Профессор
Игорь Н. Бекман

КОМПЬЮТЕРНЫЕ НАУКИ

Курс лекций

Лекция 7. АЛГОРИТМЫ

Содержание

Теория алгоритмов – одно из основных понятий математики и информатики. Даже происхождение самого термина «алгоритм» связано с математикой. Известно, что основная особенность всех вычислений машины состоит в том, что в основе её работы лежит программный принцип управления. Это означает, что для решения как самой простой, так и самой сложной задачи пользователю необходимо использовать перечень инструкций или команд, следуя которым шаг за шагом компьютер выдаст необходимый результат. Таким образом, для того, чтобы решать задачу на компьютере, её необходимо сначала алгоритмизировать. Именно алгоритмический принцип и лежит в основе работы всех компьютеров.

В данной лекции дадим определение термина алгоритм, рассмотрим историю его развития, виды алгоритмов и алгоритмические языки. Далее мы более подробно остановимся на теории алгоритмов и закончим применением алгоритмов в компьютере.

1. АЛГОРИТМ

1.1 Определение понятий

В старой трактовке **алгори́тм** - точный набор инструкций, описывающих последовательность действий исполнителя для достижения результата решения задачи за конечное время. По мере развития параллельности в работе компьютеров слово «последовательность» стали заменять более общим словом «порядок». Это связано с тем, что какие-то действия алгоритма должны быть выполнены только друг за другом, но какие-то могут быть и независимыми. Часто в качестве исполнителя выступает некоторый механизм (компьютер, токарный станок, швейная машина), но понятие алгоритма необязательно относится к компьютерным программам, так, например, чётко описанный рецепт приготовления блюда также является алгоритмом, в таком случае исполнителем является человек.

Единого «истинного» определения понятия «алгоритм» нет.

Алгоритм - конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определённость, ввод, вывод, эффективность.

Алгоритм - всякая система вычислений, выполняемых по строго определённым правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Алгоритм - строго детерминированная последовательность действий, описывающая процесс преобразования объекта из начального состояния в конечное, записанная с помощью понятных исполнителю команд.

Алгоритм - последовательность действий, направленных на получение определённого результата за конечное число шагов».
Алгоритм - последовательность действий, либо приводящая к решению задачи, либо поясняющая почему это решение получить нельзя.
Алгоритм - это точная, однозначная, конечная последовательность действий, которую должен выполнить пользователь для достижения конкретной цели либо для решения конкретной задачи или группы задач за конечное число шагов.

Общее в этих определениях то, что алгоритм - это ***предписание***. Предписание должно быть задано (закодировано) в некоторой форме. Это может быть текст - строка символов в некотором алфавите, таблица, диаграмма, упорядоченный набор пиктограмм и т.д.

Любой алгоритм существует не сам по себе, а предназначен для определённого исполнителя. Алгоритм описывается в командах исполнителя, который это алгоритм будет выполнять. Объекты, над которыми исполнитель может совершать действия, образуют среду исполнителя. Исходные данные и результаты любого алгоритма всегда принадлежат среде того исполнителя, для которого предназначен алгоритм.

Значение слова «алгоритм»очень схоже со значением слов «рецепт», «метод», способ. Однако любой алгоритм, в отличие от рецепта или способа, обязательно обладает следующими свойствами:
Дискретность - алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, т. е. преобразование исходных данных в результат осуществляется во времени дискретно. Можно считать, что шаги выполняются мгновенно в моменты времени t0, t1, t2…, а между этими моментами ничего не происходит.
Элементарность шагов означает, что объем работы, выполняемой на любом шаге, мажорируется некоторой константой, зависящей от характеристик исполнителя алгоритмов, но не зависящей от входных данных и промежуточных значений, получаемых алгоритмом. Для численных алгоритмов такими элементарными шагами могут быть, например, сложение, вычитание, умножение, деление, сравнение двух 32-разрядных чисел, пересылка одного числа из некоторого места памяти в другое. К элементарным шагам не относится сравнение двух файлов, так как время сравнения зависит от длины файлов, а длина потенциально неограниченна
Детерминированность - определённость. В каждый момент времени следующий шаг работы однозначно определяется состоянием системы: алгоритм выдаёт один и тот же результат для одних и тех же исходных данных. Результаты не зависят ни от каких случайных факторов. С другой стороны, существуют вероятностные алгоритмы, в которых следующий шаг работы зависит от текущего состояния системы и генерируемого случайного числа. Однако при включении метода генерации случайных чисел в список «исходных данных», вероятностный алгоритм становится подвидом обычного.
Понятность - алгоритм для исполнителя должен включать только те команды, которые ему (исполнителю) доступны, которые входят в его систему команд.
Завершаемость (конечность, определённость) - при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов. С другой стороны, вероятностный алгоритм может и никогда не выдать результат, но вероятность этого равна 0.
Конечность (финитность*)* алгоритма означает, что для получения результата нужно выполнить конечное число шагов, т. е. исполнитель в некоторый момент времени останавливается. Требуемое число шагов зависит от входных данных алгоритма и не мажорируется константой.
Массовость - алгоритм должен быть применим к разным наборам исходных данных.
Результативность - завершение алгоритма определенными результатами. Если же входные данные уникальны, то алгоритм в силу свойства определенности (детерминированности) будет давать всегда один и тот же результат и само построение алгоритма теряет смысл.
Алгоритм содержит ошибки, если приводит к получению неправильных результатов либо не дает результатов вовсе.
Алгоритм не содержит ошибок, если он дает правильные результаты для любых допустимых исходных данных.
Алгоритм всегда рассчитан на выполнение «неразмышляющим» исполнителем.
Понятие ***данных (значений)*** - исходных, промежуточных и результата также требует некоторого ограничительного толкования. Наиболее общее интуитивное понимание состоит в том, что данными в алгоритме могут служить самые разнообразные конструктивные объекты.

Конструктивный объект - это строгое математическое понятие. Можно пока считать, что конструктивный объект - это элемент какого-либо конечного множества (например, один из дней недели), либо объект, вычисленный каким-либо алгоритмом. Конструктивными объектами являются символы, логические значения, целые и вещественные числа, представимые в машине, массивы конструктивных объектов. Алгоритм (его текст) также является конструктивным объектом и, значит, может рассматриваться как данные для другого алгоритма: текст программы (алгоритма) является входными данными для программы-транслятора.

Таким образом, понятия алгоритма и данных двойственны, их определения рекурсивны: в формулировке понятия алгоритма использовались понятия данных, а они, в свою очередь, определяются с использованием понятия алгоритма, и т. д. Это определяет равную важность двух понятий в компьютерных науках, что отражается и в современных языках программирования.

Замечание об определенности и конечности: иногда считают, что алгоритм может заканчиваться без получения результата (безрезультатная остановка) или даже не заканчиваться вовсе при некоторых исходных данных (неприменимость к этим исходным данным). Взгляд на это с точки зрения теории - машины Тьюринга - обсудим позже. С практической точки зрения такая ситуация тоже требует внимания: операционная система (ОС) вычислительной машины, являясь совокупностью алгоритмов, при нормальной работе не предполагает остановки и выдачи каких-либо результатов; она лишь добросовестно получает периодически входные данные-задания и запускает их; задания-алгоритмы сами получают результаты. Таким образом, ОС не выдаёт продукции, если не считать протокола её работы. Другие диалоговые программные системы также требуют для своего описания более широкой интерпретации понятия алгоритма: они не получают входные данные сразу и не всегда можно говорить об априорной ограниченности объёма данных некоторой константой. Однако все сказанное не умаляет важности приведённого понятия алгоритма, а говорит лишь о богатстве проблематики компьютерных наук.

Дадим уточняющее понятие алгоритма, которое не является определением в математическом смысле слова, но более формально описывает понятие алгоритма, раскрывающего его сущность.

Алгоритм – конечная система правил, сформулированная на языке исполнителя, которая определяет последовательность перехода от допустимых исходных данных которая определяет последовательность перехода от допустимых исходных данных к конечному результату и которая обладает свойствами дискретности, детерминированности, результативности, конечности и массовости.

Для каждого исполнителя набор допустимых действий всегда ограничен – не может существовать исполнителя, для которого любое действие является допустимым. Перефразируя И.Канта: «Если бы такой исполнитель существовал, то среди его допустимых действий было бы создание такого камня, который он не может поднят. Но это противоречит допустимости действия «поднять любой камень».

Ограничение на выбор допустимых действий означает, что для любого исполнителя имеются задачи, которые нельзя решить с его помощью. При изучении алгоритмов важно разделять два понятия: запись алгоритма и выполнение алгоритма.

Алгоритм является предписанием, а наличие предписания предполагает, что результат будет получен неким исполнителем, действующим по этому предписанию. Исполнитель (компьютер или программист, вручную отлаживающий свою программу) получает предписание и исходные данные. После этого он начинает действовать как автомат, т.е. выполнять в реальном времени описанные в алгоритме шаги. В результате выполнения каждого шага могут образовываться промежуточные результаты, которые исполнитель должен где-то фиксировать так, чтобы они могли быть использованы в качестве исходных данных для следующего шага. Исполнитель совершит конечное число шагов (даже если отдельные описания шагов использовались неоднократно) и после этого остановится, зафиксировав окончательный результат подобно промежуточным результатам.

Если есть текст некоторого предписания, то нужно убедиться в том, что это предписание является алгоритмом. Для этого необходимо проверить, выполняются ли перечисленные выше свойства.

Пример 1. Проверка на примере текста (отыскание максимального и минимального элементов массива):

«Исходные данные - положительное число *N*, определяющее количество элементов массива *A*, и целочисленные элементы *A[1], A[2], ..., A[N]* массива *A*. Значения всех чисел находятся в пределах непосредственно представимых в вычислительной машине. Кроме исходных данных вводятся целочисленные переменные *Max, Min, i.* Первые две по окончании работы алгоритма определяют его результаты, третья является вспомогательной. Действия алгоритма состоят в выполнении следующих шагов:

1. Установить значения $Max = A[1]$, $Min = A[1]$, $i = 2$.
2. Пока $i <= N$ повторять шаги с 3 по 5.
3. Если $Max < A[i]$, то положить $Max = A[i]$.
4. Если $Min > A[i]$, то положить $Min = A[i]$.

5. Увеличить *i* на 1.
6. Вывести результаты *Max* и *Min*.
7. Остановиться».

Проверим выполнение основных свойств.

Дискретность очевидна.

Элементарность шагов. Шаг 1 содержит три присваивания значений; шаг 2 содержит одно сравнение чисел; шаги 3-5 содержат два сравнения чисел, два присваивания значений (выполняется каждый раз только одно из них), одно увеличение значения на единицу; шаг 6 - вывод на экран или на печать данных ограниченного объема.

Определенность. Каждый шаг и алгоритм в целом заканчивается определенным результатом; строго определена последовательность шагов.

Конечность. Шаги 1 и 6 выполняются по одному разу. Количество выполнений шагов 2-5 зависит от значений переменной *i* и уровня *N*. Поскольку *i* монотонно возрастает, то ее значение достигнет уровня *N* через конечное число шагов. Если начальное значение переменной *i* больше уровня *N*, то шаг 2 выполняется один раз, а шаги 3-5 не выполняется ни разу. Таким образом, в любом случае выполнение алгоритма завершится через конечное число шагов.

Массовость. Алгоритм может воспринимать в качестве исходных данных различные массивы разной длины.

Однако не всегда так легко доказать выполнимость основных свойств алгоритма. Особенно это касается свойства конечности. Рассмотрим, например, алгоритм с одним входным аргументом - натуральным числом *k*.

1. Если $k = 1$, то остановиться.
2. Если k четное, то положить $k = k/2$; если *k* нечетное, то положить $k = 3k+1$.
3. Повторить, используя новое значение *k*.

Как следует из текста алгоритма, имеется некоторый процесс изменения значения k, начинающийся с определенного начального значения, затем на каждом шаге *k* либо увеличивается, либо уменьшается и, наконец, возможно, приходит к значению 1, на котором и останавливается. В общем случае процесс немонотонный. Например, для $k = 40$: $k = 40, 20, 10, 5, 16, 8, 4, 2, 1$ (8 шагов). Решить вопрос о конечности данного алгоритма - это значит доказать одно из двух утверждений: для любого *k* процесс заканчивается единицей; для некоторого *k* процесс не заканчивается.

Если *k* равно степени двойки (2, 4, 8, 16, 32, ...), то процесс будет монотонно убывающим и завершится за число шагов, равное этой степени. В противном случае на некотором промежуточном шаге значение *k* станет нечётным, но не равным единице, и на следующем шаге значение *k* увеличится. Оба варианта (алгоритм заканчивается, алгоритм не заканчивается) не кажутся очевидными. С одной стороны, увеличение *k* происходит в три раза, а уменьшение только в два раза и, если шаги увеличения и уменьшения строго чередуются, то для такого процесса имеется общая тенденция к возрастанию. С другой стороны, за шагом увеличения обязательно следует шаг уменьшения, но обратное неверно, т. е. шагов уменьшения может быть и больше, чем шагов увеличения. Потенциально возможно также зацикливание процесса, когда на очередном шаге получается уже встречавшееся ранее значение. Вот типичный пример с чередованием шагов: $k = 127, 382, 191, 574, 287, 862, 431, 1294, 647, 1942, 971, 2914, 1457, 4372, 2186, 1093, 3280, 1640, 820, 410, 205, 616, 308, 154, 77, 232, 116, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

До значения $k = 4372$ шаги строго чередуются, затем начинается отрезок нерегулярного изменения, на котором имеется 20 четных и 8 нечетных чисел, и заканчивается процесс «хвостом» из степеней двойки. Можно показать, что для всех нечетных $k = 2j - 1$ или даже для $k = n2j - 1$, где *n* - нечётное натуральное число, начальный участок последовательности является строго чередующимся до достижения величины $2(n3j - 1)$; причем длина участка строгого чередования шагов пропорциональна величине *j*. Это плохая тенденция, поскольку *k* удаляется от 1. С другой стороны, для многих сочетаний *n* и *j* величина $n3j - 1 = m2p$ - пропорциональна степени двойки. В этом случае вслед за возрастанием *k* идёт группа операций деления на 2, «сбрасывающая» значение *k* до величины *m*. В целом вопрос о конечности этого алгоритма должен решаться методами теории чисел. Несмотря на ряд усилий, предпринимавшихся математиками, решение пока не найдено.

1.2 История термина

Современное формальное определение алгоритма было дано в 30-50-х годы XX века в работах Тьюринга, Поста, Чёрча (тезис Чёрча — Тьюринга), Н. Винера, А.А.Маркова.

Само слово «алгоритм» происходит от имени учёного Абу Абдуллах Мухаммеда ибн Муса аль-Хорезми. В 825 он написал сочинение, в котором впервые дал описание придуманной в Индии позиционной десятичной системы счисления. Аль-Хорезми сформулировал правила вычислений в новой

системе и, вероятно, впервые использовал цифру 0 для обозначения пропущенной позиции в записи числа (её индийское название арабы перевели как *as-sifr* или просто *sifr*, отсюда такие слова, как «цифра» и «шифр»). Приблизительно в это же время индийские цифры начали применять и другие арабские учёные. В первой половине XII века книга аль-Хорезми в латинском переводе проникла в Европу. Переводчик дал ей название *Algoritmi de numero Indorum* («Алгоритми о счёте индийском»). По-арабски же книга именовалась *Китаб аль-джебр валь-мукабала* («Книга о сложении и вычитании»). Из оригинального названия книги происходит слово Алгебра.

В средние века слово *algorism* (или *algorismus*), неизменно присутствовавшее в названиях математических сочинений, обрело значение способа выполнения арифметических действий посредством арабских цифр, то есть на бумаге, без использования абака. Именно в таком значении оно вошло во многие европейские языки.

Алгоритм - это искусство счёта с помощью цифр, но поначалу слово «цифра» относилось только к нулю. Знаменитый французский трувер Готье де Куанси (Gautier de Coincy, 1177-1236) в одном из стихотворений использовал слова *algorismus-cipher* (которые означали цифру 0) как метафору для характеристики абсолютно никчёмного человека. Очевидно, понимание такого образа требовало соответствующей подготовки слушателей, а это означает, что новая система счисления уже была им достаточно хорошо известна.

Многие века абак был фактически единственным средством для практичных вычислений, им пользовались и купцы, и менялы, и учёные. Достоинства вычислений на счётной доске разъяснял в своих сочинениях такой выдающийся мыслитель, как Герберт Аврилакский (938—1003), ставший в 999 папой римским под именем Сильвестра II. Новое с огромным трудом пробивало себе дорогу, и в историю математики вошло упорное противостояние лагерей абацистов и алгорисмиков, которые пропагандировали использование для вычислений абака вместо арабских цифр. Прошло не одно столетие, прежде чем новый способ счёта окончательно утвердился, столько времени потребовалось, чтобы выработать общепризнанные обозначения, усовершенствовать и приспособить к записи на бумаге методы вычислений. В Западной Европе учителей арифметики вплоть до XVII века продолжали называть «магистрами абака», как, например, математика Никколо Тарталью (1500—1557).

Итак, сочинения по искусству счёта назывались *Алгоритмами*. Из многих сотен можно выделить и такие необычные, как написанный в стихах трактат «*Carmen de Algorismo*» (латинское *carmen* и означает стихи) Александра де Вилла Деи, ум. 1240) или учебник венского астронома и математика Георга Пурбаха (1423-1461) «*Opus algorismi jocundissimi*» («Веселейшее сочинение по алгоритму»). Постепенно значение слова расширялось. Учёные начинали применять его не только к сугубо вычислительным, но и к другим математическим процедурам. Например, в 1360 французский философ Николай Орем (1323/25-1382) написал математический трактат «*Algorismus proportionum*» («Вычисление пропорций»), в котором впервые использовал степени с дробными показателями и фактически вплотную подошёл к идее логарифмов. Когда же на смену абаку пришёл так называемый счёт на линиях, многочисленные руководства по нему стали называть «*Algorithmus linealis*», то есть правила счёта на линиях. Можно обратить внимание на то, что первоначальная форма *algorismi* спустя какое-то время потеряла последнюю букву, и слово приобрело более удобное для европейского произношения вид *algorism*. Позднее и оно подверглось искажению связанному со словом *arithmetic*.

В 1684 Готфрид Лейбниц в сочинении «*Nova Methodvs pro maximis et minimis, itemque tangentibus*…» впервые использовал слово «алгоритм» (*Algorithmo*) в ещё более широком смысле: как систематический способ решения проблем дифференциального исчисления. В XVIII веке в одном из германских математических словарей, *Vollstandiges mathematisches Lexicon* (изданном в Лейпциге в 1747), термин *algorithmus* всё ещё объясняется как понятие о четырёх арифметических операциях. Но такое значение не было единственным, ведь терминология математической науки в те времена ещё только формировалась. В частности, выражение *algorithmus infinitesimalis* применялось к способам выполнения действий с бесконечно малыми величинами. Пользовался словом алгоритм и Леонард Эйлер, одна из работ которого так и называется- «Использование нового алгоритма для решения проблемы Пелля. Понимание Эйлером алгоритма как синонима способа решения задачи уже очень близко к современному.

Однако потребовалось ещё почти два столетия, чтобы все старинные значения слова вышли из употребления. Этот процесс можно проследить на примере проникновения слова «алгоритм» в русский язык.

Историки датируют 1691 годом один из списков древнерусского учебника арифметики, известного как «Счётная мудрость». Это сочинение известно во многих вариантах (самые ранние из них почти на сто

лет старше) и восходит к ещё более древним рукописям XVI в. По ним можно проследить, как знание арабских цифр и правил действий с ними постепенно распространялось на Руси. Полное название этого учебника -«Сия книга, глаголемая по еллински и по-гречески арифметика, а по-немецки алгоризма, а по-русски цифирная счётная мудрость».

Таким образом, слово «алгоритм» понималось первыми русскими математиками так же, как и в Западной Европе. Однако его не было ни в знаменитом словаре В.И.Даля, ни спустя сто лет в «Толковом словаре русского языка» под редакцией Д.Н.Ушакова (1935). Зато слово «алгорифм» можно найти и в популярном дореволюционном Энциклопедическом словаре братьев Гранат, и в первом издании Большой Советской Энциклопедии (БСЭ), изданном в 1926. И там, и там оно трактуется одинаково: как правило, по которому выполняется то или иное из четырёх арифметических действий в десятичной системе счисления. Однако к началу XX в. для математиков слово «алгоритм» уже означало любой арифметический или алгебраический процесс, выполняемый по строго определённым правилам, и это объяснение также даётся в БСЭ.

Алгоритмы становились предметом все более пристального внимания учёных, и постепенно это понятие заняло одно из центральных мест в современной математике. Что же касается людей, от математики далёких, то к началу сороковых годов это слово они могли услышать разве что во время учебы в школе, в сочетании «алгоритм Евклида». Несмотря на это, алгоритм все ещё воспринимался как термин сугубо специальный, что подтверждается отсутствием соответствующих статей в менее объёмных изданиях. В частности, его нет даже в десятитомной Малой Советской Энциклопедии (1957), не говоря уже об однотомных энциклопедических словарях. Но зато спустя десять лет, в третьем издании Большой советской энциклопедии (1969) алгоритм уже характеризуется как одна из основных категорий математики, «не обладающих формальным определением в терминах более простых понятий, и абстрагируемых непосредственно из опыта». За сорок лет алгоритм превратился в одно из ключевых понятий математики, и признанием этого стало включение слова уже не в энциклопедии, а в словари. Например, оно присутствует в академическом «Словаре русского языка» (1981) именно как термин из области математики.

Одновременно с развитием понятия алгоритма постепенно происходила и его экспансия из чистой математики в другие сферы. И начало ей положило появление компьютеров, благодаря которому слово «алгоритм» вошло в 1985 во все школьные учебники информатики и обрело новую жизнь. Вообще можно сказать, что его сегодняшняя известность напрямую связана со степенью распространения компьютеров. Например, в третьем томе «Детской энциклопедии» (1959) о вычислительных машинах говорится немало, но они ещё не стали чем-то привычным и воспринимаются скорее как некий атрибут светлого, но достаточно далёкого будущего. Соответственно и алгоритмы ни разу не упоминаются на её страницах. Но уже в начале 70-х гг. прошлого столетия, когда компьютеры перестали быть экзотической диковинкой, слово «алгоритм» стремительно входит в обиход. В «Энциклопедии кибернетики» (1974) в статье «Алгоритм» он уже связывается с реализацией на вычислительных машинах, а в «Советской военной энциклопедии» (1976) даже появляется отдельная статья «Алгоритм решения задачи на ЭВМ». За последние два десятилетия компьютер стал неотъемлемым атрибутом нашей жизни, компьютерная лексика становится все более привычной. Слово «алгоритм» в наши дни известно, вероятно, каждому. Оно уверенно шагнуло даже в разговорную речь, и сегодня мы нередко встречаем в газетах выражения вроде «алгоритм поведения», «алгоритм успеха» или даже «алгоритм предательства».

1.3 Виды алгоритмов

Прикладные алгоритмы, предназначены для решения определенных прикладных задач. Алгоритм считается правильным, если он отвечает требованиям задачи (например, даёт физически правдоподобный результат). Алгоритм (программа) содержит ошибки, если для некоторых исходных данных он даёт неправильные результаты, сбои, отказы или не даёт никаких результатов вообще. Важную роль играют **рекурсивные алгоритмы** (алгоритмы, вызывающие сами себя до тех пор, пока не будет достигнуто некоторое условие возвращения). Начиная с конца XX - начала XXI века активно разрабатываются **параллельные алгоритмы**, предназначенные для вычислительных машин, способных выполнять несколько операций одновременно.

Рекурсия - метод определения функции через её предыдущие и ранее определенные значения, а так же способ организации вычислений, при котором функция вызывает сама себя с другим аргументом.

Большинство современных языков высокого уровня поддерживают механизм рекурсивного вызова, когда функция, как элемент структуры языка программирования, возвращающая вычисленное значение по своему имени, может вызывать сама себя с другим аргументом. Эта возможность позволяет напрямую

реализовывать вычисление рекурсивно определенных функций. Аппарат рекурсивных функций равномощен машине Тьюринга (см. ниже), и, следовательно, любой рекурсивный алгоритм может быть реализован итерационно.

Алгоритм - это точно определённая инструкция, последовательно применяя которую к исходным данным, можно получить решение задачи. Для каждого алгоритма есть некоторое множество объектов, допустимых в качестве исходных данных. Например, в алгоритме деления вещественных чисел делимое может быть любым, а делитель не может быть равен нулю.

Алгоритм служит, как правило, для решения не одной конкретной задачи, а некоторого класса задач. Так, алгоритм сложения применим к любой паре натуральных чисел. В этом выражается его свойство массовости, то есть возможности применять многократно один и тот же алгоритм для любой задачи одного класса.

Для разработки алгоритмов и программ используется **алгоритмизация** - процесс систематического составления алгоритмов для решения поставленных прикладных задач. Алгоритмизация считается обязательным этапом в процессе разработки программ и решении задач на ЭВМ. Именно для прикладных алгоритмов и программ принципиально важны детерминированность, результативность и массовость, а также правильность результатов решения поставленных задач.

Алгоритм может быть записан словами и изображён схематически. Обычно сначала (на уровне идеи) алгоритм описывается словами, но по мере приближения к реализации он обретает всё более формальные очертания и формулировку на языке, понятном исполнителю (например, машинный код). Например, для описания алгоритма применяются блок-схемы. Другим вариантом описания, не зависимым от языка программирования, является псевдокод.

Хотя в определении алгоритма требуется лишь конечность числа шагов, требуемых для достижения результата, на практике выполнение даже хотя бы миллиарда шагов является слишком медленным. Также обычно есть другие ограничения (на размер программы, на допустимые действия). В связи с этим вводят такие понятия как сложность алгоритма (временная, по размеру программы, вычислительная и др.).

Для каждой задачи может существовать множество алгоритмов, приводящих к цели. Увеличение эффективности алгоритмов составляет одну из задач современной информатики. В 50-х гг. XX века появилась даже отдельная её область – быстрые алгоритмы. В частности, в известной всем с детства задаче об умножении десятичных чисел обнаружился ряд алгоритмов, позволяющих существенно (в асимптотическом смысле) ускорить нахождение произведения.

Быстрые алгоритмы - область вычислительной математики, которая изучает алгоритмы вычисления заданной функции с заданной точностью с использованием как можно меньшего числа битовых операций.

Замечания. 1) Существует много разных способов для записи (описания) одного и того же алгоритма: текстовая форма записи; запись в виде блок-схемы; запись алгоритма на каком-либо алгоритмическом языке; представление алгоритма в виде машины Тьюринга или машины Поста. Выбор способа записи алгоритма зависит от нескольких причин. Если важна наглядность записи алгоритма, то разумно использовать блок-схему. Если алгоритм небольшой, то его можно записать в текстовой форме. При этом команды могут быть пронумерованы или записаны в виде сплошного текста.

2) Вне зависимости от выбранной формы записи элементарные шаги алгоритма (команды) при укрупнении объединяются в алгоритмические конструкции: последовательные, ветвящиеся, циклические, рекурсивные. В 1969 Э.Дейкстра доказал, что для записи любого алгоритма достаточно трёх основных алгоритмических конструкций: последовательных, ветвящихся, циклических.

3) Если задача имеет алгоритмическое решение, то можно придумать множество различных способов её решения, т.е. различных алгоритмов решения одной и той же задачи, на основе которого можно выбрать самый эффективный способ (наилучший) алгоритм.

1.3 Исполнитель алгоритмов

Понятие исполнителя невозможно определить с помощью какой-либо формализации. Исполнителем может быть человек, группа людей, робот, станок, компьютер, язык программирования и т.д. Важнейшим свойством, характеризующим любого из этих исполнителей, является то, что исполнитель умеет выполнять некоторые команды. Так исполнитель-человек умеет выполнять такие команды, как «встать», «сесть», «включить компьютер» и т.д., а исполнитель - язык программирования Бейсик - команды ***PRINT***, ***END***, ***LIST*** и другие аналогичные. Вся совокупность команд, которые данный исполнитель умеет выполнять, называется ***системой команд исполнителя (СКИ)***.

Пример 2. Рассмотрим исполнителя-робота, (**Рис. 1**) работа которого состоит в собственном перемещении по рабочему полю (квадрату произвольного размера, разделенному на клетки) и перемещении объектов, в начальный момент времени находящихся на "складе" (правая верхняя клетка).

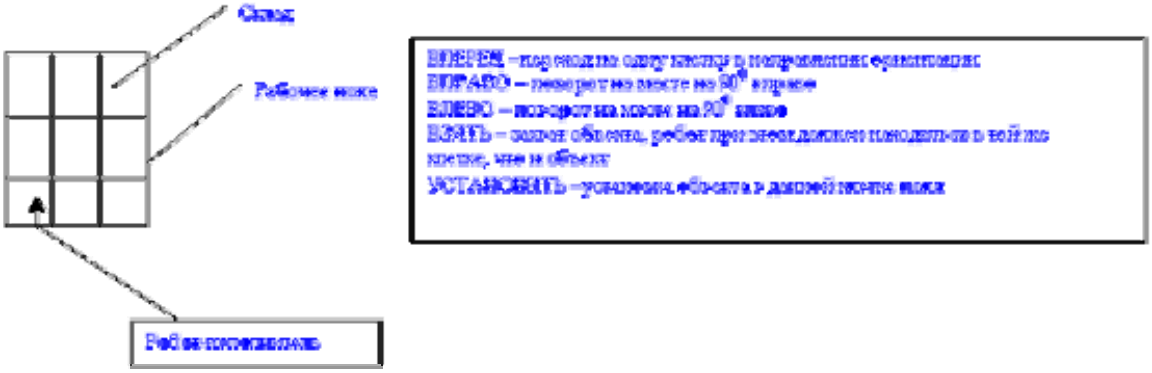


Рис.1. Исполнитель ***Робот***

Одно из принципиальных обстоятельств состоит в том, что исполнитель не вникает в смысл того, что он делает, но получает необходимый результат. В таком случае говорят, что исполнитель действует формально, т.е. отвлекается от содержания поставленной задачи и только строго выполняет некоторые правила, инструкции. Это - важная особенность алгоритмов. Наличие алгоритма формализует процесс решения задачи, исключает рассуждение исполнителя. Использование алгоритма и дает возможность решать задачу формально, механически исполняя команды алгоритма в указанной последовательности. Целесообразность предусматриваемых алгоритмом действий обеспечивается точным анализом со стороны того, кто составляет этот алгоритм. Введение в рассмотрение понятия «исполнитель» позволяет определить алгоритм как понятное и точное предписание исполнителю совершить последовательность действий, направленных на достижение поставленной цели. В случае исполнителя-робота мы имеем пример алгоритма «в обстановке», характеризующегося отсутствием каких-либо величин. Наиболее же распространенными и привычными являются алгоритмы работы с величинами - числовыми, символьными, логическими и т.д.

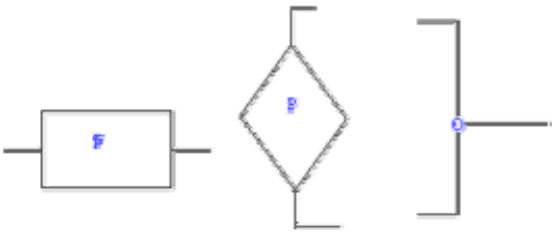


Рис. 2. Типы вершин

На **Рис. 2** изображены «функциональная» (а) вершина (имеющая один вход и один выход); "предикатная" (б) вершина, имеющая один вход и два выхода (в этом случае функция ***P*** передает управление по одной из ветвей в зависимости от значения ***P*** (***T***, т.е. ***True***, означает «истина», ***F***, т.е. ***False*** – «ложь»); «объединяющая» (в) вершина (вершина «слияния»), обеспечивающая передачу управления от одного из двух входов к выходу. Иногда вместо **T** пишут «да» (либо знак «+»), вместо ***F*** – «нет» (либо знак «-»).

Из данных элементарных блок-схем можно построить четыре блок-схемы (**Рис. 3**), имеющих особое значение для практики алгоритмизации.

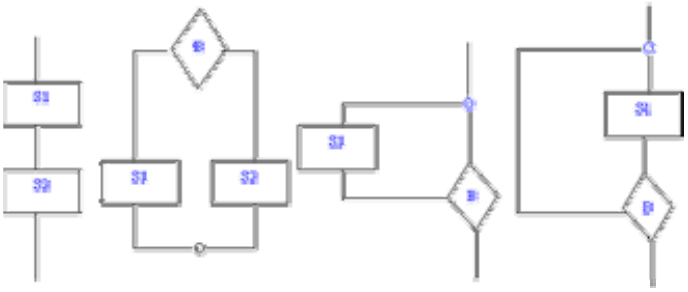


Рис. 3. Виды блок-схем

На **Рис. 3** изображены следующие блок-схемы: а - композиция, или следование; б - альтернатива, или развилка, в и г - блок-схемы, каждую из которых называют итерацией, или циклом (с предусловием (в), с постусловием (г)). ***S*1** и ***S*2** представляют собой в общем случае некоторые серии команд для соответствующего исполнителя, ***B*** - это условие, в зависимости от истинности (***T***) или ложности (**F**) которого управление передается по одной из двух ветвей. Можно доказать что для составления любого алгоритма достаточно представленных выше четырех блок-схем, если пользоваться их последовательностями и/или суперпозициями.

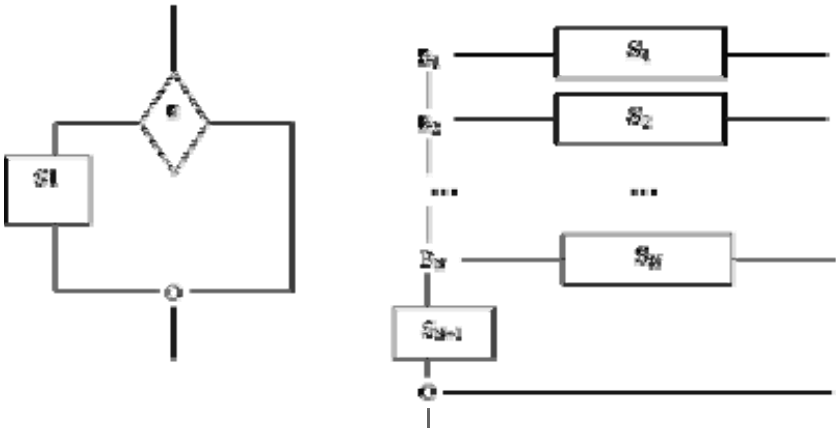


Рис. 4. Развитие структуры типа "альтернатива" а) неполная развилка; б) структура "выбор"

Блок-схема "альтернатива" может иметь и сокращенную форму в которой отсутствует ветвь ***S*2**(**Рис. 4а**). Развитием блок-схемы типа "альтернатива" является блок-схема "выбор" (**Рис.** 4**б**).

На практике при составлении блок-схем оказывается удобным использовать и другие графические знаки, некоторые из них приведены в **Табл. 1**.

Табл. 1. Символы, используемые при построении блок-схем

Символ	Описание
	Начало и конец алгоритма
	Вызов вспомогательного алгоритма
	Выполнение операций
	Ввод-вывод данных

1.4 Алгоритмический язык

Достаточно распространенным способом представления алгоритма является его запись на алгоритмическом языке, представляющем в общем случае систему обозначений и правил для единообразной и точной записи алгоритмов и исполнения их. Отметим, что между понятиями «алгоритмический язык» и «языки программирования» есть различие; прежде всего, под исполнителем в алгоритмическом языке может подразумеваться не только компьютер, но и устройство для работы «в обстановке». Программа, записанная на алгоритмическом языке, не обязательно, предназначена компьютеру. Практическая же реализация алгоритмического языка - отдельный вопрос в каждом конкретном случае.

Как и каждый язык, алгоритмический язык имеет свой словарь. Основу этого словаря составляют слова, употребляемые для записи команд, входящих в систему команд исполнителя того или иного алгоритма. Такие команды называют простыми командами. В алгоритмическом языке используют слова, смысл и способ употребления которых задан раз и навсегда. Эти слова называют служебными. Использование служебных слов делает запись алгоритма более наглядной, а форму представления различных алгоритмов – единообразной.

Алгоритм, записанный на алгоритмическом языке, должен иметь название. Название желательно выбирать так, чтобы было ясно, решение какой задачи описывает данный алгоритм. Для выделения названия алгоритма перед ним записывают служебное слово ***АЛГ (АЛГоритм)***. За названием алгоритма (обычно с новой строки) записывают его команды. Для указания начала и конца алгоритма его команды заключают в пару служебных слов ***НАЧ (НАЧало)*** и ***КОН (КОНец)***. Команды записывают последовательно.

Приведем последовательность записи алгоритма:

```
АЛГ название алгоритма
```

```
НАЧ
  Серия команд алгоритма
КОН
```

Например, алгоритм, определяющий движение исполнителя-робота, может иметь вид:

```
АЛГ в_склад
НАЧ
    ВПЕРЕД
    ВПРАВО
    ВПРАВО
    ВПЕРЕД
    ВПЕРЕД
КОН
```

При построении новых алгоритмов могут использоваться алгоритмы, составленные ранее. Алгоритмы, целиком используемые в составе других алгоритмов, называют ***вспомогательными алгоритмами***. Вспомогательным может оказаться любой алгоритм из числа ранее составленных. Не исключается также, что вспомогательным в определенной ситуации может оказаться алгоритм, сам содержащий ссылку на вспомогательные алгоритмы. Очень часто при составлении алгоритмов возникает необходимость использования в качестве вспомогательного одного и того же алгоритма, который к тому же может быть весьма сложным и громоздким. Было бы нерационально, начиная работу, каждый раз заново составлять и запоминать такой алгоритм для его последующего использования. Поэтому в практике широко используют так называемые ***встроенные*** (или ***стандартные***) вспомогательные алгоритмы, т.е. такие алгоритмы, которые постоянно имеются в распоряжении исполнителя. Обращение к таким алгоритмам осуществляется так же, как и к «обычным» вспомогательным алгоритмам. У исполнителя-робота встроенным вспомогательным алгоритмом может быть перемещение в склад из любой точки рабочего поля; у исполнителя - язык программирования Бейсик - это, например, встроенный алгоритм «***SIN***».

Алгоритм может содержать обращение к самому себе как вспомогательному и в этом случае его называют ***рекурсивным***. Если команда обращения алгоритма к самому себе находится в самом алгоритме, то такую рекурсию называют ***прямой***. Возможны случаи, когда рекурсивный вызов данного алгоритма происходит из вспомогательного алгоритма, к которому в данном алгоритме имеется обращение. Такая рекурсия называется ***косвенной***. Пример прямой рекурсии:

```
АЛГ движение
НАЧ
    вперед
    вперед
    вправо
    движение
КОН
```

Алгоритмы, при исполнении которых порядок следования команд определяется в зависимости от результатов проверки некоторых условий, называют ***разветвляющимися***. Для их описания в алгоритмическом языке используют специальную составную команду - ***команду ветвления***. Она соответствует блок-схеме "альтернатива" и также может иметь полную или сокращенную формуй. Применительно к исполнителю-роботу условием может быть проверка нахождения робота у края рабочего поля (край не_край); проверка наличия объекта в текущей клетке (есть/нет) и некоторые другие:

```
ЕСЛИ условие
  ТО серия 1
  ИНАЧЕ серия 2
ВСЕ
ЕСЛИ условие
  ТО серия
ВСЕ
ЕСЛИ край
  ТО вправо
  ИНАЧЕ вперед
ВСЕ
```

Ниже приводится запись на алгоритмическом языке команды выбора, являющейся развитием команды ветвления:

```
ВЫБОР
  ПРИ условие 1: серия 1
  ПРИ условие 2: серия 2
     .   .   .   .   .
  ПРИ условие N: серия N
```

```
      ИНАЧЕ серия N+1
    ВСЕ
```

Алгоритмы, при исполнении которых отдельные команды или серии команд выполняются неоднократно, называют ***циклическими***. Для организации циклических алгоритмов в алгоритмическом языке используют специальную составную команду цикла. Она соответствует блок-схемам типа "итерация" и может принимать следующий вид:

```
    ПОКА условие
    НЦ
       серия 1
    КЦ
или
    НЦ
       серия 1
    ДО условие
    КЦ
```

В случае составления алгоритмов работы с величинами можно рассмотреть и другие возможные алгоритмические конструкции, например, цикл с параметром или выбор. Подробно эти конструкции будут рассматриваться при знакомстве с реальными языками программирования. В заключение приведем алгоритм, составленный для исполнителя-робота, по которому робот переносит все объекты со склада в левый нижний угол рабочего поля (поле может иметь произвольные размеры):

```
    АЛГ до_края
    НАЧ
      ПОКА не_край
      НЦ
        вперед
      КЦ
    КОН
    АЛГ в_угол3
    НАЧ
      до_края
      вправо
      до_края
      вправо
    КОН
    АЛГ перенос
    НАЧ
      в_угол3
      ЕСЛИ есть
        ТО
          взять
          в_угол3
          установить
          перенос
        ИНАЧЕ
          в_угол3
      ВСЕ
    КОН
```

2. ТЕОРИЯ АЛГОРИТМОВ

Тео́рия алгори́тмов - наука, изучающая общие свойства и закономерности алгоритмов и разнообразные формальные модели их представления. К задачам теории алгоритмов относятся формальное доказательство алгоритмической неразрешимости задач, асимптотический анализ сложности алгоритмов, классификация алгоритмов в соответствии с классами сложности, разработка критериев сравнительной оценки качества алгоритмов и т. п.

2.1 Возникновение теории алгоритмов

Развитие теории алгоритмов начинается с доказательства К. Гёделем теорем о неполноте формальных систем, включающих арифметику, первая из которых была доказана в 1931. Возникшее в связи с этими теоремами предположение о невозможности алгоритмического разрешения многих математических проблем вызвало необходимость стандартизации понятия алгоритма. Первые стандартизованные варианты этого понятия были разработаны в 30-х годах XX века в работах А. Тьюринга, А. Чёрча и Э. Поста. Предложенные ими машина Тьюринга, машина Поста и лямбда-исчисление Чёрча оказались

эквивалентными друг другу. Основываясь на работах Гёделя, С. Клини ввел понятие рекурсивной функции, также оказавшееся эквивалентным вышеперечисленным.

Одним из наиболее удачных стандартизованных вариантов алгоритма является введённое А.А.Марковым понятие нормального алгоритма. Оно было разработано десятью годами позже работ Тьюринга, Поста, Чёрча и Клини в связи с доказательством алгоритмической неразрешимости ряда алгебраических проблем.

Следует отметить также немалый вклад в теорию алгоритмов, сделанный Д. Кнутом, А. Ахо и Дж. Ульманом. Одной из лучших работ на эту тему является книга «Алгоритмы: построение и анализ» Томаса Х. Кормена, Чарльза И. Лейзерсона, Рональда Л. Ривеста, Клиффорда Штайна.

Алан Тьюринг высказал предположение (известное как Тезис Чёрча - Тьюринга), что любой алгоритм в интуитивном смысле этого слова может быть представлен эквивалентной машиной Тьюринга. Уточнение представления о вычислимости на основе понятия машины Тьюринга (и других эквивалентных ей понятий) открыло возможности для строгого доказательства алгоритмической неразрешимости различных массовых проблем (то есть проблем о нахождении единого метода решения некоторого класса задач, условия которых могут варьироваться в известных пределах). Простейшим примером алгоритмически неразрешимой массовой проблемы является так называемая проблема применимости алгоритма (называемая также проблемой остановки). Она состоит в следующем: требуется найти общий метод, который позволял бы для произвольной машины Тьюринга (заданной посредством своей программы) и произвольного начального состояния ленты этой машины определить, завершится ли работа машины за конечное число шагов, или же будет продолжаться неограниченно долго.

В течение первого десятилетия истории теории алгоритмов неразрешимые массовые проблемы были обнаружены лишь внутри самой этой теории, а также внутри математической логики. Поэтому считалось, что теория алгоритмов представляет собой обочину математики, не имеющую значения для таких её классических разделов, как алгебра или анализ. Положение изменилось после того, как А.А. Марков и Э.Л.Пост в 1947 установили алгоритмическую неразрешимость известной в алгебре проблемы равенства для конечнопорождённых и конечноопределённых полугрупп. Впоследствии была установлена алгоритмическая неразрешимость и многих других «чисто математических» массовых проблем.

В настоящее время теория алгоритмов развивается, главным образом, по трем направлениям. Классическая теория алгоритмов изучает проблемы формулировки задач в терминах формальных языков, вводит понятие задачи разрешения, проводит классификацию задач по классам сложности и др. Теория асимптотического анализа алгоритмов рассматривает методы получения асимптотических оценок ресурсоемкости или времени выполнения алгоритмов, в частности, для рекурсивных алгоритмов. Асимптотический анализ позволяет оценить рост потребности алгоритма в ресурсах (например, времени выполнения) с увеличением объема входных данных. Теория практического анализа вычислительных алгоритмов решает задачи получения явных функции трудоёмкости, интервального анализа функций, поиска практических критериев качества алгоритмов, разработки методики выбора рациональных алгоритмов.

2.2 Анализ трудоёмкости алгоритмов

Целью анализа трудоёмкости алгоритмов является нахождение оптимального алгоритма для решения данной задачи. В качестве критерия оптимальности алгоритма выбирается трудоемкость алгоритма, понимаемая как количество элементарных операций, которые необходимо выполнить для решения задачи с помощью данного алгоритма. Функцией трудоемкости называется отношение, связывающие входные данные алгоритма с количеством элементарных операций.

Трудоёмкость алгоритмов по-разному зависит от входных данных. Для некоторых алгоритмов трудоемкость зависит только от объема данных, для других алгоритмов - от значений данных, в некоторых случаях порядок поступления данных может влиять на трудоемкость. Трудоёмкость многих алгоритмов может в той или иной мере зависеть от всех перечисленных выше факторов.

Одним из упрощенных видов анализа, используемых на практике, является асимптотический анализ алгоритмов. Целью асимптотического анализа является сравнение затрат времени и других ресурсов различными алгоритмами, предназначенными для решения одной и той же задачи, при больших объемах входных данных. Используемая в асимптотическом анализе оценка функции трудоёмкости, называемая **сложностью алгоритма**, позволяет определить, как быстро растет трудоёмкость алгоритма с увеличением объема данных. В асимптотическом анализе алгоритмов используются обозначения, принятые в математическом асимптотическом анализе.

В рамках классической теории осуществляется классификация задач по классам сложности (*P*-сложные, *NP*-сложные, экспоненциально сложные и др.). К классу *P* относятся задачи, которые могут быть решены за время, полиномиально зависящее от объёма исходных данных, с помощью детерминированной вычислительной машины (например, машины Тьюринга), а к классу *NP* - задачи, которые могут быть решены за полиномиально выраженное время с помощью недетерминированной вычислительной машины, то есть машины, следующее состояние которой не всегда однозначно определяется предыдущими. Работу такой машины можно представить как разветвляющийся на каждой неоднозначности процесс: задача считается решённой, если хотя бы одна ветвь процесса пришла к ответу. Другое определение класса *NP*: к классу *NP* относятся задачи, решение которых с помощью дополнительной информации полиномиальной длины, данной нам свыше, мы можем проверить за полиномиальное время. В частности, к классу *NP* относятся все задачи, решение которых можно *проверить* за полиномиальное время. Класс *P* содержится в классе *NP*. Классическим примером *NP*-задачи является задача о коммивояжёре.

Задача коммивояжёра (коммивояжёр - бродячий торговец) является одной из самых известных задач комбинаторной оптимизации. Задача заключается в отыскании самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый, совокупный критерий и т. п.) и соответствующие матрицы расстояний, стоимости и т. п. Как правило, указывается, что маршрут должен проходить через каждый город только один раз - в таком случае выбор осуществляется среди гамильтоновых циклов.

Поскольку класс *P* содержится в классе *NP*, принадлежность той или иной задачи к классу NP зачастую отражает наше текущее представление о способах решения данной задачи и носит неокончательный характер. В общем случае нет оснований полагать, что для той или иной *NP*-задачи не может быть найдено *P*-решение. Вопрос о возможной эквивалентности классов *P* и *NP* (то есть о возможности нахождения *P*-решения для любой *NP*-задачи) считается многими одним из основных вопросов современной теории сложности алгоритмов. Ответ на этот вопрос не найден до сих пор. Сама постановка вопроса об эквивалентности классов *P* и *NP* возможна благодаря введению понятия *NP*-полных задач. *NP*-полные задачи составляют подмножество *NP*-задач и отличаются тем свойством, что все *NP*-задачи могут быть тем или иным способом сведены к ним. Из этого следует, что если для *NP*-полной задачи будет найдено *P*-решение, то *P*-решение будет найдено для всех задач класса *NP*. Примером *NP*-полной задачи является задача о конъюнктивной форме

Исследования сложности алгоритмов позволили по-новому взглянуть на решение многих классических математических задач и найти для ряда таких задач (умножение многочленов и матриц, решение линейных систем уравнений и др.) решения, требующие меньше ресурсов, нежели традиционные.

2.3 Объекты алгоритмов

Одной из причин расплывчатости интуитивного понятия алгоритма является разнообразие объектов, с которыми работают алгоритмы. В вычислительных алгоритмах объектами являются числа. В алгоритме шахматной игры объектами являются фигуры и их позиции на шахматной доске. В алгоритме формирования текста – слова некоторого языка и правила переноса слов. Однако во всех этих случаях можно считать, что алгоритм имеет дело не с объектами реального мира, а с некоторыми изображениями этих объектов. Например, есть алгоритм сложения двух целых чисел. Результатом сложения числовых объектов 26 и 32 будет числовой результат 48. Но мы можем считать, что объектом этого алгоритма является входная последовательность, состоящая из двух символов: 48. При этом мы исходим из того, что имеется набор из 11 различных символов {0, 1. 2, 3, 4, 5, 6, 7. 8, 9, +}. Используемые символы будем называть буквами, а их набор – алфавитом. В общем случае буквами могут служить любые символы, требуется только, чтобы они были различны между собой и чтобы их число было конечным.

Любая конечная последовательность букв из некоторого алфавита называется словом в этом алфавите. Количество букв в слове называется длиной слова. Слово, в котором нет букв, называется пустым словом. Оно часто изображается символом «Λ» или «a_0». Так, алгоритм сложения двух целых чисел перерабатывает слово, которое состоит из двух слагаемых, разделённых символом «+», в слово, изображающее сумму.

Итак, объекты реального мира можно изображать словами в различных алфавитах. Это позволяет считать, что объектами работы алгоритмов могут быть только слова.

Слово, к которому применяется алгоритм, называется входным словом; слово, получаемое в результате работы алгоритма называется выходным. Совокупность слов, к которым применим алгоритм, называется областью применимости алгоритма. К сожалению, нельзя доказать, что все возможные объекты

можно описать словами, так как само понятие объекта не было формально (т.е. строго) определено. Но можно проверить, что для любого наугад взятого алгоритма, работающего не над словами, его объекты можно выразить так, что они становятся словами, а суть алгоритма от этого не меняется.

Любой алгоритм можно заменить другим. Такая замена называется кодированием. Например, пусть каждой букве первого алфавита ставится в соответствие код, представляющий собой слово во втором алфавите. В качестве второго алфавита достаточно иметь алфавит из двух букв, т.к. любое слово из любого алфавита можно закодировать в двухбуквенном алфавите с гарантией однозначного восстановления исходного слова. Следовательно, любой алгоритм можно свести к алгоритму над словами в алфавите {0, 1}, а перед применением алгоритма входное слово следует закодировать, после применения алгоритма выходное слово надо раскодировать.

Будем считать, что алгоритмы работают со словами, и мы формально описываем объекты-слова, над которыми работают алгоритмы, в некотором алфавите.

2.4 Машина Тьюринга

Маши́на Тью́ринга (МТ) - абстрактный исполнитель (абстрактная вычислительная машина).

Предложена Аланом Тьюрингом в 1936 для формализации понятия алгоритма. Машина Тьюринга является расширением конечного автомата и, согласно тезису Чёрча - Тьюринга, способна имитировать все другие исполнители (с помощью задания правил перехода), каким-либо образом реализующие процесс пошагового вычисления, в котором каждый шаг вычисления достаточно элементарен.

Те́зис Чёрча - Тью́ринга - фундаментальное утверждение для многих областей науки, таких, как теория вычислимости, информатика, теоретическая кибернетика и др. Это утверждение было высказано Алонзо Чёрчем и Аланом Тьюрингом в середине 1930-х годов. В самой общей форме оно гласит, что любая интуитивно вычислимая функция является частично вычислимой, или, что тоже самое, может быть вычислена некоторой машиной Тьюринга. Тезис Чёрча - Тьюринга невозможно строго доказать или опровергнуть, поскольку он устанавливает «равенство» между строго формализованным понятием частично вычислимой функции и неформальным понятием «интуитивно вычислимой функции».

Физический тезис Чёрча - Тьюринга гласит: Любая функция, которая может быть вычислена физическим устройством, может быть вычислена машиной Тьюринга.

У Тьюринга целью создания абстрактной воображаемой машины было получение возможности доказательства существования или несуществования алгоритмов решения различных задач. Руководствуясь этой целью, Тьюринг искал как можно более простую, «бедную» алгоритмическую схему, лишь бы она была универсальной.

Машина Тьюринга – это строгое математическое построение, математический аппарат (аналогичный, например, аппарату дифференциальных уравнений), созданный для решения определённых задач. Этот математический аппарат был назван «машиной» по той причине, что по описанию его составляющих частей и функционированию он похож на вычислительную машину. Принципиальное отличие машины Тьюринга от вычислительных машин состоит в том, что её запоминающее устройство представляет собой бесконечную ленту: у реальных вычислительных машин запоминающее устройство может быть как угодно большим, но обязательно конечным. Машину Тьюринга нельзя реализовать именно из-за бесконечности её ленты. В этом смысле она мощнее любой вычислительной машины.

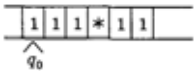
В состав машины Тьюринга входит бесконечная в обе стороны *лента* (возможны машины Тьюринга, которые имеют несколько бесконечных лент), разделённая на ячейки, и *управляющее устройство*, способное находиться в одном из *множества состояний* – *автомат* (головка для считывания/записи, управляемая программой). Число возможных состояний управляющего устройства конечно и точно задано. Управляющее устройство может перемещаться влево и вправо по ленте, читать и записывать в ячейки ленты символы некоторого конечного алфавита. Выделяется особый *пустой* символ, заполняющий все клетки ленты, кроме тех из них (конечного числа), на которых записаны входные данные. Управляющее устройство работает согласно *правилам перехода*, которые представляют алгоритм, *реализуемый* данной машиной Тьюринга. Каждое правило перехода предписывает машине, в зависимости от текущего состояния и наблюдаемого в текущей клетке символа, записать в эту клетку новый символ, перейти в новое состояние и переместиться на одну клетку влево или вправо. Некоторые состояния машины Тьюринга могут быть помечены как *терминальные*, и переход в любое из них означает конец работы, остановку алгоритма.

Машина Тьюринга называется *детерминированной*, если каждой комбинации состояния и ленточного символа в таблице соответствует не более одного правила. Если существует пара (ленточный символ - состояние), для которой существует 2 и более команд, такая машина Тьюринга называется *недетерминированной*.

С каждой машиной Тьюринга связаны два конечных алфавита: алфавит входных сигналов $A=\{a_0, a_1,\ldots,a_m\}$ и алфавит $Q=\{q_0, q_1,\ldots,q_p\}$. (С разными машинами Тьюринга могут быть связаны разные алфавиты A и Q). Состояние q_0 называется пассивным. Считается, что если машина попала в это состояние, то она закончила свою работу. Состояние q_1 называется начальным. Находясь в этом состоянии, машина начинает свою работу. Входное слово размещается на ленте по одной букве в расположенных подряд ячейках. Слева и справа от входного слова находятся только пустые ячейки (в алфавит A всегда входит «пустая» буква a_0 – признак того, что ячейка пуста.

Конкретная машина Тьюринга задаётся перечислением элементов множества букв алфавита A, множества состояний Q и набором правил, по которым работает машина. Они имеют вид: $q_i a_j \rightarrow q_{i1} a_{j1} dk$ (если головка находится в состоянии q_i, а в обозреваемой ячейке записана буква a_j, то головка переходит в состояние q_{i1}, в ячейку вместо a_j записывается a_{j1}, головка делает движение dk, которое имеет три варианта: на ячейку влево (L), на ячейку вправо (R), остаться на месте (N)). Для каждой возможной конфигурации $<q_i, a_j>$ имеется ровно одно правило. Правил нет только для заключительного состояния, попав в которое машина останавливается. Кроме того, необходимо указать конечное и начальное состояния, начальную конфигурацию на ленте и расположение головки машины.

Автомат может двигаться вдоль ленты влево или вправо, читать содержимое ячеек и записывать в ячейки буквы. Ниже представлена схема машины Тьюринга, автомат которой обозревает первую ячейку с данными:



Автомат каждый раз «видит» только одну ячейку. В зависимости от того, какую букву a_i он видит, а также в зависимости от своего состояния q_j, автомат может выполнять следующие действия: записать новую букву в обозреваемую ячейку; выполнить сдвиг по ленте на одну ячейку вправо/влево или остаться на месте; перейти в новое состояние. То есть у машины Тьюринга есть три вида команд. Каждый раз для очередной пары (q_j, a_i) машина Тьюринга может выполнить определённую команду в соответствии с программой.

Машина Тьюринга представляет собой простейшую вычислительную машину с линейной памятью, которая согласно формальным правилам преобразует входные данные с помощью последовательности элементарных действий. Элементарность действий заключается в том, что действие меняет лишь небольшой кусочек данных в памяти (в случае машины Тьюринга - лишь одну ячейку), и число возможных действий конечно. Несмотря на простоту машины Тьюринга на ней можно вычислить всё, что можно вычислить на любой другой машине, осуществляющей вычисления с помощью последовательности элементарных действий. Это свойство называется полнотой.

Машина Тьюринга может выполнять все возможные преобразования слов, реализуя тем самым все возможные алгоритмы.

Один из естественных способов доказательства того, что алгоритмы вычисления, которые можно реализовать на одной машине, можно реализовать и на другой, - это имитация первой машины на второй. Имитация заключается в следующем. На вход второй машине подаётся описание программы (правил работы) первой машины D и входные данные X, которые должны были поступить на вход первой машины. Нужно описать такую программу (правила работы второй машины), чтобы в результате вычислений на выходе оказалось то же самое, что вернула бы первая машина, если бы получила на вход данные X.

На машине Тьюринга можно имитировать (с помощью задания правил перехода) все другие исполнители, каким-либо образом реализующие процесс пошагового вычисления, в котором каждый шаг вычисления достаточно элементарен. На машине Тьюринга можно имитировать машину Поста, нормальные алгоритмы Маркова и любую программу для обычных компьютеров, преобразующую входные данные в выходные по какому-либо алгоритму. В свою очередь, на различных абстрактных исполнителях можно имитировать Машину Тьюринга. Исполнители, для которых это возможно, называются *полными по Тьюрингу*.

Богатство возможностей машины Тьюринга проявляется в том, что если какие-то алгоритмы A и B реализуются машинами Тьюринга, то можно построить машины Тьюринга, реализующие различные композиции алгоритмов A и B. Например, «Выполнить A, затем выполнить B» или «Выполнить A. Если в результате получилось слово «да», выполнить B. В противном случае не выполнять B» или «Выполнять поочерёдно A, B, пока B не даст ответ 0».

Очевидно, что такие композиции также являются алгоритмами, поэтому их реализация посредством машины Тьюринга подтверждает, что конструкция Тьюринга является универсальным исполнителем.

Всякий алгоритм может быть реализован соответствующей машиной Тьюринга. Все алгоритмы, придуманные человечеством в течение столетий могут быть реализованы машиной Тьюринга.

Как уже упоминалось, каждый алгоритм предназначен для какого-то конкретного исполнителя, у каждого исполнителя есть своя система команд, есть свой круг задач. Тьюрингом же был построен универсальный исполнитель, который может решить любую известную задачу. Этот фундаментальный результат был получен в то время, когда универсальных вычислительных машин ещё не существовало. Более того, сам факт построения воображаемого универсального исполнителя позволил высказать предположение о целесообразности построения универсальной вычислительной машины, которая бы могла решать любые задачи при условии соответствующего кодирования исходных данных и разработки соответствующей программы действий исполнителя.

Есть программы для обычных компьютеров, имитирующие работу машины Тьюринга. Но данная имитация неполная, так как в машине Тьюринга присутствует абстрактная бесконечная лента. Бесконечную ленту с данными невозможно в полной мере имитировать на компьютере с конечной памятью (суммарная память компьютера - оперативная память, жёсткие диски, различные внешние носители данных, регистры и кэш процессора и др. - может быть очень большой, но, тем не менее, всегда конечна).

2.5 Машина Поста

Машина Поста (1937**)-** абстрактная вычислительная машина, предложенная американским математиком и логиком (основатель многозначной логики) Эмилем Леоном Постом (1897-1954), которая отличается от машины Тьюринга большей простотой. Обе машины «эквивалентны» и были созданы для уточнения понятия «алгоритм».

Машина Поста состоит из каретки (или считывающей и записывающей головки) и разбитой на секции бесконечной в обе стороны ленты. Каждая секция ленты может быть либо пустой - 0, либо помеченной меткой 1 (в машине Поста в ячейках бесконечной ленты можно записывать всего два знака: 0 и 1, но это ограничение не влияет на её универсальность, т.к. любой алфавит может быть закодирован двумя знаками). За один шаг каретка может сдвинуться на одну позицию влево или вправо, стоять на месте. Машина умеет читать содержимое, считать, поставить или уничтожить символ в том месте, где она стоит (умеет стирать и записывать 0 или 1). Работа машины определяется программой, состоящей из конечного числа строк. Всего команд шесть. Как и машина Тьюринга, машина Поста может находиться в различных состояниях, но каждому состоянию соответствует не строка состояния с клетками, а некоторая команда одного из шести типов. Для работы машины нужно задать программу и ее начальное состояние (т. е. состояние ленты и позицию каретки). После запуска возможны варианты: работа может закончиться невыполнимой командой (стирание несуществующей метки или запись в помеченное поле); работа может закончиться командой *Stop*; работа никогда не закончится.

«Машиной» эта математическая конструкция называется потому, что в ней используются некоторые понятия реальных машин – память, команда и пр. Машина Поста, несмотря на внешнюю простоту, может производить различные вычисления, для чего надо задать начальное состояние каретки и программу, которая эти вычисления сделает.

Алгоритм (по Посту) – программа для машины Поста, приводящая к решению поставленной задачи.

Тезис Поста: Всякий алгоритм представим в форме машины Поста.

Этот тезис одновременно является формальным определение алгоритма.

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

2.6 Алгоритмически неразрешимые задачи и вычислимые функции

Существуют задачи, которые алгоритмически разрешить невозможно.

В начале 20-го века немецкий математик Давид Гильберт в 1900 сформулировал 23 математические проблемы. Для нас интересно, что сегодня доказана невозможность построения алгоритма для решения десятой проблемы Гильберта о диофантовых уравнениях. Доказана также невозможность создания универсального (пригодного для любой программы) алгоритма отладки программы. Впрочем, для конкретных алгоритмов и некоторых классов алгоритмов проблему останова и/или отладки соответствующей программы решить можно. Так, программа, состоящая только из линейных конструкций всегда закончит свою работу.

Сформулированная Г. Лейбницем (1646-1716) проблема проверки правильности любых математических утверждений также является алгоритмически неразрешимой. Напомним, что в

современной математике почти все математические теории строятся на аксиоматической основе. Суть соответствующего аксиоматического метода состоит в следующем: все предложения (теоремы) данной теории получаются посредством формально-логического вывода из нескольких предложений (аксиом), принимаемых в данной теории без доказательства. Ранее других была осуществлена аксиоматизация геометрии. Так вот, в рамках теории алгоритмов был получен отрицательный ответ на вопрос об алгоритмической разрешимости проблемы распознавания выводимости (построения общего метода решения любой математической задачи). В 1936 американский математик Чёрч доказал следующую теорему: Проблема распознавания выводимости алгоритмически неразрешима. Тем самым выяснилась не только причина безуспешности всех прошлых попыток создания соответствующего алгоритма, но и доказана бессмысленность дальнейших попыток.

Вернёмся к задачам, которые имеют алгоритмическое решение. Ранее показано, что если задача имеет решение, то можно написать и машину Тьюринга и машину Поста для решения этой задачи. При этом на выходные данные накладываются формальные ограничения (входные слова кодируются в некотором алфавите), а сами алгоритмы сконструированы в разных алгоритмических моделях (схемах). Возникает вопрос: можно ли в принципе говорить об общих свойствах алгоритмов при такой конкретизации?

В теории алгоритмов строго доказано, что любой алгоритм, описанный в одной модели, может быть описан и в другой. Такая взаимная сводимость алгоритмических моделей позволила создать систему понятий, не зависящую от выбора конкретной формализации понятия алгоритма. В основе этой системы лежит понятие вычислимой функции.

Относительно каждого алгоритма A можно сказать, что он вычисляет значение функции FA (реализует функцию FA) при некоторых значениях входных величин.

Функция, вычисляемая некоторыми алгоритмами, называется вычислимой функцией (алгоритмически вычислимой).

Введённое понятие вычислимой функции, так же, как и понятие алгоритма, является интуитивным.

Фактически алгоритм – это способ задания функции. Функции могут задаваться и другими способами, например, таблицей и формулой. Однако существуют и такие задачи, в которых связь между входными и выходными параметрами настолько сложна, что нельзя составит алгоритм преобразования входных данных в результат. Такие функции являются не вычислимыми.

Понятия алгоритма и вычислимой функции являются наиболее фундаментальными понятиями информатики и математики. Систематическое изучение алгоритмов и различных моделей вычислений привело к созданию особой дисциплины, пограничной между математикой и информатикой – **теории алгоритмов**, в которой выделен раздел «*теории вычислимости*».

Теория вычислимых (с помощью компьютеров) функций появилась в 30-е году 20-го столетия, когда никаких компьютеров ещё не было. Первые компьютеры появились в 40-х годах, и их появление стало возможным именно благодаря достижениям теории вычислимости. Так, в рамках теории алгоритмов было сформулировано понятие вычислительной машины и было показано, что для осуществления всевозможных преобразований вовсе не обязательно строить каждый раз специализированные вычислительные устройства: всё это можно сделать на одном универсальном устройстве при помощи подходящей программы и соответствующего кодирования.

2.7 Понятие сложности алгоритма

Если для решения задачи существует один алгоритм, то можно придумать и много других алгоритмов для решения этой же задачи.

Как правило, мы высказываем суждение об алгоритме на основе его оценки исполнителем-человеком. Алгоритм кажется нам сложным, если даже после внимательного его изучения мы не можем понять, что же он делает. Мы можем назвать алгоритм сложным и запутанным из-за того, что он обладает разветвлённой логической структурой, содержащей много проверок условий и переходов. Однако для компьютера выполнение программы, реализующей такой алгоритм, не составит труда, т.к. он выполняет одну команду за другой, и для компьютера неважно – операция ли это умножения или проверка условия.

Более того, мы можем написать громоздкий алгоритм, в котором выписаны подряд повторяющиеся действия (без использования циклической структуры). Однако с точки зрения компьютерной реализации практически нет никакой разницы, использован ли в программе оператор цикла (например, 10 раз на экран выводится слово «Привет») или 10 раз последовательно выписаны операторы вывода на экран слова «Привет». Поэтому для оценки эффективности алгоритмов введено понятие сложности алгоритма.

Вычислительным процессом, порождённым алгоритмом, называется последовательность шагов алгоритма, пройденных при исполнении этого алгоритма.

В дальнейшем будем понимать под сложностью алгоритма количество элементарных действий в вычислительном процессе этого алгоритма, как функцию от исходных данных: именно в вычислительном процессе, а не в самом алгоритме. Для сравнения сложности разных алгоритмов необходимо, чтобы сложность подсчитывалась в одних и тех же элементарных действиях.

Временная сложность алгоритма – время T, необходимое для его выполнения в зависимости от исходных данных. Оно равно произведению числа элементарных действий k на среднее время выполнения одного действия t: $T=kt$.

Поскольку t зависит от исполнителя, реализующего алгоритм, то естественно считать, что сложность алгоритма в первую очередь определяется значением k. Очевидно, что в наибольшей степени количество операций при выполнении алгоритма зависит от количества обрабатываемых данных. Действительно, для упорядочивания по алфавиту списка из 100 фамилий требуется существенно меньше операций, чем для упорядочивания списка из 100000 фамилий. Поэтому сложность алгоритма выражают в виде функции от объёма входных данных.

Пусть есть алгоритм A. Для него существуют параметр n, характеризующий объём обрабатываемых алгоритмом данных, этот параметр часто называют размерностью задачи. Обозначим через $T(n)$ время выполнения алгоритма в худшем случае, через f – некую функцию от n.

Будем говорить, что $T(n)$ алгоритма имеет порядок роста $f(n)$, или алгоритм имеет теоретическую сложность $O(f(n))$ (читается «о большое от $f(n)$», если для $T(n)$ найдётся такая константа c, что, начиная с некоторого n0, выполняется условие $T(n) \le cf(n)$. Здесь предполагается, что функция $f(n)$ неотрицательна, по крайней мере при $n \ge n_0$.

Так, например, алгоритм, выполняющий только операции чтения данных и занесения их в оперативную память, имеет линейную сложность $O(n)$. Алгоритм сортировки методом прямого выбора имеет квадратичную сложность $O(n^2)$, так как при сортировке любого массива этот алгоритм будет выполнять $(n^2-n)/2$ операций сравнений (при этом операций перестановок вообще может не быть, например, на упорядоченном массиве). А сложность алгоритма умножения матриц (таблиц) размера nxn будет уже кубической $O(n^3)$, т.к. для вычисления каждого элемента результирующей матрицы требуется n умножений и n-1 сложений, а всего этих элементов n^2.

Для решения задачи могут быть разработаны алгоритмы различной сложности. Логично воспользоваться лучшим среди них, т.е. имеющим наименьшую сложность.

Наряду со сложностью важной характеристикой алгоритма является **эффективность**. Под эффективностью понимается выполнение следующего требования: не только весь алгоритм, но и каждый шаг его должны быть такими, чтобы исполнитель был способен выполнить их за разумное время. Например, если алгоритм, выдающий прогноз погоды на ближайшие сутки, будет выполняться неделю, то такой алгоритм просто-напросто никому не нужен.

Если мы рассматриваем алгоритмы, реализующиеся на компьютере, то к требованию выполнения за разумное время прибавляется требование выполнения в ограниченном объёме оперативной памяти.

2.8 Анализ алгоритмов поиска

Рассмотрим теперь классические задачи поиска и обсудим алгоритмы, реализующие эти задачи, т.е. изучим свойства алгоритмов. Остановимся только на двух алгоритмах, предназначенных для решения этой востребованной задачи.

Рассматривая различные алгоритмы решения одной и той же задачи, полезно проанализировать, сколько вычислительных ресурсов (время работы, память), и выбрать наиболее эффективный. Однако вначале надо договориться, какая модель вычислений будет использоваться. Будем считать, что наши алгоритмы выполняются на обычной однопроцессорной машине с произвольным доступом к памяти (данным).

В алгоритмах поиска существует две возможности окончания работы: либо поиск оказался удачным, т.е. позволил определить положение соответствующего элемента, либо он оказался неудачным, т.е. показал, что необходимого элемента в данном объёме информации нет. Хотя целью поиска является значение элемента, алгоритмы поиска в случае удачного окончания выдают местоположение искомого элемента, например, номер элемента в массиве. В качестве критерия оценки алгоритма используют такую характеристику, как сложность.

Известны алгоритмы последовательного поиска в неупорядоченной последовательности (неупорядоченном массиве) и в упорядоченном массиве. В задачи поиска может входить: поиск

минимального элемента в неупорядоченном массиве; поиск в неупорядоченном массиве максимального и минимального элементов одновременно и др. В качестве примера рассмотрим поиск элемента в большом упорядоченном массиве информации находящимся в оперативной памяти компьютера. Для решения этой задачи разработаны эффективные алгоритмы, наиболее распространённым из них является алгоритм бинарного (двоичного) поиска, его иногда называют логарифмическим поиском, или методом деления пополам (дихотомией).

Основная идея бинарного поиска довольно проста, детали же нетривиальны, и правильно работающий алгоритм удаётся написать далеко не с первого раза. Одна из наиболее популярных реализаций этого алгоритма использует два указателя (l и u), соответствующие нижней и верхней границам поиска. С помощью этого алгоритма ищется элемент k в упорядоченном по возрастанию массиве a, содержащем n элементов.

Алгоритм бинарного поиска в упорядоченном массиве состоит из стадий:

1. Начальная установка: l=1, u=n.
2. Если $u<n$, то алгоритм окончен неудачно. В противном случае найти середину $[l;u]$. В этот момент мы знаем, что если k есть в массиве, то выполняются неравенства $a_l \le k \le a_u$. Установить i=[(l+u)/2]. Теперь i указывает примерно в середину рассматриваемой части массива.
3. Если $k<a_i$, то перейти к шагу 4, если $k>a_i$, то перейти к шагу 5, если $k=a_i$, алгоритм окончен удачно.
4. Установить u=i-1 и перейти к шагу 2.
5. Установить l=i+1 и перейти к шагу 2.

Шаг 3 алгоритма бинарного поиска выполняется порядка $\log_2 n$ раз, т.е. данный алгоритм имеет логарифмическую сложность по числу сравнений.

Перейдём теперь к анализу сортировки – одному из наиболее распространённых процессов современной обработки данных. Сортировкой называется распределение элементов множества по группам с определёнными правилами. Например, сортировка элементов массива, в результата которой получается массив, каждый элемент которого, начиная со второго, не больше стоящего от него слева, называется сортировкой по невозрастанию.

Здесь мы будем рассматривать только алгоритмы внутренние сортировки, которые применяются для переупорядочивания данных, которые полностью располагаются в оперативной (внутренней) памяти. В этом случае мы имеем прямой доступ к элементам массива. В отличие от внутренней сортировки, существует внешняя сортировка, алгоритмы такой сортировки используют память на внешних носителях (например, сортировка данных с последовательным доступом, хранящихся в файлах на жёстком диске).

Способов сортировки очень много, их можно разбить на группы в зависимости от идей, лежащей в их основе. Перечислим их для следующей задачи: дан одномерный массив целых чисел, требуется отсортировать его так, чтобы все элементы были расположены в порядке неубывания: $a[i] \le a[i+1]$, i=1, 2,…, n-1.

Типичным алгоритмом, относящимся к обменным сортировкам является метод пузырька, который получил своё название по ассоциации: если мы будем сортировать этим алгоритмом массив по убыванию, то минимальный элемент «всплывает», а «тяжёлые» элементы опускаются на одну позицию к началу массива при каждом шаге алгоритма. Алгоритм сортировки «пузырьком» имеет квадратичную сложность $O(n^2)$. На практике этот метод используется редко, т.к. в нём много раз приходится просматривать массив, поэтому программа работает долго.

Другой метод обменной сортировки – сортировка выбором. В этом методе сначала находится элемент в массиве из n элементов. Пусть его место имеет номер max. Он меняется местами с элементом, стоящим на n-ом месте, при условии, что $n \ne max$. Из оставшихся неупорядоченными n-1 первых элементов снова выделяется наибольший и меняется местами с элементом, стоящим на (n-1)-м месте и т.д. Алгоритм заканчивает свою работу, когда элементы, стоящие на 1-м и 2-м местах в массиве, будут напечатаны (для этого понадобится n-1 итерация алгоритма). Аналогично данный алгоритм можно применять и к наименьшим элементам.

Алгоритм сортировки выбором имеет квадратичную сложность $O(n^2)$ относительно операций сравнения и линейную сложность $O(n)$ относительно операций перестановки. Данный алгоритм целесообразно применять, если операция обмена над элементами массива трудоёмка, например, если элементом массива является запись с большим числом полей.

Сортировка выбором и сортировка «пузырьком» относятся к обменным сортировкам с убывающем шагом. Действительно, после выполнения каждой итерации алгоритма, количество элементов в неотсортированной части уменьшается на единицу. Сортировка вставками построена на ином принципе.

Вначале упорядочивается два первых элемент массива. Они образуют начальное упорядоченное множество *S*. Далее на каждом шаге берётся следующий по порядку элемент и вставляется в уже упорядоченное множество *S* так, чтобы слева от него все элементы были не больше, а справа – не меньше обрабатываемого. Место для вставки текущего элемента в упорядоченное множество *S* ищется методом деления пополам. Алгоритм сортировки заканчивает свою работу, когда элемент, первоначально стоящий на n-м месте, будет вставлен на соответствующее ему место.

Алгоритм сортировки вставками имеет квадратичную сложность $O(n^2)$ по числу присваиваний и сложность $O(n\log_2 n)$ по числу сравнений.

Все рассмотренные выше алгоритмы сортировки имеют квадратичную сложность, по крайней мере, по одному из параметров.

Существуют и более эффективные алгоритмы сортировки; сложность подобных универсальных алгоритмов составляет $O(n\log_2 n)$ по каждому из параметров.

Рассмотренные алгоритмы сортировок вставками, выбором, пузырьком являются примерами алгоритма, действующего по шагам: в отсортированную часть добавляются новые элементы один за другим. Существует другой подход, основанный на методе «разделяй и властвуй».

Многие алгоритмы по своей природе рекурсивны: решая некоторую задачу, они вызывают самих себя для решения её подзадач. Идея метода сортировки слиянием состоит как раз в этом. Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова. Наконец, их решения комбинируются и получается решение исходной задачи.

Для задачи сортировки эти три этапа выглядят следующим образом.

1. Сначала мы разбиваем массив на две половины.
2. Затем сортируем каждую из половин отдельно.
3. После этого соединяем два упорядоченных массива половинного размера в один.

Это соотношение влечёт $T(n)=O(n\log_2 n)$. Следовательно, для больших n сортировка слиянием эффективнее рассмотренных ранее алгоритмов сортировки, имеющих сложность $O(n^2)$.

Отметим, что часто разница между плохим и хорошим алгоритмом более существенна, чем разница между быстрым и медленным компьютером.

3. АЛГОРИТМЫ В КОМПЬЮТЕРЕ

Несмотря на огромное фактическое разнообразие, все существующие компьютеры и вычислительные системы с точки зрения пользователя условно можно разделить на две большие группы: последовательные и параллельные.

В простейшей интерпретации **последовательные компьютеры** выглядят следующим образом. Имеются два основных устройства. Одно из них, называемое *процессором* (центральным процессором, решающим устройством, арифметико-логическим устройством и т.п.), предназначено для выполнения некоторого ограниченного набора простых операций. В набор операций обычно входят сложение, вычитание и умножение чисел, логические операции над отдельными разрядами и их последовательностями, операции над символами и многое другое. Наборы операций, выполняемые процессорами разных компьютеров, могут отличаться как частично, так и полностью. Другое устройство, называемое *памятью* (запоминающим устройством и т.п.), предназначено для хранения всей информации, необходимой для организации работы процессора. Процессор является активным устройством, т.е. он имеет возможность преобразовывать информацию. Память является пассивным устройством, т.е. она не имеет такой возможности. Процессор и память связаны между собой *каналами* обмена информацией.

Рис. 5. Абстрактная модель последовательного компьютера.

Работа однопроцессорного компьютера заключается в последовательном выполнении отдельных команд. Каждая команда содержит информацию о том, какая операция из заданного набора должна быть выполнена, а также из каких ячеек памяти должны быть взяты аргументы операции и куда должен быть помещен результат. Описание упорядоченной последовательности команд в виде *программы* находится в памяти. Там же размещаются необходимые для реализации алгоритма начальные данные и результаты

промежуточных вычислений. Координирует работу всех узлов компьютера *устройство* управления. Оно организует последовательную выборку команд из памяти и их расшифровку, передачу из памяти в процессор операндов, а из процессора в память результатов выполнения команд, управляет работой процессора. Ввод начальных данных и выдачу результатов осуществляет *устройство ввода-вывода*.

По такой схеме устроены все однопроцессорные компьютеры. И это справедливо как для первых в истории электронных вычислительных машин - медленно работающих монстров, занимающих огромные помещения и потребляющих чудовищное количество энергии, так и для современных компактных высокоскоростных персональных компьютеров, размещающихся на письменном столе и потребляющих энергию меньше, чем обычная электрическая лампочка. Конечно, в действительности схемы однопроцессорных компьютеров обрастают большим числом дополнительных деталей. Но всегда процессор является *единственным* устройством, выполняющим полезную с точки зрения пользователя работу. В этом смысле у любого компьютера все другие устройства по отношению к процессору оказываются обслуживающими, и их работа направлена только на то, чтобы обеспечить наиболее эффективный режим функционирования процессора.

Каким бы сложным ни был однопроцессорный компьютер, построенный по классическим канонам, в основе его архитектуры и организации процесса функционирования всегда лежит *принцип последовательного выполнения отдельных действий*. С точки зрения пользователя отклонения от этого принципа, как правило, не существенны. Именно поэтому такие компьютеры и называются *последовательными*.

На каждом конкретном последовательном компьютере время реализации любого алгоритма пропорционально числу выполняемых операций, и почти не зависит от того, как внутренне устроен сам алгоритм. Конечно, какие-то различия во временах реализаций могут появляться. Но они невелики и в обычной практике их можно не принимать во внимание. Это свойство последовательных компьютеров исключительно важно и влечёт за собой разнообразные следствия. Пожалуй, самым главным из них является то, что для таких компьютеров оказалось возможным создавать компьютерно-независимые или, как их называют иначе, *машинно-независимые* языки программирования. По замыслу их создателей любая программа, написанная на любом из таких языков, должна без какой-либо переделки реализовываться на любой последовательной машине. Единственное, что формально требовалось для обеспечения работы программы, - это наличие на машине компилятора с соответствующего языка. Подобные языки стали возникать в большом количестве: Алгол, Кобол, Фортран, Си и др. Многие из них успешно используются до сих пор. Поскольку эти языки ориентированы на последовательные компьютеры, их также стали называть последовательными. Во всех программах, написанных на последовательных языках программирования, порядок выполнения команд всегда является строго *последовательным* и при заданных входных данных фиксируется *однозначно*.

Для математиков и разработчиков прикладного программного обеспечения такая ситуация открывала заманчивую перспективу. Не нужно было вникать в устройство вычислительных машин, так как языки программирования по существу мало чем отличались от языка математических описаний. В разработке вычислительных алгоритмов становились очевидными главные целевые функции их качества - минимизация числа выполняемых операций и устойчивость к влиянию ошибок округления. И больше ничего об алгоритмах не надо было знать, поскольку никаких причин для получения каких-либо других знаний и, тем более, знаний о структуре алгоритмов не возникало.

Привлекательность подобной перспективы на долгие годы сделала последовательную организацию вычислений неявным и во многом даже неосознанным фундаментом развития не только численных методов, но и всей вычислительной математики. Заметим, что по своему влиянию на общую направленность исследований в вычислительных делах эта перспектива и в настоящее время во многом остается доминирующей.

На самом деле все, что связано с последовательными компьютерами, развивалось и достаточно сложно, и в какой-то степени драматично. Погоня за производительностью и конкуренция привели к тому, что появилось много разных последовательных машин. Для каждой из них приходилось делать свой компилятор, так или иначе учитывающий особенности конкретной машины. Не в каждом компиляторе удавалось оптимально учитывать эти особенности на всем множестве программ. Поэтому в реальности при переносе программ с одной машины на другую многие программы приходилось модифицировать. Объем изменений мог быть большим или малым и зависел от сложности машин и языков программирования. Проблема переноса программ с одного последовательного компьютера на другой последовательный компьютер становилась со временем все более актуальной и были предприняты значительные усилия на ее решение. В первую очередь, за счет введения различных стандартов на конструирование компьютеров и правила написания программ.

Машины, которые принято называть последовательными, можно называть таковыми лишь с некоторой оговоркой, поскольку в каждый момент времени в них независимо или, другими словами, *параллельно* выполняется много различных действий. Именно, реализуются какие-то операции, передаются данные от одного устройства к другому, происходит обращение к памяти и т.п. Весь этот параллелизм учитывается при создании компилятора. Степень учёта параллелизма компилятором прямым образом сказывается на эффективности работы программ. Однако если параллелизм не виден через язык программирования, то для пользователя он как бы и не существует. Поэтому с точки зрения пользователя можно считать последовательными любые машины, эффективное общение с которыми осуществляется на уровне последовательных языков программирования. Подобная трактовка удобна для пользователей. Но есть в ней и серьезная опасность. Уповая на долговременную перспективу общения с вычислительной техникой на уровне последовательных языков, можно пропустить момент, когда количественные изменения в технике перейдут в качественные и общение с ней при помощи таких языков окажется невозможным. И тогда вроде бы совсем неожиданно, вдруг возникает вопрос о том, что же делать дальше. Именно это и произошло в истории освоения вычислительной техники.

В развитии вычислительной техники многое определяется стремлением повысить производительность и увеличить объем быстрой памяти. Мощности первых компьютеров были очень малы. Поэтому сразу после их появления стали предприниматься попытки объединения нескольких компьютеров в единую систему. Идея была чрезвычайно проста: если мощности одного компьютера не хватает для решения конкретной задачи, то нужно разделить задачу на две части и решать каждую часть на своем компьютере. А чтобы было удобно передавать данные с одного компьютера на другой, необходимо соединить сами компьютеры подходящими по пропускной способности *линиями связи*. Так появились двухмашинные комплексы. Естественно, на них можно было решать задачи примерно вдвое быстрее. Аналогичным образом строились многомашинные комплексы, объединяющие три, четыре, пять и более отдельных однопроцессорных компьютеров в единую систему. Соответственно повышалась и мощность комплексов. Больших проблем с разделением исходной задачи на несколько независимых подзадач не возникало, поскольку их общее число было невелико.

Несмотря на плодотворность идеи объединения отдельных машин в единый комплекс, долгое время она не получала необходимое развитие. Основные трудности на пути её практической реализации были связаны с большими размерами первых компьютеров и большими временными потерями в процессах передачи информации между ними. Как следствие, значительного увеличения мощности добиться не удавалось. Но совершенствовались технологии, уменьшались размеры компьютеров, снижалось их энергопотребление. И через некоторое время стало возможно создавать многомашинный комплекс как единую многопроцессорную вычислительную систему с приемлемыми производственными параметрами.

Количество процессоров в системах увеличивалось постепенно. Вообще говоря, в целях достижения максимальной производительности составлять программы для каждого процессора надо было бы индивидуально. Но для пользователя это и не удобно и не привычно. К тому же, необходимо было обеспечить преемственность использования обычных последовательных программ, которых накопилось в мире уже очень много, на системах с несколькими процессорами. Поэтому решение проблемы адаптации последовательных программ к таким системам взяли на себя компиляторы. Однако постепенно внутреннего параллелизма в системах становилось все больше и больше. И, наконец, его стало столь много, что наработанные технологии компилирования программ оказались не в состоянии образовывать оптимальный машинный код. Для его получения в случае многих процессоров компилятору приходится иметь дело с задачей составления оптимального расписания. Решается она перебором и требует, в общем случае, выполнения экспоненциального объема операций по отношению к числу процессоров. Пока число процессоров было невелико, компиляторы как-то справлялись с такой задачей. Но как только их стало очень много, развитие традиционных технологий компилирования зашло в тупик.

Быстрое развитие элементной базы привело к тому, что уже в начале 60-х годов прошлого столетия стали серийно выпускаться вычислительные системы, в которых насчитывалось порядка десятка процессоров, работающих параллельно. Создателям компиляторов все еще удавалось прикрывать пользователей от такого параллелизма, и язык общения оставался практически последовательным. В конце 70-х годов появились серийные вычислительные машины векторного типа, в которых ускорение достигалось за счет быстрого выполнения операций над векторами. Уровень внутреннего параллелизма в них был достаточно высок, хотя весьма специального вида. Создатели компиляторов снова попытались сделать язык программирования последовательным. И на этот раз потерпели неудачу. Формально все еще оставалась возможность пользоваться некоторым последовательным языком. Но заложенная в компилятор

технология автоматического выявления векторных конструкций из текста программ оказалась не эффективной. Поэтому, если пользователя не устраивала скорость работы откомпилированной программы, ему нужно было просматривать служебную информацию о работе компилятора и на основе её анализа самому находить узкие места компиляции и вручную перестраивать программу под векторные конструкции. О том, как именно это делать, конструктивных советов не предлагалось. На практике процедуру перестройки программ приходилось делать многократно.

По существу на этом закончился период, когда задачу, полностью описанную на последовательном языке, можно было более или менее эффективно решать на любой вычислительной технике. На этом же стало заканчиваться время создания классических последовательных компьютеров и началась эра параллельных компьютеров и больших вычислительных систем параллельной архитектуры. По сравнению с последовательными компьютерами в них все обстоит иначе. Рядовые их представители имеют десятки процессоров, а самые большие - десятки и даже сотни тысяч. Мощности параллельных компьютеров огромны и теоретически не ограничены. Практические скорости уже сегодня достигают сотен триллионов операций в секунду. Однако все эти преимущества имеют серьезную негативную сторону: в отличие от последовательных компьютеров использовать параллельные чрезвычайно трудно, интерфейс с ними оказывается совсем не дружественным, языки программирования перестали быть универсальными, от пользователя требуется много новой и трудно доступной информации о структуре алгоритмов и т.д.

На вычислительных системах **параллельной архитектуры** время решения задач *решающим образом* зависит от того, какова внутренняя структура алгоритма и в каком порядке выполняются его операции. Возможность ускоренной реализации алгоритма на параллельных системах достигается за счет того, что в них имеется достаточно большое число процессоров, которые могут *параллельно* выполнять операции алгоритма. Предположим для простоты, что все процессоры имеют одинаковую производительность и работают в синхронном режиме. Тогда общий коэффициент ускорения по сравнению с тем случаем, когда алгоритм реализуется на одном универсальном процессоре такой же производительности, оказывается примерно равным числу операций алгоритма, выполняемых в среднем на всех процессорах в каждый момент времени. Если параллельные вычислительные системы имеют десятки и сотни тысяч процессоров, то отсюда никак не следует, что при решении конкретных задач всегда можно и, тем более, достаточно легко получить ускорение счета такого же порядка.

На любой вычислительной технике одновременно могут выполняться только *независимые* операции. Это означает следующее. Пусть снова все процессоры имеют одинаковую производительность и работают в синхронном режиме. Допустим, что в какой-то момент времени на каких-то процессорах выполняются какие-то операции алгоритма. Результат ни одной из них не только не может быть аргументом любой из выполняемых операций, но даже не может никаким косвенным образом оказывать влияние на их аргументы. Если рассмотреть процесс реализации алгоритма во времени, то в силу сказанного сам процесс на любой вычислительной системе, последовательной или параллельной, разделяет операции алгоритма на группы. Все операции каждой группы независимы и выполняются одновременно, а сами группы реализуются во времени последовательно одна за другой. Это неявно порождает некоторую специальную форму представления алгоритма, в которой фиксируются как группы операций, так и их последовательность. Называется *она параллельной формой алгоритма*. Ясно, что при наличии в алгоритме ветвлений или условных передач управления его параллельная форма может зависеть от значений входных данных.

Параллельную форму алгоритма можно ввести и как эквивалентный математический объект, не зависящий от вычислительных систем. Зафиксируем входные данные и разделим все операции алгоритма на группы. Назовем их *ярусами* и пусть они обладают следующими свойствами. Во-первых, в каждом ярусе находятся только независимые операции. И, во-вторых, существует такая последовательная нумерация ярусов, что каждая операция из любого яруса использует в качестве аргументов либо результаты выполнения операций из ярусов с меньшими номерами, либо входные данные алгоритма. Ясно, что все операции, находящиеся в ярусе с наименьшим номером, всегда используют в качестве аргументов только входные данные. Будем считать в дальнейшем, что нумерация ярусов всегда осуществляется с помощью натуральных чисел подряд, начиная с 1.

Число операций в ярусе принято называть *шириной* яруса, число ярусов в параллельной форме - *высотой* параллельной формы. Очевидно, что ярусы математической параллельной формы являются не чем иным как теми самыми группами, о которых говорилось выше. При одних и тех же значениях входных данных между математическими параллельными формами алгоритма и реализациями того же алгоритма на конкретных или гипотетических вычислительных системах с одним или несколькими процессорами

существует взаимно однозначное соответствие. Если какая-то параллельная форма отражает реализацию алгоритма на некоторой вычислительной системе, то ширина ярусов говорит о числе используемых в каждый момент времени независимых устройств, а высота - о времени реализации алгоритма.

Каждый алгоритм при фиксированных входных данных в общем случае имеет много параллельных форм. Формы, в которых все ярусы имеют ширину, равную 1, существуют всегда. Все они отражают последовательные вычисления и имеют максимально большие высоты, равные числу выполняемых алгоритмом операций. Даже таких параллельных форм алгоритм может иметь несколько, если он допускает различные эквивалентные реализации. Наибольший интерес представляют параллельные формы *минимальной* высоты, так как именно они показывают, насколько быстро может быть реализован алгоритм, по крайней мере, теоретически. По этой причине минимальная высота всех параллельных форм алгоритма называется *высотой алгоритма*. Существует параллельная форма, в которой каждая операция из яруса с номером k, $k > 1$, получает в качестве одного из аргументов результат выполнения некоторой операции из (k-1)-го яруса. Такая параллельная форма называется *канонической*. Для любого алгоритма при заданных входных данных каноническая форма всегда существует, единственна и имеет минимальную высоту. Кроме этого, в канонической параллельной форме, как и в любой другой форме минимальной высоты, ярусы в среднем имеют максимально возможную ширину. Таким образом, решая любую задачу на любой вычислительной системе с развитым параллелизмом на уровне функциональных устройств, пользователь *неизбежно*, явно или неявно, соприкасается с параллельной формой реализуемого алгоритма. И это происходит даже тогда, когда он ничего не знает обо всех этих понятиях. Если не сам пользователь или разработчик программы, то кто-то другой или что-то другое, например, компилятор, операционная система, какая-нибудь сервисная программа, а скорее всего все они вместе, закладывают в вычислительную систему некоторую программу действий, что и порождает соответствующую им параллельную форму.

А теперь вспомним, что достигаемое ускорение в среднем пропорционально числу операций, выполняемых в каждый момент времени. Если оно равно общему числу имеющихся процессоров, то данный алгоритм при заданных входных данных реализуется на используемой вычислительной системе эффективно. В этом случае возможный ресурс ускорения использован *полностью* и более быстрого счета достичь на выбранной системе *невозможно* ни практически, ни теоретически. Но если ускорение значительно меньше числа устройств?

Предположим, что оно существенно меньше средней ширины ярусов канонической параллельной формы реализуемого алгоритма. Это означает, что *не очень удачно* выбрана схема реализации алгоритма. Изменив ее, можно попытаться более полно использовать имеющийся потенциал параллелизма в алгоритме. Заметим, что в практике общения с параллельными компьютерами именно эта ситуация возникает наиболее часто. Если же ускорение равно средней ширине ярусов канонической параллельной формы, то весь потенциал параллелизма в алгоритме выбран полностью. В данном случае никаким изменением схемы счета нельзя использовать большее число устройств системы и, следовательно, нельзя добиться большего ускорения. И, наконец, вполне возможно, что даже при использовании всех процессоров реальное ускорение не соответствует ожидаемому. Это означает, что при реализации алгоритма приходится осуществлять какие-то передачи данных, требующие длительного времени. Для того чтобы теперь понять причины замедления, необходимо особенно тщательно изучить *структуру* алгоритма и/или вычислительной системы. В практике общения с параллельными компьютерами эта ситуация также возникает достаточно часто.

Таким образом, как только возникает необходимость решать какие-то вопросы, связанные с анализом ускорения при решении задачи на вычислительной системе параллельной архитектуры, так обязательно требуется получить какие-то сведения относительно структуры алгоритма на уровне связей между отдельными операциями. Более того, чаще всего эти сведения приходится сопоставлять со сведениями об архитектуре вычислительной системы. Отсутствие нужных сведений о структуре алгоритмов не могло остановить развитие собственно вычислительной техники. Стали создаваться новые языки и системы программирования, в огромном количестве и самого различного типа: от расширения последовательных языков параллельными конструкциями до создания на макроуровне параллельных языков типа автокода.

Во всех новых языках и системах программирования стали вводиться конструкции, позволяющие *описывать* такие множества. А вот *ответственность* за поиск в алгоритмах этих множеств и их описание соответствующими конструкциями языка была *возложена на разработчиков программ*. Как следствие, на них же перекладывалась и ответственность за эффективность функционирования создаваемого ими программного продукта. Конечно, в таких условиях даже не шла речь о какой-либо преемственности

программ. Главными проблемами пользователей стали нахождение многочисленных и трудно добываемых характеристик решаемых задач и организация параллельных вычислительных процессов. Вычислительным сообществом это рассматривается как неизбежная плата за возможность быстро решать задачи. Но во всем сообществе расплачиваются, главным образом, пользователи, постоянно переписывая свои программы.

Невнимание со стороны математиков к развитию вычислительной техники привело к серьезному разрыву между имеющимися знаниями в области алгоритмов и теми знаниями, которые были необходимы для быстрого решения задач на новейшей вычислительной технике. Образовавшийся разрыв сказывается до сих пор, и именно он лежит в основе многих трудностей практического освоения современных вычислительных систем параллельной архитектуры.

Перспектива внедрения в практику параллельных вычислительных систем потребовала разработки математической концепции построения *параллельных алгоритмов*, т.е. алгоритмов, специально приспособленных для реализации на подобных системах. Такая концепция необходима для того, чтобы научиться понимать, как следует конструировать параллельные алгоритмы, что можно ожидать от них в перспективе и какие подводные камни могут встретиться на этом пути. Концепция начала активно развиваться в конце 50-х годов 20-го века и получила название концепции *неограниченного параллелизма*.

Истоки этого названия связаны с выбором для нее абстрактной модели параллельной вычислительной системы. Поскольку концепция разрабатывалась для проведения математических исследований, то в требованиях к модели могло присутствовать самое минимальное число технических параметров. Тем более что в то время о структуре параллельных вычислительных систем и путях их совершенствования было вообще мало что известно. Лишь быстрое развитие элементной базы подсказывало, что число процессоров в системе вскоре может стать очень большим. Но что-нибудь другое спрогнозировать было трудно. Поэтому явно или неявно в модели остались только следующие предположения. Число процессоров может быть сколь угодно большим, все они работают в синхронном режиме и за единицу времени выполняют абсолютно точно любую операцию из заданного множества. Процессоры имеют общую память. Все вспомогательные операции, взаимодействие с памятью, управление компьютером и любые передачи информации осуществляются мгновенно и без конфликтов. Входные данные перед началом вычислений записаны в память. Каждый процессор считывает свои операнды из памяти и после выполнения операции записывает результат в память. После окончания вычислительного процесса все результаты остаются в памяти.