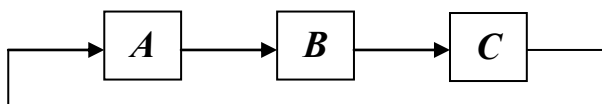
Другим ярким примером является последовательность чисел Фибоначчи, в которой каждое число, начиная с третьего, является суммой двух предыдущих:

$$F_0 = 0,$$

$$F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{при } n > 1.$$

Неявный или опосредованный вызов алгоритмом самого себя принято называть *косвенной рекурсией*. Для реализации косвенной рекурсии необходимо наличие некоторой последовательности вложенных друг в друга вызовов одними алгоритмами других, в которой какой-нибудь алгоритм последнего уровня вложения вызывает какой-либо алгоритм предыдущего уровня вложения. Глубина вложения вызовов при этом может быть произвольной. Например, рассмотрим такую последовательность вложенных вызовов, в которой пусть алгоритм *A* вызывает алгоритм *B*, вызывающий алгоритм *C*, который вновь вызывает алгоритм *A*:



Для рекурсии в любой ее форме типична необходимость отсрочки некоторых действий. Реализация механизма этих отложенных действий осуществляется при помощи вспомогательной памяти, организованной в виде динамической структуры данных и называемой стеком времени выполнения. Стек организован по принципу *LIFO* (от слов *Last In First Out*), что означает “последний пришел” – “первый ушел”. В стек записываются или как говорят “вталкиваются” (от слова *push*) аргументы вызова рекурсивных алгоритмов, которые затем извлекаются оттуда в обратном порядке или как говорят “выталкиваются” (от слова *pop*) при выполнении отложенных действий. Операции вталкивания и выталкивания выполняются путем модифицирования значения указателя вершины стека.

В дальнейшем станет известно, что при программировании рекурсивных алгоритмов глубина рекурсии во время выполнения программы будет зависеть от размера стека, а для реализации косвенной рекурсии будет необходим механизм опережающего объявления процедур и функций.

В качестве первого примера разработки рекурсивного алгоритма для случая прямой рекурсии рассмотрим функцию с одним параметром ***Factorial(n)*** для вычисления факториала неотрицательного целого числа. Напомним, что функции возвращают результат в виде значения функции посредством своего имени.

Представим вначале традиционное описание алгоритма на естественном языке:

Функция Factorial(n)

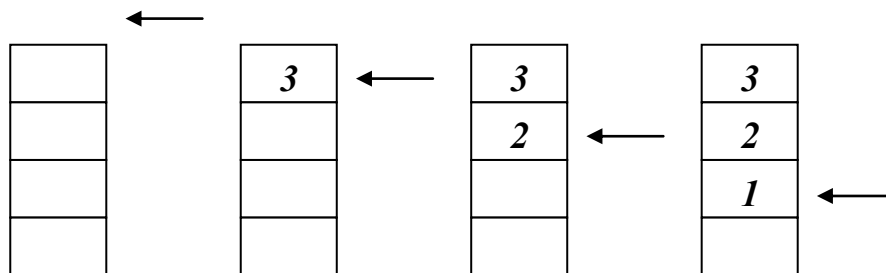
1. ***Начало***
2. ***Если $n = 0$, то перейти к пункту 3, иначе перейти к пункту 5***
3. ***Положить $Factorial = 1$***
4. ***Конец***
5. ***Положить $Factorial = n \cdot Factorial(n - 1)$***
6. ***Перейти к пункту 4***

Представим теперь структурированное описание алгоритма на естественном языке:

Функция Factorial(n)

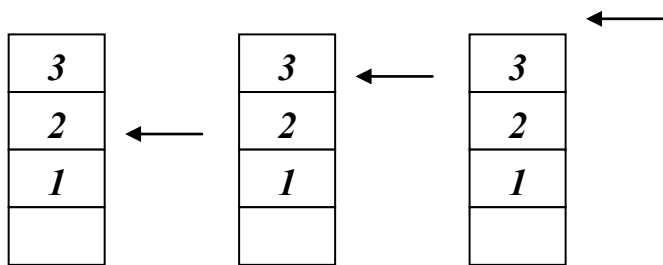
1. ***Начало***
2. ***Если $n = 0$***
то
 - 2.1. ***Положить $Factorial = 1$******иначе***
 - 2.2. ***Положить $Factorial = n \cdot Factorial(n - 1)$***
3. ***Конец***

При вычислении, например, значения $3!$ три раза будет откладываться операция умножения, в результате чего в стек будут вталкиваться друг за другом три аргумента – в первый раз значение 3 , во второй раз значение 2 , а в третий раз значение 1 :



Операция вталкивания представляет собой изменение значения указателя вершины стека и последующую запись элемента данных в стек. Здесь стрелкой показаны позиции указателя вершины стека при вталкивании в стек аргументов вызова функции ***Factorial(n)***.

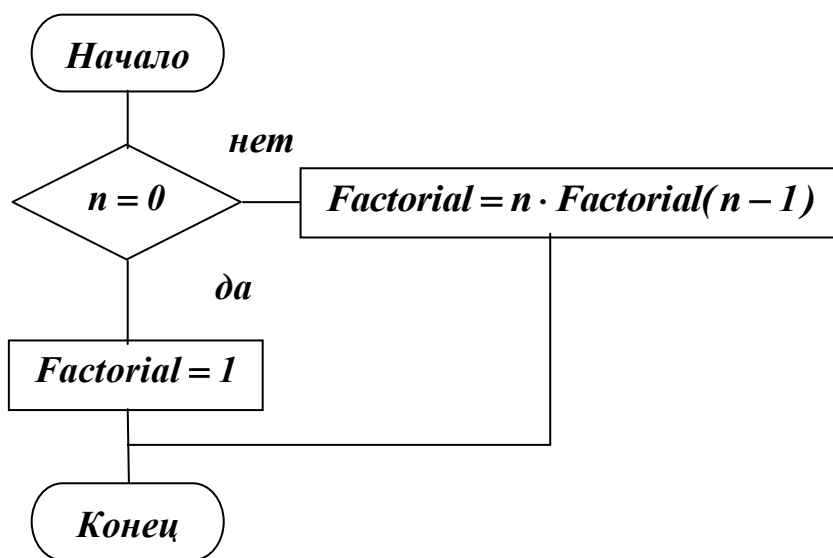
Затем при $n = 0$ выполнится операция присваивания ***Factorial = 1***, после которой друг за другом будут выполняться отложенные операции умножения, первым операндом которых будут вытолкнутые из стека значения аргументов – в первый раз ***Factorial = 1 · 1 = 1***, во второй раз ***Factorial = 2 · 1 = 2***, а в третий раз ***Factorial = 3 · 2 = 6***:



Заметим, что выталкивание данных из стека не означает, что они из него исчезают. Операция выталкивания представляет собой чтение элемента данных из стека и последующее изменение значения указателя вершины стека. Здесь стрелкой показаны позиции указателя вершины стека после выталкивания очередного элемента данных.

А теперь представим описание алгоритма на языке блок-схем:

Функция ***Factorial(n)***



В качестве второго примера разработки рекурсивного алгоритма для случая прямой рекурсии рассмотрим функцию с одним параметром ***Fibonacci(n)*** для вычисления члена последовательности чисел Фибоначчи.

Представим только структурированное описание этого алгоритма на естественном языке:

Функция *Fibonacci*(*n*)

1. *Начало*
2. *Если* ($n = 0$) *или* ($n = 1$)
то
 - 2.1. *Положить* $Fibonacci = n$*иначе*
 - 2.2. *Положить* $Fibonacci = Fibonacci(n - 1) + Fibonacci(n - 2)$
3. *Конец*

В завершение разговора о последовательности чисел Фибоначчи уместным будет здесь напомнить, что всякое положительное число m можно единственным образом представить в виде суммы чисел Фибоначчи, причем в этом разложении наибольшее число не будет превосходить m и никакие два числа не будут соседними членами последовательности чисел Фибоначчи.

И в заключение обсудим одну важную проблему, связанную с эффективностью программных реализаций рассматриваемых рекурсивных алгоритмов. Как видим, рекурсивные алгоритмы ясны и прозрачны для понимания, изящны и совершенны как строгие математические формулы. Однако у рекурсивных алгоритмов есть один существенный недостаток – это большие затраты времени при их исполнении, связанные с издержками по управлению стеком при выполнении отложенных действий. Традиционное решение этой проблемы – это отказ от рекурсии и переход к итеративным реализациям алгоритмов.

В качестве примера такого перехода от рекурсивных алгоритмов к циклическим представим структурированные описания этих алгоритмов на естественном языке, а также обсудим их достоинства и недостатки:

Функция *Factorial*(*n*)

1. *Начало*
2. *Если* $n = 0$
то
 - 2.1. *Положить* $Factorial = 1$*иначе*
 - 2.2. *Положить* $p = 1$
 - 2.3. *От* $k = 1$ *до* $k = n$ *повторить*
 - 2.3.1. *Положить* $p = p \cdot k$
 - 2.4. *Положить* $Factorial = p$
3. *Конец*

Функция *Fibonacci*(*n*)

- 1. Начало**
- 2. Если ($n = 0$) или ($n = 1$)**
то
 - 2.1. Положить $Fibonacci = n$****иначе**
 - 2.2. Положить $a = 0$**
 - 2.3. Положить $b = 1$**
 - 2.4. От $k = 2$ до $k = n$ повторить**
 - 2.4.1. Положить $c = a + b$**
 - 2.4.2. Положить $a = b$**
 - 2.4.3. Положить $b = c$**
 - 2.5. Положить $Fibonacci = c$**
- 3. Конец**

Как видим, циклические алгоритмы в отличие от рекурсивных не так ясны и прозрачны для понимания, кроме того, они требуют дополнительных переменных. Зато программные реализации циклических алгоритмов весьма эффективны по времени выполнения. Если у задачи есть две схемы решения – итерационная и рекурсивная, – то предпочтение, как правило, отдается итеративным алгоритмам.

С другой стороны, есть задачи, для решения которых применяются только рекурсивные схемы, так как построение итерационной схемы может оказаться либо слишком сложным, либо не естественным.

Эвристические алгоритмы

Существует определенный класс задач, решение которых возможно лишь на основе применения специальных методов, построенных на проверке всяких догадок, удачных идей или любых разумных способов поиска требуемых величин.

Такие процессы поиска решения задачи называются *эвристическими*, а сами методы – *эвристиками*. Исторически понятие эвристики возникло в Древней Греции как метод обучения, применявшийся Сократом: в ходе беседы ее участники с помощью системы наводящих вопросов и приемов наталкивались учителем на правильный ответ. В современном понимании одно из значений понятия эвристики представляет собой метод или алгоритм выработки правильного решения задачи с помощью неформализованных или самообучающихся методик.

В качестве примера разработки эвристических алгоритмов рассмотрим несколько задач, решение которых основано на применении различных эвристик.

Многим известна задача о перевозчике, который должен перевезти с одного берега реки на другой козу, капусту и волка при условии, что за один раз в лодке можно перевезти либо козу, либо капусту, либо волка. В результате анализа исходных данных эвристика здесь может быть построена на предположении, что на берегу реки нельзя оставлять вместе как козу с капустой, так и волка с козой.

Представим традиционное описание двух алгоритмов на естественном языке:

1. *Начало*
2. *Перевезти козу*
3. *Вернуться обратно*
4. *Перевезти волка*
5. *Вернуться обратно с козой*
6. *Перевезти капусту*
7. *Вернуться обратно*
8. *Перевезти козу*
9. *Конец*

1. *Начало*
2. *Перевезти козу*
3. *Вернуться обратно*
4. *Перевезти капусту*
5. *Вернуться обратно с козой*
6. *Перевезти волка*
7. *Вернуться обратно*
8. *Перевезти козу*
9. *Конец*

Рассмотрим еще одну известную задачу, в которой требуется поделить поровну воду из полного ведра емкостью **8 л** с помощью двух пустых ведер соответственно емкостью **5 л** и **3 л** за минимальное количество переливаний. Существует и другая формулировка задачи, где используются, например, ведра емкостью **10 л**, **7 л** и **4 л**.

В результате анализа исходных данных эвристика здесь может быть построена на предположении, что ведро максимальной емкости должно служить своеобразным накопителем остатков воды из других ведер. Также следует предположить, что перед последней операцией переливания в одном из первых двух ведер должен остаться **1 л** воды, а ведро минимальной емкости при этом должно быть полным.

Представим описание двух алгоритмов переливания воды для ведер, которые назовем, например, *A*, *B* и *C*, где с помощью стрелки будем указывать, из какого ведра в какое переливается вода, а после переливания будем отмечать состояние всех ведер, т.е. количество литров воды в каждом ведре:

		<i>A</i>	<i>B</i>	<i>C</i>			<i>A</i>	<i>B</i>	<i>C</i>
		8	0	0			10	0	0
1.	$A \rightarrow B$	3	5	0	1.	$A \rightarrow C$	6	0	4
2.	$B \rightarrow C$	3	2	3	2.	$C \rightarrow B$	6	4	0
3.	$C \rightarrow A$	6	2	0	3.	$A \rightarrow C$	2	4	4
4.	$B \rightarrow C$	6	0	2	4.	$C \rightarrow B$	2	7	1
5.	$A \rightarrow B$	1	5	2	5.	$B \rightarrow A$	9	0	1
6.	$B \rightarrow C$	1	4	3	6.	$C \rightarrow B$	9	1	0
7.	$C \rightarrow A$	4	4	0	7.	$A \rightarrow C$	5	1	4
					8.	$C \rightarrow B$	5	5	0

Заметим, что здесь каждый алгоритм представлен только одной своей оптимальной версией, конечно же, существуют и другие версии как оптимальные, так и не оптимальные по количеству переливаний.

Рассмотрим теперь задачу о программировании автомата с конечным набором операций и известным начальным состоянием, который позволяет за минимальное количество операций перейти от исходного числа, например, равного нулю, к заданному натуральному числу при помощи, например, двух операций – прибавить единицу и умножить на два.

При построении эвристики здесь необходимо воспользоваться результатом процесса перехода от заданного натурального числа к исходному при помощи набора обратных операций – вычесть единицу и разделить на два. Операция вычитания единицы позволит переходить от нечетных к четным числам, которые затем можно будет делить на два. Зная последовательность выполнения обратных операций и соответственно их тип, легко получить решение этой задачи.

Представим словесно-формульное описание алгоритма для обратного и прямого хода решения, например, для числа **21**:

1.	<i>Начало</i>		1.	<i>Начало</i>	
2.	<i>Вычесть 1</i>	$21 - 1 = 20$	2.	<i>Прибавить 1</i>	$0 + 1 = 1$
3.	<i>Разделить на 2</i>	$20 : 2 = 10$	3.	<i>Умножить на 2</i>	$1 \cdot 2 = 2$
4.	<i>Разделить на 2</i>	$10 : 2 = 5$	4.	<i>Умножить на 2</i>	$2 \cdot 2 = 4$
5.	<i>Вычесть 1</i>	$5 - 1 = 4$	5.	<i>Прибавить 1</i>	$4 + 1 = 5$
6.	<i>Разделить на 2</i>	$4 : 2 = 2$	6.	<i>Умножить на 2</i>	$5 \cdot 2 = 10$
7.	<i>Разделить на 2</i>	$2 : 2 = 1$	7.	<i>Умножить на 2</i>	$10 \cdot 2 = 20$

8. *Вычесть 1* $1 - 1 = 0$

9. *Конец*

8. *Прибавить 1* $20 + 1 = 21$

9. *Конец*

Обратный ход решения

Прямой ход решения

Как видим, здесь неявно подразумевается, что автомат “знает”, какое число является четным, а какое – нечетным. Чтобы от частного решения задачи перейти к общему, можно воспользоваться, например, стеком времени выполнения, в который во время обратного хода будут вталкиваться мнемонические обозначения прямых операций, а во время прямого хода, выталкивая их из стека, можно формировать строки решения для каждой операции.

Представим структурированное описание алгоритма на естественном языке для обратного и прямого хода, например, для заданного натурального числа n :

1. *Начало*

2. *Пока $n > 0$ повторить*

2.1. *Если n нечетное
то*

2.1.1. *Положить $n = n - 1$*

2.1.2. *Втолкнуть в стек символ +
иначе*

2.1.3. *Положить $n = \frac{n}{2}$*

2.1.4. *Втолкнуть в стек символ **

3. *Положить $r = 0$*

4. *Положить $r = 0$*

5. *Пока стек не пустой повторить*

5.1. *Вытолкнуть из стека оператор*

5.2. *Если оператор +
то*

5.2.1. *Положить $r = r + 1$*

5.2.2. *Вывести строку для слагаемых r и 1 и суммы r*

5.2.3. *Положить $p = r$*

иначе

5.2.4. *Положить $r = r \cdot 2$*

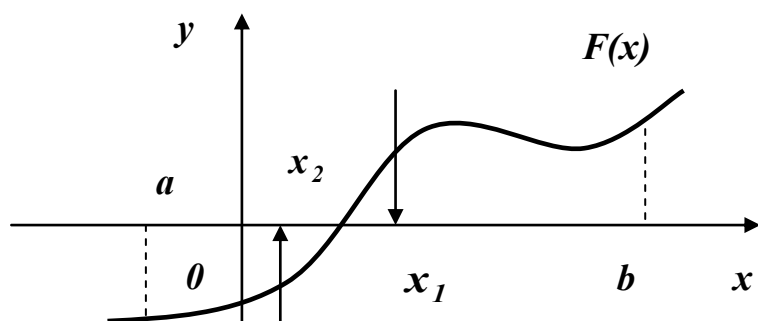
5.2.5. *Вывести строку для множителей p и 2 и произведения r*

5.2.6. *Положить $p = r$*

6. *Конец*

Здесь символы $+$ и $*$ являются общепринятыми обозначениями соответственно операторов сложения и умножения.

В заключение рассмотрим задачу отыскания действительных корней нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$:



$$x_1 = \frac{a+b}{2}$$

$$x_2 = \frac{a+x_1}{2}$$

Рассмотрим эвристику, которая лежит в основе метода дихотомии или так называемого метода деления отрезка пополам. Если на границах некоторого отрезка (на рисунке отрезок $[a, b]$) функция $F(x)$ меняет знак, то существует, по крайней мере, одна точка на этом отрезке, в которой функция $F(x)$ обращается в нуль. Если разделить этот отрезок пополам и определить знак функции $F(x)$ в точке деления, то тем самым можно определить другой отрезок, на границах которого функция $F(x)$ меняет знак (на рисунке отрезок $[a, x_1]$). В принципе, повторное применение этого метода, т.е. деление отрезка локализации корня пополам, может позволить сколь угодно близко подойти к корню. Например, на рисунке показана вторая итерация отыскания корня, которая позволяет перейти от отрезка $[a, x_1]$ к отрезку $[x_2, x_1]$.

“В принципе”, потому что существуют погрешности вычислений, т.е. определение знака функции $F(x)$ для какой-либо очередной итерации может быть неверным из-за ошибки округления.

Рассмотрим функцию с тремя параметрами $Dichotomy(a, b, Tolerance)$ для отыскания действительного корня нелинейного уравнения $F(x) = 0$ методом дихотомии на отрезке локализации корня $x \in [a, b]$ с заданной точностью $Tolerance$. Для реализации этой функции можно выполнить разработку как циклического, так и рекурсивного алгоритмов.

Представим только структурированное описание циклического алгоритма на естественном языке:

Функция $Dichotomy(a, b, Tolerance)$

1. **Начало**
2. **Положить $x_left = a$**
3. **Положить $f_left = F(x_left)$**
4. **Положить $x_right = b$**
5. **Положить $f_right = F(x_right)$**

6. Повторить

6.1. Положить $x_root = \frac{x_left + x_right}{2}$

6.2. Положить $f_root = F(x_root)$

6.3. Если $f_left \cdot f_root > 0$

то

6.3.1. Положить $x_left = x_root$

6.3.2. Положить $f_left = f_root$

иначе

6.3.3. Положить $x_right = x_root$

пока не будет $|f_root| < Tolerance$

7. Положить $Dichotomy = x_root$

8. Конец

Итерационная процедура деления отрезка локализации корня пополам завершается благодаря выполнению неравенства $|F(x_root)| < Tolerance$, где x_root является найденным значением действительного корня для заданного значения “нуля” $Tolerance$