**Алгоритмы. Оценка сложности алгоритмов**

Понятие “алгоритм” давно является привычным не только для математиков. Оно является концептуальной основой разнообразных процессов обработки информации. Возможность автоматизации таких процессов обеспечивается наличием соответствующих алгоритмов. С алгоритмами первое знакомство происходит в начальной школе при изучении арифметических действий с натуральными числами. В упрощенном понимании “алгоритм” – это то, что можно запрограммировать на ЭВМ. Понятие алгоритма в общем его виде принадлежит к числу основных первоначальных понятийматематики, не допускающих определения в терминах более простых понятий.

1. **«Алгоритм – точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату» - по ГОСТу**.
2. **Алгоритм** – последовательность действий, которую ИСПОЛНИТЕЛЬ должен выполнить, чтобы решить задачу. Исполнитель глуп, туп и бестолков. Он умеет выполнять только небольшое количество действий (например, складывать, вычитать, умножать, делить). Но он всегда выполняет одно и то же действие одинаково. Поэтому, при составлении алгоритма нужно учитывать глупость (простоту) исполнителя, т.е. действия, из которых составляется алгоритм, должны принадлежать множеству действий, которые умеет выполнять исполнитель.
3. **Алгоритм**, алгорифм, - точное предписание, которое задает вычислительный процесс (называемый в этом случае алгоритмическим), начинающийся с некоторого исходного данного (из некоторой совокупности возможных для данного алгоритма исходных данных) и направленный на получение полностью определяемого этим исходным алгоритмом результата.
4. **Алгоритм**- система правил, четко описывающая последовательность действий, которые необходимо выполнить для решения конкретной задачи
5. **Алгоритм** – это формально описанная вычислительная процедура, которая получает исходные данные ( **input**), называемые также входом алгоритма или его аргументом, и которая выдает результат вычислений на выход (**output**). Алгоритмы строятся для решения тех или иных вычислительных задач. Формулировка задачи описывает каким требованиям должно удовлетворять решение задачи, а алгоритм , находит объект, который этим требованиям удовлетворяет.

**"Уточнение" понятия алгоритма**. Как правило, для данного алгоритма можно выделить семь характеризующих его параметров:

1) совокупность возможных исходных данных,

2) совокупность возможных результатов,

3) совокупность возможных промежуточных результатов,

4) правило начала,

5) правило непосредственной переработки,

6) правило окончания,

7) правило извлечения результата**.**

Таким образом, уточнение понятия алгоритма связано с уточнением алфавита данных и формы их представления, памяти и размещения в ней данных, элементарных шагов алгоритма и механизма реализации алгоритма. Однако эти понятия сами нуждаются в уточнении. Ясно, что их словесные определения потребуют введения новых понятий, для которых в свою очередь, снова потребуются уточнения и т.д. Поэтому в теории алгоритмов принят другой подход, основанный на конкретной алгоритмической модели, в которой все сформулированные требования выполняются очевидным образом. При этом используемые алгоритмические модели универсальны, т.е. моделируют любые другие разумные алгоритмические модели, что позволяет снять возможное возражение против такого подхода: не приводит ли жесткая фиксация алгоритмической модели к потере общности формализации алгоритма ? Поэтому данные алгоритмические модели отождествляются с формальным понятием алгоритма.

Возникло важное направление в теории алгоритмов - сложность алгоритмов и вычислений. Сами алгоритмы стали объектом точного исследования как и те объекты, для работы с которыми они предназначены. В этой области естественно выделяются задачи получения верхних и задачи получения нижних оценок сложности алгоритмов, и методы решения этих задач совершенно различны.

**Основные требования к алгоритмам.**

**1. Каждый алгоритм имеет дело с данными – входными, промежуточными, выходными. Для того, чтобы уточнить понятие данных, фиксируется конечный алфавит исходных символов ( цифры, буквы и т.п.) и указываются правила построения алгоритмических объектов. Типичным используемым средством является индуктивное построение. Например, определение идентификатора в PASCAL**: идентификатор – это либо буква, либо символ подчеркивания, к которому приписана справа либо буква, либо цифра. Слова конечной длины в конечных алфавитах - наиболее обычный тип алгоритмических данных, а число символов в слове - естественная мера объема данных. Другой случай алгоритмических объектов - формулы. Примером могут служить формулы алгебры предикатов и алгебры высказываний. В этом случае не каждое слово в алфавите будет формулой.

2. **Алгоритм для размещения данных требует памяти. Память обычно считается однородной и дискретной, т.е. она состоит из одинаковых ячеек, причем каждая ячейка может содержать один символ данных, что позволяет согласовать единицы измерения объема данных и памяти**.

3. **Алгоритм состоит из отдельных элементарных шагов, причем множество различных шагов, из которых составлен алгоритм, конечно. Типичный пример множества элементарных шагов – система команд ЭВМ.**

4. **Последовательность шагов алгоритма детерминированна, т.е. после каждого шага указывается, какой шаг следует выполнять дальше, либо указывается, когда следует работу алгоритма считать законченной**.

5. **Алгоритм должен обладать результативностью, т.е. останавливаться после конечного числа шагов ( зависящего от исходных данных ) с выдачей результата. Данное свойство иногда называют сходимостью алгоритма**.

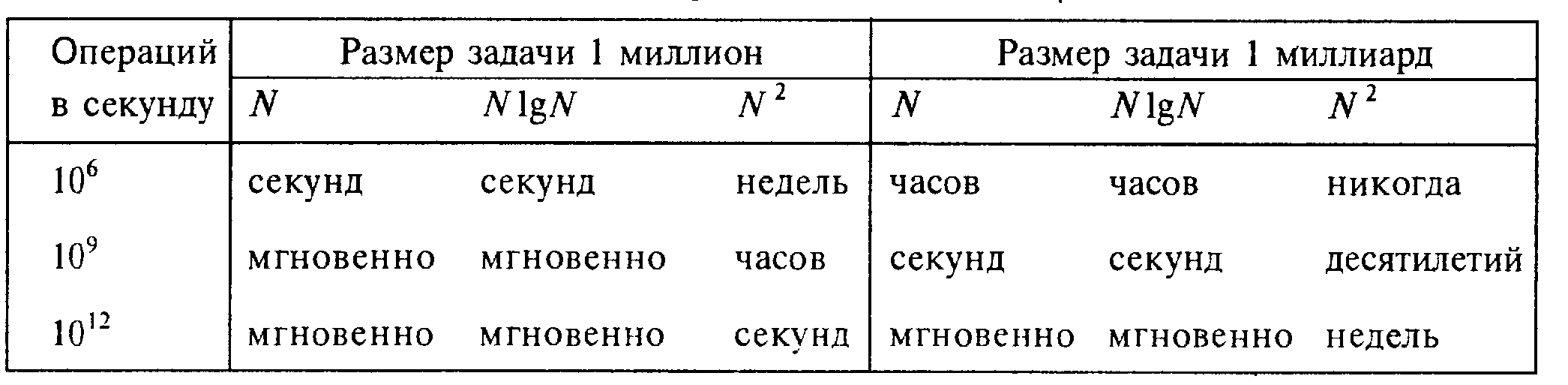
6. **Алгоритм предполагает наличие механизма реализации, который по описанию алгоритма порождает процесс вычисления на основе исходных данных. Предполагается, что описание алгоритма и механизм его реализации конечны**.

**Можно заметить аналогию с вычислительными машинами. Требование 1 соответствует цифровой природе ЭВМ, требование 2 - памяти ЭВМ, требование 3 – программе машины, требование 4 – ее логической природе, требования 5, 6 – вычислительному устройству и его возможностям.**

Алгоритм считают **правильным**, если на любом допустимом входе (для данной задачи) он заканчивает работу и выдает результат, удовлетворяющий требованиям задачи. В этом случае говорят, что алгоритм решает данную вычислительную задачу. **Неправильный** алгоритм может дать неправильный результат или вовсе не завершиться. Интуитивно сложность алгоритма – это мера зависимости между размером исходных данных и количеством операций, выполняемых алгоритмом в худшем случае, при наиболее «неудобных» исходных данных и количеством операций

**Анализ сложности алгоритма**

Для многих приложений нашим единственным шансом решить гигантскую задачу является использование эффективного алгоритма. Быстрый алгоритм позволяет решить задачу на медленной машине, но быстрая машина не может помочь, когда используется медленный алгоритм.



Существует ли некий "универсальный", наилучший алгоритм ? Вообще говоря, нет. Однако, имея приблизительные характеристики входных данных, можно подобрать метод, **работающий оптимальным образом. Для того, чтобы обоснованно сделать такой выбор, рассмотрим параметры, по которым будет производиться оценка алгоритмов**.

1. **Время выполнения-** основной параметр, характеризующий быстродействие алгоритма.
2. **Память** - ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.
3. **Устойчивость –** на пример устойчивая сортировка не меняет взаимного расположения равных элементов.
4. **Естественность поведения** - эффективность метода при обработке на пример уже отсортированных, или частично отсортированных данных. Алгоритм ведет себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Еще одним важным свойством алгоритма является его сфера применения. Здесь основных позиций две:

* внутренние сортировки работают с данным в оперативной памяти с произвольным

доступом;

* внешние сортировки упорядочивают информацию, расположенную на внешних носителях.

Это накладывает некоторые дополнительные ограничения на алгоритм:

* + доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим
  + объем данных не позволяет им разместиться в ОЗУ

Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью

**Анализ трудоёмкости алгоритмов**

**Анализ — это ключ к пониманию алгоритмов в степени, достаточной для их эффективного применения при решении практических задач**

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. Ниже перечислены основные оценки сложности.

Математическая запись, позволяющая отбрасывать подробности при анализе алгоритмов, называется **О-нотацией**.

В общем случае единственным методом определения быстродействия программного кода является оценка времени ее выполнения. Это справедливо в отношении любых алгоритмов. В вычислительной технике есть короткая и удобная схема –О-нотация (**big-Oh notation**). В этой нотации используется специальная математическая функция от n( количества элементов), которой пропорционально быстродействие алгоритм говорят ,что алгоритм принадлежит к классу O(f(n)), где f(n) –некоторая функция от n. Исследования показали, что последовательный поиск принадлежит к классу O(f(n)), а бинарный поиск к классу O(log(n). Для положительных чисел n log(n)<n поэтому бинарный поиск быстрее ,чем последовательный.

О-нотация проста и удобна. Предположим ,что экспериментальным путем показано, что алгоритм Х принадлежит к классу O(n2+n) , т.е. его быстродействие пропорционально (n2+n). А это означает, что можно найти такую константу k для которой

Быстродействие=k\*(n2+n))

Т.е. умножение (n2+n) на константу не оказывает никакого влияния на смысл нотации. Например O(3\*f(n)) эквивалентно O(f(n)). Если величина n при тестировании алгоритма Х достаточно велика, то влияние « +n» поглощается «n2». Т.е. при больших значениях n алгоритм O(n2+n) эквивалентен алгоритму O(n2). О-нотация подчиняется простым арифметическим правилам. Предположим, что есть алгоритм, который выполняет несколько различных задач. 1-ая задача принадлежит классу O(n), вторая - O(n2), а третья к классу O(log(n)). Необходимо определить быстродействие всего алгоритма в целом. Очевидно ,что ответом будет O(n2), поскольку к этому классу принадлежит доминантная часть алгоритма. Отсюда следует сделать вывод ,что значения О большого являются репрезентативными для больших значений, для маленьких значений О-нотация не имеет смысла, а на общий результат имеет влияние другие члены нотации. Допустим проводилось тестирование двух алгоритмов ,на основе статистических данных получены результаты:

Быстродействие 1-го алгоритма =k1\*(n+100000)

Быстродействие 2-го алгоритма =k2\*n2

Пусть константы сравнимы по величине. Если следовать О-нотации, то лучшим будет первый алгоритм, т.к. он принадлежит классу O(n), тем не менее ,если n не превышает 100, более эффективным окажется второй алгоритм. Таким образом алгоритм нужно выбирать и с учетом его назначения

Если алгоритм выполняется за фиксированное время, не зависящее от размера задачи, говорят, что его сложность равна O(1).

Это определение обобщается в случае, если время выполнения существенно зависит от нескольких параметров. Например, алгоритм, определяющий, входит ли множество m элементов в множество n элементов, может иметь, в зависимости от используемых структур данных, сложность O (m n) или O (m+n).

Практически время выполнения алгоритма может зависеть от значений данных. Так, время выполнения некоторых алгоритмов сортировки существенно сокращается, если первоначально эти данные были частично упорядочены. Чтобы учитывать это, сохраняя возможность анализировать алгоритм независимо от их данных, различают:

* *максимальную сложность*, определяемую значением Fmax(n) — время выполнения алгоритма, когда выбранный набор n данных порождает наиболее долгое время выполнения алгоритма;
* *среднюю сложность*, определяемую значениемFср(n) — средним временем выполнения алгоритма, примененного к n произвольным данным.

Эти понятия без труда распространяются на измерение сложности в единицах объема памяти: можно говорить о средней и максимальной пространственной сложности.

Самыми лучшими являются *линейные* алгоритмы, имеющие сложность порядка an=b. Они называются также алгоритмами порядка O(n) где n — размерность входных данных. Такие алгоритмы действительно существуют. Например, сложение двух чисел столбиком в случае, если одно из них состоит из n, а другое — из m цифр, требует не более max(n, m) сложений и не более max(n, m) запоминаний. Т.е. данный алгоритм имеет сложность порядка O(n+m). Разумеется, это выражение показывает только порядок величины — постоянные факторы в нем не учитываются.

Обобщение линейности дает нам **первый большой класс алгоритмов — *полиномиальных*.**

**Полиномиальным** (или алгоритмом полиномиальной временной сложности) называется алгоритм, у которого временная сложность есть O(p(n)), где p(n) — полином от n. Задачи, где для решения известен алгоритм, сложность которого составляет полином заданной, постоянной и не зависящей от размерности входной величины n степени, называют "хорошими" и относят их к классу P.

Экспоненциальной по природе считается задача сложностью не менее порядка xn, где x — константа или полином от n. Например, это задачи, в которых возможное число ответов уже экспоненциально. В частности, к ним относятся задачи, где требуется построить все подмножества заданного множества или все поддеревья заданного графа. Экспоненциальные задачи относят к классу E.

Соответственно, и алгоритмы, в оценку сложности которых n входит в показатель степени, относятся к ***экспоненциальным*.**

Необходимо отметить, что при небольших значениях n экспоненциальный алгоритм может быть даже менее сложным, чем полиномиальный. Тем не менее, различие между этими типами алгоритмов весьма велико и проявляется при больших значениях n.

Особую группу по значениям сложности, близким к полиномиальным, составляют алгоритмы, сложность которых является полиномиальной функцией от log n (поскольку log n растет медленнее, чем n).

Для большей убедительности и сравнения полиномиальных и экспоненциальных алгоритмов приведем таблицу, где единица времени — 1 мкс, а сложность совпадает с необходимым количеством единиц времени для обработки набора n данных:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Таблица 3.1. Сложность и время выполнения** | | | | | | |
| **Сложность** | **Размер задачи — n** | | | | | |
|  | **10** | **20** | **30** | **40** | **50** | **60** |
| **n** | **0.00001 с** | **0.00002 с** | **0.00003 с** | **0.00004 с** | **0.00005 с** | **0.00006 с** |
| **n²** | **0.0001 с** | **0.0004 с** | **0.0009 с** | **0.0016 с** | **0.0025 с** | **0.0036 с** |
| **n³** | **0.001 с** | **0.008 с** | **0.027 с** | **0.064 с** | **0.125 с** | **0.216 с** |
| **n5** | **0.1 с** | **3.2 с** | **24.3 с** | **1.7 мин** | **5.2 мин** | **13.0 мин** |
| **2n** | **0.01 с** | **1.0 с** | **17.9 мин** | **12.7 дней** | **35.7 лет** | **366 веков** |
| **3n** | **0.59 с** | **58 мин** | **6.5 лет** | **3855 веков** | **2·108 веков** | **1.3·1013 веков** |

**Приведенная таблица иллюстрирует причины, по которым полиномиальные алгоритмы считаются более предпочтительным, чем экспоненциальные**

**Зачем нужна оценка сложности**

* 1. Нужно уметь оценить применимость алгоритма до его реального применения
  2. В рефакторинге на скорость нужно заменять алгоритмы на лучшие и быть в этом уверенными
  3. Нужно уметь понимать, когда лучше пользоваться встроенными функциями, а когда нет

1. Зависимость от начальных данных
   1. Оценка производится как бы «для наихудшего варианта»
   2. Сложность алгоритма определяется размерностью входных данных
2. Примеры оценки сложности
   1. Простой поиск по массиву
   2. Простой поиск по упорядоченному массиву
   3. Бинарный поиск по упорядоченному массиву
   4. Сортировка выбором наименьшего значения

Поиск подстроки в строке

## Общие правила анализа алгоритмов

1. Время выполнения операторов присваивания, чтения и записи обычно имеет порядок О(1).
2. Время выполнения последовательности операторов определяется с помощью правила сумм. Поэтому степень роста времени выполнения последовательности операторов без определения констант пропорциональности совпадает с наибольшим временем выполнения оператора в данной последовательности.
3. Время выполнения [условных операторов](http://ru.wikipedia.org/wiki/%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80) состоит из времени выполнения условно исполняемых операторов и времени вычисления самого логического выражения. Время вычисления логического выражения обычно имеет порядок О(1). Время для всей конструкции if-then-else состоит из времени вычисления логического выражения и наибольшего из времени, необходимого для выполнения операторов, исполняемых при значении логического выражения true (*истина*) и при значении false (*ложь*).
4. Время выполнения [цикла](http://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB) является суммой времени всех исполняемых [итераций](http://ru.wikipedia.org/w/index.php?title=%D0%98%D1%82%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F&action=edit) цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла (обычно последнее имеет порядок О(1)). Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количества выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла. Время выполнения каждого цикла, если в программе их несколько, должно определяться отдельно.
5. Для программ, содержащих несколько процедур (среди которых нет рекурсивных), можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения процедур, начиная с той, которая не имеет вызовов других процедур.
6. При анализе времени выполнения программ мы неявно предполагали, что все ветвления в ходе выполнения процедур осуществлялись с помощью условных операторов и операторов циклов. Мы останавливаемся на этом факте, так как определяли время выполнения больших групп операторов путем применения правила сумм к этом группам. Однако операторы безусловного перехода (такие как goto) могут порождать более сложную логическую групповую структуру. В принципе, операторы безусловного перехода можно исключить из программы. Но, к сожалению, язык [Pascal](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%81%D0%BA%D0%B0%D0%BB%D1%8C) не имеет операторов досрочного прекращения циклов, поэтому операторы перехода по необходимости часто используются в подобных ситуациях. Если же переход осуществляется к ранее выполненным операторам, то в этом случае вообще нельзя игнорировать оператор безусловного перехода, поскольку такой оператор создает петлю в выполнении программы, что приводит в нарастанию времени выполнения всей программы. Нам не хотелось бы, чтобы у читателя сложилось мнение о невозможности анализа времени выполнения программ, использующих операторы безусловного перехода, создающих петли. Если программные петли имеют простую структуру, т.е. о любой паре петель можно сказать, что они или не имеют общих элементов, или одна петля вложена в другую, то в этом случае можно применить методы анализа времени выполнения, описанные в данном разделе. (Конечно, бремя ответственности за правильное определение структуры петли ложится на того, кто проводит анализ.)
7. **Виды анализа**
8. • Наихудший случай: T(N) - максимальное время для любых входных данных размера N.
9. • Средний случай: T(N) - ожидаемое время для любых входных данных размера N. (Необходимо допущение???)
10. • Наилучший случай - надувательство.