Алгоритмы и структуры данных

Введение

# Введение

Без понимания структур данных и алгоритмов невозможно создать серь­езный программный продукт. Знание теории структур, методов представления данных на логическом и физическом уровнях, а также эффективных алгорит­мов обработки различных структур данных, умение проводить сравнительный анализ и оценку эффективности применяемых алгоритмов совершенно необхо­димы для высококвалифицированных программистов.

Решение любой задачи сводится к обработке множества данных, которые представляют собой абстракцию некоторой части реального мира. Для решения конкретной задачи с использованием вычислительной техники необходимо, во-первых, выбрать подходящий уровень абстрагирования, т.е. определить множе­ство данных, представляющих реальную ситуацию и относящуюся к реальной задаче. И, во-вторых, необходимо выбрать способ представления этих данных с учетом возможностей языка программирования и вычислительной техники.

При выборе представления данных необходимо учитывать, какие алго­ритмы обработки будут к ним применяться. В то же время выбор алгоритма, в свою очередь, существенно зависит от строения данных. В связи с этим струк­тура данных и структура программы связаны между собой, причем данные предшествуют алгоритму, так как прежде чем выполнять какие-либо операции, нужно определить объекты, к которым они применяются.

Решение одной и той же задачи можно получить с помощью различных алгоритмов, поэтому при разработке программных продуктов важное место за­нимает оценка эффективности применяемых алгоритмов. Это требует умения проводить сравнительный анализ алгоритмов и правильно выбирать из них наиболее эффективные алгоритмы для решения поставленной задачи.

Таким образом, знание структур данных, методов представления данных на логическом и физическом уровнях, а также эффективных алгоритмов обра­ботки различных структур данных совершенно необходимо для высококвали­фицированных прикладных и системных программистов, разработчиков ин­формационных систем различной степени сложности, автоматизированных систем управления, трансляторов, операционных систем, систем управления базами данных, систем искусственного интеллекта и др.

# 1.1 Толковый словарь исходных терминов

*Алгоритм* – это набор инструкций для выполнения некоторой задачи.

*Алгоритм* – это конечная последовательность предписании, исполнение ко­торых позволяет получить решение некоторой задачи.

*Сортировка* – это упорядочение совокупности объектов в соответствии с заданным отношением порядка.

*Список (списковая структура)* – это набор элементов данных, связанных друг с другом таким образом, что каждый элемент содержит адрес последующего элемента.

*Структура данных* - множество элементов данных, упорядоченных одним нз принятых способов.

*Тип данных* — множество значений, которые могут принимать переменная или выражение в совокупности с множеством операций, допустимыми над этими значениями.

*Файл* - набор данных на логическом уровне рассмотрения.

# 1.2 Понятие структур данных и алгоритмов

Структуры данных и алгоритмы служат теми материалами, из которых строятся программы. Более того, сам компьютер состоит из структур данных и алгоритмов. Встроенные структуры данных представлены теми регистрами и словами памяти, где хранятся двоичные величины. Заложенные в конструкцию аппаратуры алгоритмы - это воплощенные в электронных логических цепях жесткие правила, по которым занесенные в память данные интерпретируются как команды, подлежащие исполнению. Поэтому в основе работы всякого компьютера лежит умение оперировать только с одним видом данных - с отдельными битами, или двоичными цифрами. Работает же с этими данными компьютер только в соответствии с неизменным набором алгоритмов, которые определяются системой команд центрального процессора.

Задачи, которые решаются с помощью компьютера, редко выражаются на языке битов. Как правило, данные имеют форму чисел, литер, текстов, символов и более сложных структур типа последовательностей, списков и деревьев. Еще разнообразнее алгоритмы, применяемые для решения различных задач; фактически алгоритмов не меньше чем вычислительных задач.

Для точного описания абстрактных структур данных и алгоритмов программ используются такие системы формальных обозначений, называемые языками программирования, в которых смысл всякого предложения определяется точно и однозначно. Среди средств, представляемых почти всеми языками программирования, имеется возможность ссылаться на элемент данных, пользуясь присвоенным ему именем (идентификатором). Одни именованные величины являются константами, которые сохраняют постоянное значение в той части программы, где они определены, другие - переменными, которым с помощью оператора в программе может быть присвоено любое новое значение. Но до тех пор, пока программа не начала выполняться, их значение не определено.

Имя константы или переменной помогает программисту, но компьютеру оно ни о чем не говорит. Компилятор же, транслирующим текст программы в двоичный код, связывает каждый идентификатор с определенным адресом памяти. Но для того чтобы компилятор смог это выполнить, нужно сообщить о «типе» каждой именованной величины. Человек, решающий какую-нибудь задачу «вручную», обладает интуитивной способностью быстро разобраться в тинах данных и тех операциях, которые для каждого типа справедливы. Так, например, нельзя извлечь квадратный корень из слова или написать число с заглавной буквы. Одна из причин, позволяющих легко провести такое распознавание, состоит в том, что слова, числа и другие обозначения выглядят по- разному. Однако, для компьютера все типы данных сводятся в конечном счете к последовательности битов, поэтому различие в типах следует делать явным.

Типы данных, принятые в языках программирования, включают натуральные и целые числа, вещественные (действительные) числа (в виде приближенных десятичных дробей), литеры, строки и т.п.

В некоторых языках программирования тип каждой константы или переменной определяется компилятором по записи присваиваемого значения: наличие десятичной точки, например, может служить признаком вещественного числа. В других языках требуется, чтобы программист явно задал тип каждой переменной, и это дает одно важное преимущество. Хотя при выполнении программы значение переменной может многократно меняться, тип же меняться не должен никогда; это значит, что компилятор может проверить операции, выполняемые над этой переменной, и убедиться в том, что все они согласуются с описанием типа переменной. Такая проверка может быть проведена путем анализа всего текста программы, и в этом случае она охватит все возможные действия, определяемые данной программой.

В зависимости от назначения языка программирования защита типов, осуществляемая на этапе компиляции, может быть более или менее жесткой. Так, например, язык С++, изначально являвшийся прежде всего инструментом для иллюстрирования структур данных и алгоритмов, сохраняет от своего первоначального назначения весьма строгую защиту типов. С++-компилятор в большинстве случаев расценивает смешение в одном выражении данных разных типов или применение к типу данных несвойственных ему операций как фатальную ошибку. Напротив, язык С, предназначенный, прежде всего для системного программирования, является языком с весьма слабой зашитой типов. С-компиляторы в таких случаях лишь выдают предупреждения. Отсутствие жесткой защиты типов дает системному программисту, разрабатывающему программу на языке С, дополнительные возможности, но такой программист сам отвечает за правильность своих действий.

Структура данных относится, по существу, к «пространственным» понятиям: ее можно свести к схеме организации информации в памяти компьютера. Алгоритм же является соответствующим процедурным элементом в структуре программы - он служит рецептом расчета.

Первые алгоритмы были придуманы для решения численных задач типа умножения чисел, нахождения наибольшего общего делителя, вычисления тригонометрических функций и других. Сегодня в равной степени важны и не численные алгоритмы; они разработаны для таких задач, как, например, поиск в тексте заданного слова, планирование событий, сортировка данных в указанном порядке и т.п. Нечисленные алгоритмы оперируют с данными, которые не обязательно являются числами; более того, не нужны никакие глубокие математические понятия, чтобы их конструировать или понимать. Из этого, однако, вовсе не следует, что в изучении таких алгоритмов математике нет места; напротив, точные, математические методы необходимы при поиске наилучших решений нечисленных задач при доказательстве правильности этих решений.

Структуры данных, применяемые в алгоритмах, могут быть чрезвычайно сложными. В результате выбор правильного представления данных часто служит ключом к удачному программированию и может в большей степени сказываться на производительности программы, чем детали используемого алгоритма. Вряд ли когда-нибудь появится общая теория выбора структур данных. Самое лучшее, что можно сделать, - это разобраться во всех базовых «кирпичиках» и в собранных из них структурах. Способность приложить эти знания к конструированию больших систем - это прежде всего дело инженерного мастерства и практики.

# 1.3 Информация и ее представление в памяти

Начиная изучение структур данных или информационных структур, необходимо ясно установить, что понимается под информацией, как информация передается и как она физически размещается в памяти вычислительной машин.

В цифровых вычислительных машинах можно выделить три основных ви­да запоминающих устройств: сверхоперативная, оперативная и внешняя па­мять. Обычно сверхоперативная память строится на регистрах. Регистры ис­пользуются для временного хранения и преобразования информации.

Некоторые из наиболее важных **регистров** содержатся в центральном про­цессоре компьютера. Сложение, вычитание, умножение и деление зане­сенной в аккумуляторы информации выполняется с помощью очень сложных логических схем. Кроме того, с целью проверки необходимости изменения нормальной последовательности передачи управления, в аккумуляторах могут анализироваться отдельные биты. Кроме запоминания операндов и результатов арифметических операций, регистры используются также для временного хра­нения команд программы и управляющей информации о номере следующей выполняемой команды.

**Оперативная память** предназначена для запоминания более постоянной по своей природе информации. Важнейшим свойством оперативной памяти явля­ется **адресуемость**. Это означает, что каждая ячейка памяти имеет свой иденти­фикатор, однозначно идентифицирующий ее в общем массиве ячеек памяти. Этот идентификатор называется адресом. Адреса ячеек являются операндами тех машинных команд, которые обращаются к оперативной памяти. В подав­ляющем большинстве современных вычислительных систем единицей адреса­ции является **байт** - ячейка, состоящая из 8 двоичных разрядов. Определенная ячейка оперативной памяти или множество ячеек могут быть связаны с кон­кретной переменной в программе. Однако для выполнения арифметических вычислений, в которых участвует переменная, необходимо, чтобы до начала вычислений значение переменной было перенесено из ячейки памяти в регистр. Если результат вычисления должен быть присвоен переменной, то результи­рующая величина снова должна быть перенесена из соответствующего регистра в связанную с этой переменной ячейку оперативной памяти.

Во время выполнения программы ее команды и данные в основном разме­щаются в ячейках оперативной памяти. Полное множество элементов опера­тивной памяти часто называют основной памятью.

**Внешняя память** служит прежде всего для долговременного хранения дан­ных. Характерным для данных на внешней памяти является то, что они могут сохраняться там даже после завершения создавшей их программы и могут быть впоследствии многократно использованы той же программой при повторных се запусках или другими программами. Внешняя память используется также для хранения самих программ, когда они не выполняются. Поскольку стоимость внешней памяти значительно меньше оперативной, а объем значительно боль­ше, то еще одно назначение внешней памяти - временное хранение тех кодов и данных выполняемой программы, которые не используются на данном этапе ее выполнения. Активные коды выполняемой программы и обрабатываемые ею на данном этапе данные должны обязательно быть размещены в оперативной па­мяти, так как прямой обмен между внешней памятью и операционными устрой­ствами (регистрами) невозможен.

Как хранилище данных, внешняя память обладает в основном теми же свойствами, что и оперативная, в том числе и свойством **адресуемости**. Поэто­му в принципе структуры данных на внешней памяти могут быть теми же, что и в оперативной, и алгоритмы их обработки могут быть одинаковыми. Но внеш­няя память имеет совершенно иную физическую природу, для нее применяются (на физическом уровне) иные методы доступа, и этот доступ имеет другие вре­менные характеристики. Это приводит к тому, что структуры и алгоритмы, эф­фективные для оперативной памяти, не оказываются таковыми для внешней памяти.

# 1.4 Классификация структур данных

Теперь можно дать более конкретное определение данного на машинном уровне представления информации.

Независимо от содержания и сложности любые данные в памяти ЭВМ представляются последовательностью двоичных разрядов, или битов, а их значениями являются соответствующие двоичные числа. Данные, рассматриваемые в виде последовательности битов, имеют очень простую организацию или, другими словами, слабо структурированы. Для человека описывать и исследовать сколько-нибудь сложные данные в терминах последовательностей битов весьма неудобно. Более крупные и содержательные, нежели бит, «строительные блоки» для организации произвольных данных получаются на основе понятия «структуры данного».

Под структурой данных в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур данных, направления которой соответствуют различным аспектам их рассмотрения. Прежде чем приступать к изучению конкретных структур данных, дадим их общую классификацию.

Структуры данных

Простые базовые структуры

Статические структуры

Полустатические структуры

Динамические структуры

Файловые структуры (файлы)

Числовые

Символьные

Логические

Перечисление

Интервал

Указатели

Вектор

Массивы

Множества

Записи

Таблицы

Стеки

Очереди

Деки

Строки

Последовательные

Прямого доступа

Комбинированного

доступа

Организованные

разделами

Графы

Деревья

Линейные связные списки

Разветвлённые связные списки

*Рисунок 1. Классификация структур данных*

Информация по каждому типу однозначно определяет:

1. структуру хранения данных указанного типа, т.е. выделение памяти и представление данных в ней, с одной стороны, и интерпретирование двоичного представления, с другой;
2. множество допустимых значений, которые может иметь тот или иной объект описываемого типа;
3. множество допустимых операций, которые применимы к объекту опи­сываемого типа.

В последующем мы будем рассматривать структуры данных и соответствующие им типы данных.

# 1.5 Операции над структурами данных

**Над любыми структурами данных** могут выполняться **четыре** общие операции:

1. создание,
2. уничтожение,
3. выбор (доступ),
4. обновление.

Операция **создания** заключается в выделении памяти для структуры данных. Память может выделяться в процессе выполнения программы или на этапе компиляции. В ряде языков (например, в С) для структурированных данных, конструируемых программистом, операция создания включает в себя также установку начальных значений параметров, создаваемой структуры.

Например, в С++ (int I) в результате описания типа будет выделена память для соответствующих переменных. Для структур данных, объявленных в программе, память выделяется автоматически средствами систем программирования либо на этапе компиляции, либо при активизации процедурного блока, в котором объявляются соответствующие переменные. Программист может и сам выделять память для структур данных, используя имеющиеся в системе программирования процедуры/функцни выделения/освобождения памяти. В объектно-ориентированных языках программирования при разработке нового объекта для него должны быть определены процедуры создания и уничтожения.

Главное заключается в том, что независимо от используемого языка программирования, имеющиеся в программе структуры данных не появляются «из ничего», а явно или неявно объявляются операторами создания структур. В результате этого всем экземплярам структур в программе выделяется память для их размещения.

Операция **уничтожения** структур данных противоположна по своему действию операции создания. Некоторые языки, такие как BASIC, не дают возможности программисту уничтожать созданные структуры данных. В языках С++ структуры данных, имеющиеся внутри блока, уничтожаются в процессе выполнения программы при выходе из этого блока. Операция уничтожения помогает эффективно использовать память.

Операция **выбора** используется программистами для доступа к данным внутри самой структуры. Форма операции доступа зависит от типа структуры данных, к которой осуществляется обращение. Метод доступа - один из наиболее важных свойств структур, особенно в связи с тем, что это свойство имеет непосредственное отношение к выбору конкретной структуры данных.

Операция **обновления** позволяет изменить значения данных в структуре данных. Примером операции обновления является операция присваивания, или, более сложная форма - передача параметров.

**Вышеуказанные четыре операции обязательны для всех структур и типов данных**. Помимо этих общих операций для каждой структуры данных могут быть определены операции специфические, работающие только с данными данного типа (данной структуры). Специфические операции будем рассматривать при рассмотрении каждой конкретной структуры данных.

# 1.6 Структурность данных и технология

# программирования

Большинство авторов публикаций, посвященных структурам и организации данных, делают основной акцент на том, что знание структуры данных позволяет организовать их хранение и обработку максимально эффективным образом - с точки зрения минимизации затрат как памяти, так и процессорного времени. Другим не менее, а может быть, и более важным преимуществом, которое обеспечивается структурным подходом к данным, является возможность структурирования сложного программного изделия. Современные промышленно выпускаемые программные пакеты — изделия чрезвычайно сложные, объем которых исчисляется тысячами и миллионами строк кода, а трудоемкость разработки — сотнями человек. Естественно, что разработать такое программное изделие «все сразу» невозможно, оно должно быть представлено в виде какой- то структуры - составных частей и связей между ними. Правильное структурирование изделия даст возможность на каждом этапе разработки сосредоточить внимание разработчика на одной обозримой части изделия или поручить реализацию разных его частей разным исполнителям.

При структурировании больших программных изделий возможно применение подхода, основанного на структуризации алгоритмов и известного, как «нисходящее» проектирование или «программирование сверху вниз», или подхода, основанного на структуризации данных и известного, как «восходящее» проектирование или «программирование снизу вверх».

В первом случае структурируют прежде всего действия, которые должна выполнять программа. Большую и сложную задачу, стоящую перед проектируемым программным изделием, представляют в виде нескольких подзадач меньшего объема. Таким образом, модуль самого верхнего уровня, отвечающий за решение всей задачи в целом, получается достаточно простым и обеспечивает только последовательность обращений к модулям, реализующим подзадачи. На первом этапе проектирования модули подзадач выполняются в виде «заглушек». Затем каждая подзадача в свою очередь подвергается декомпозиции по тем же правилам. Процесс дробления на подзадачи продолжается до тех пор, пока на очередном уровне декомпозиции получают подзадачу, реализация которой будет вполне обозримой. В предельном случае декомпозиция может быть доведена до того, что подзадачи самого нижнего уровня могут быть решены элементарными инструментальными средствами (например, одним оператором выбранного языка программирования).

Другой подход к структуризации основывается на данных. Программисту, который хочет, чтобы его программа имела реальное применение в некоторой прикладной области, не следует забывать о том, что программирование - это обработка данных. В программах можно изобретать сколь угодно замысловатые и изощренные алгоритмы, но у реального программного изделия всегда есть Заказчик. У Заказчика сеть входные данные, и он хочет, чтобы по ним были получены выходные данные, а какими средствами это обеспечивается - его не интересует. Таким образом, задачей любого программного изделия является преобразование входных данных в выходные. Инструментальные средства программирования предоставляют набор базовых (простых, примитивных) типов данных и операции над ними. Интегрируя базовые типы, программист создает более сложные типы данных и определяет новые операции над сложными типами. Можно здесь провести аналогию со строительными работами: базовые типы - «кирпичики», из которых создаются сложные типы — «строительные блоки». Полученные на первом шаге композиции «строительные блоки» используются в качестве базового набора для следующего шага, результатом которого будут еще более сложные конструкции данных и еще более мощные операции над ними и т.д. В идеале последний шаг композиции даст типы данных, соответствующие входным и выходным данным задачи, а операции над этими типами реализуют в полном объеме задачу проекта.

Программисты, поверхностно понимающие структурное программирование, часто противопоставляют нисходящее проектирование восходящему, придерживаясь одного выбранного ими подхода. Реализация любого реального проекта всегда ведется встречными путями, причем, с постоянной коррекцией структур алгоритмов по результатам разработки структур данных и наоборот.

Еще одним чрезвычайно продуктивным технологическим приемом, связанным со структуризацией данных является инкапсуляция. Смысл се состоит в том, что сконструированный новый тип данных - 'Строительный блок» - оформляется таким образом, что его внутренняя структура становится недоступной для программиста - пользователя этого типа. Программист, использующий этот тип данных в своей программе (в модуле более высокого уровня), может оперировать с данными этого типа только через вызовы процедур, определенных для этого типа. Новый тип данных представляется для него в виде «черного ящика» для которого известны входы и выходы, но содержимое — неизвестно и недоступно.

Инкапсуляция чрезвычайно полезна и как средство преодоления сложности, и как средство защиты от ошибок. Первая цель достигается за счет того, что сложность внутренней структуры нового типа данных и алгоритмов выполнения операций над ним исключается из поля зрения программиста-пользователя. Вторая цель достигается тем, что возможности доступа пользователя ограничиваются лишь заведомо корректными входными точками, следовательно, снижается и вероятность ошибок.

Современные языки программирования блочного типа (PASCAL, С) обладают достаточно развитыми возможностями построения программ с модульной структурой и управления доступом модулей к данным и процедурам. Программист, оперирующий объектами, указывает в программе ЧТО нужно сделать с объектом, а не КАК это надо делать.

Технология баз данных развивалась параллельно с технологией языков программирования и не всегда согласованно с ней. Отчасти этим, а отчасти и объективными различиями в природе задач, решаемых системами управления базами данных (СУБД) и системами программирования, вызваны некоторые терминологические и понятийные различия в подходе к данным в этих двух сферах. Ключевым понятием в СУБД является понятие модели данных, в основном тождественное понятию логической структуры данных. Отметим, что физическая структура данных в СУБД не рассматривается вообще. Но сами СУБД являются программными пакетами, выполняющими отображение физической структуры в логическую (в модель данных). Для реализации этих пакетов используются те или иные системы программирования, разработчики СУБД, следовательно, имеют дело со структурами данных в терминах систем программирования. Для пользователя же внутренняя структура СУБД и физическая структура данных совершенно прозрачна; он имеет дело только с моделью данных и с другими понятиями логического уровня.

# 3. Целочисленное умножение

Когда вы учились в начальной школе, вы, скорее всего, изучали умножение двух чисел в столбик, что, по сути, является алгоритмом — четко сформулированным набором правил для преобразования входа (два числа) в выход (их произведение). Всегда важно понимать разницу между постановкой (описанием) решаемой задачи и описанием метода ее решения (то есть алгоритма этой задачи).

## Задача целочисленного умножения

В задаче целочисленного умножения входными данными являются два *n*-разрядных числа, обозначим их *x* и *y*. Разрядность (длина) *n* чисел *x* и *y* может быть любым положительным целым числом. Однако я призываю вас оперировать большими значениями *n*, в тысячах или более. (Представьте, что вы разрабатываете некое криптографическое приложение, ведь они манипулируют очень большими числами.) В задаче целочисленного умножения требуемым выходом является результат произведения *x* \* *y*.

**ЗАДАЧА. ЦЕЛОЧИСЛЕННОЕ УМНОЖЕНИЕ**

**Вход:** два *n*-значных неотрицательных целых числа, *x* и *y*.

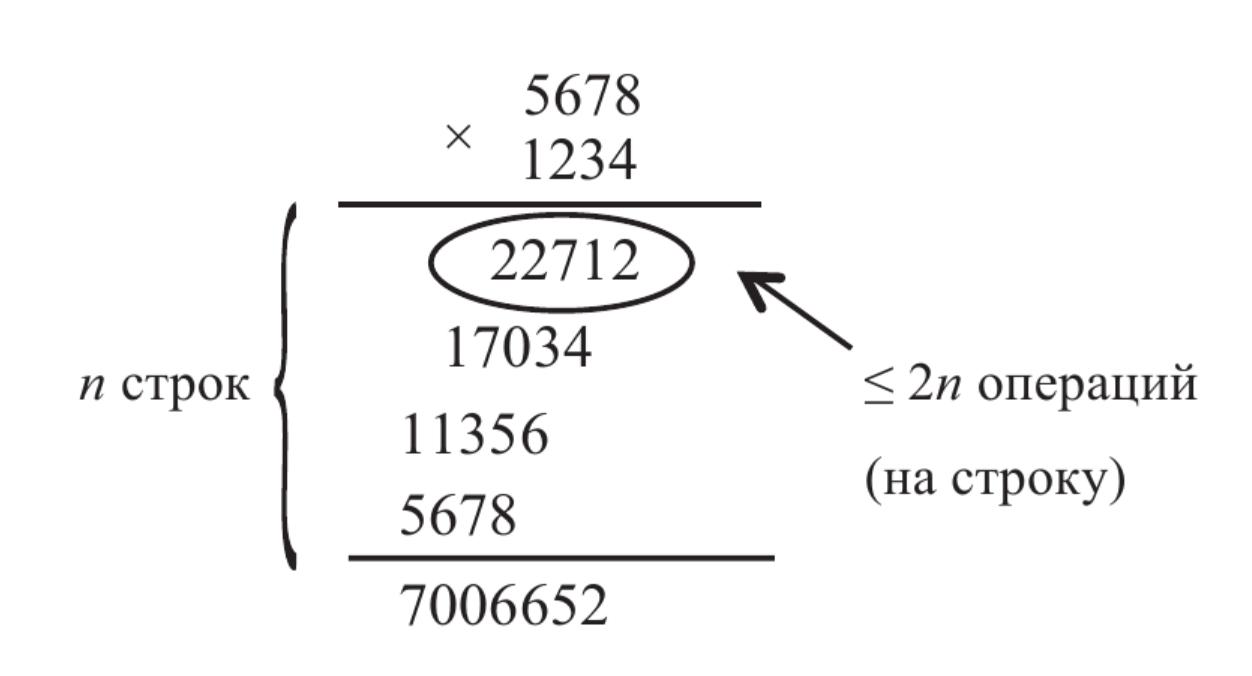
**Выход:** произведение *x* \* *y*.

## Алгоритм начальной школы

Точно определив вычислительную задачу, мы опишем алгоритм, который ее решает, — тот самый алгоритм, который вы изучали в начальной школе. Мы оценим производительность этого алгоритма числом «примитивных операций», которые он выполняет, в виде функции от количества знаков n в каждом входном числе. Пока же давайте представим примитивную операцию как любую из следующих: (i) сложение двух одноразрядных (n = 1) чисел; (ii) умножение двух одноразрядных чисел или (iii) добавление нуля к началу или концу числа.

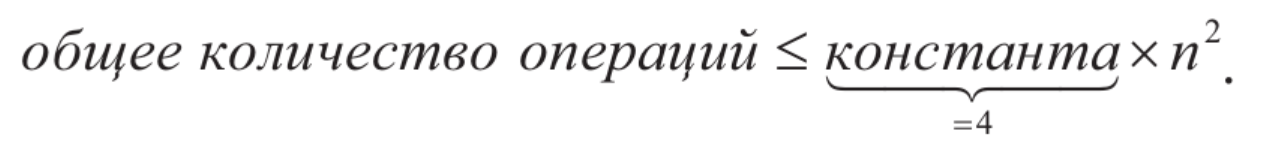
Чтобы освежить вашу память, рассмотрим конкретный пример умножения *x* = 5678 на *y* = 1234 (здесь *n* = 4) в столбик. Сначала алгоритм вычисляет «частичное произведение» первого числа и последней цифры второго числа: 5678 \* 4 = 22 712. Вычисление этого частичного произведения сводится к умножению каждой цифры первого числа на 4, записи младшего разряда результата, запоминанию («переносу» на следующий этап) старшего разряда и добавлению этих «переносов» (если они есть) на следующем умножении.

При вычислении следующего частичного произведения (5678 \* 3 = 17 034) мы делаем то же самое, сдвигая результат на один знак влево (фактически добавляя «0» в конце). И так далее для оставшихся двух частичных произведений. Заключительный шаг состоит в том, чтобы сложить все частичные произведения.

Тогда в третьем классе вы, вероятно, согласились, что этот алгоритм является правильным, имея в виду, что неважно, с каких чисел x и y начинать. При условии, что все промежуточные вычисления выполняются правильно, алгоритм в конечном итоге заканчивается получением результата произведения *x* \* *y* двух исходных чисел. 

Таким образом, вы никогда не получите неправильный ответ, и алгоритм не может зациклиться.

## Анализ числа операций

Ваш школьный учитель, возможно, не обсуждал число примитивных операций, необходимых для завершения процедуры умножения в столбик. В нашем примере, для того чтобы вычислить первое частичное произведение, мы умножили 4 раза каждую из цифр 5, 6, 7, 8 первого числа. Это четыре примитивные операции. Мы также выполнили несколько сложений из-за переносов. В общем случае вычисление частичного произведения влечет за собой n умножений (одно умножение на один знак) и не более n сложений (не более одного на один знак). Всего получается не более 2*n* примитивных операций. Первое частичное произведение ничем не отличается от других, и каждое из них требует не более 2*n* операций. Поскольку имеется n частичных произведений — по одному на каждый знак второго числа, — вычисление всех из них требует не более *n* \* 2*n* = 2*n*2 примитивных операций. Чтобы вычислить окончательный ответ, нам все еще нужно все частичные произведения сложить вместе, но для этого требуется сопоставимое число операций (не более чем еще 2*n*2). Подведем итоги:

Рассуждая о том, каким образом объем работы, который этот алгоритм выполняет, возрастает по мере того, как исходные множители становятся все длиннее и длиннее, мы видим, что объем выполняемых операций растет квадратически, следуя за увеличением разрядности. Если удвоить длину исходных множителей, то требуемый объем операций подскакивает в 4 раза. Увеличьте их длину в 4 раза, и она подскочит в 16 раз, и так далее.

## Можно ли добиться лучшего?

В зависимости от того, каким третьеклассником вы были, вы вполне могли принять эту процедуру как уникальный или, по крайней мере, оптимальный способ умножения двух чисел. Если вы захотите стать серьезным проектировщиком алгоритмов, то нужно будет преодолеть эту робость. Классическая книга по алгоритмам Ахо, Хопкрофта и Ульмана, после итеративного рассмотрения целого ряда методик разработки алгоритмов, говорит следующее:

«Для хорошего разработчика алгоритмов, пожалуй, самый важный принцип состоит в том, чтобы отказаться от соглашательства».

Или как говорится, что каждый разработчик алгоритмов должен принять как должное:

Можно ли добиться лучшего?

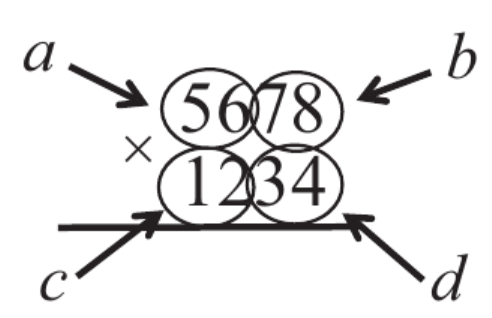
Этот вопрос особенно актуален, когда вы сталкиваетесь с очевидным или прямым решением вычислительной задачи. В третьем классе вы, возможно, не задавались вопросом о том, можно ли выстроить путь решения лучше, чем это делает простой классический алгоритм умножения в столбик. Настало время задать этот вопрос и дать на него ответ.

# 3.1 Умножение Карацубы

Чтобы почувствовать, что такое умножение Карацубы, давайте снова воспользуемся нашим предыдущим примером с *x* = 5678 и *y* = 1234. Мы выполним последовательность шагов, совершенно отличающуюся от алгоритма начальной школы (умножения в столбик), результатом которого является произведение *x* \* *y*. Этот новый алгоритм должен показаться вам очень загадочным, похожим на фокус вынимания кролика из шляпы. Позже в этом разделе мы дадим точное объяснение того, что такое умножение Карацубы и почему оно работает.

Сейчас главное — на этом примере оценить, что существует потрясающий воображение массив различных вариантов решения вычислительных задач, в частности целочисленного умножения.

Для начала обозначим первую и вторую половины числа *x* как *a* и *b*, чтобы рассматривать их отдельно. Для нашего примера получаем *a* = 56 и *b* = 78. Аналогичным образом поступим с *y*, здесь пусть *c* и *d* соответственно обозначают первую и вторую половины числа *y*, то есть *с*=12 и *d*=34 (рис. 2).



*Рисунок 2. Представление четырехзначных чисел как пар двузначных чисел*

Затем мы выполним последовательность операций с участием только двузначных чисел *a*, *b*, *c* и *d* и, наконец, волшебным образом соберем все элементы вместе. Увидим, что в итоге это приведет нас к результату произведения *x* и *y*.

**Шаг 1:** Вычислить произведение *a* \* *c* = 56 \* 12, которое составляет 672 (можете проверить сами).

**Шаг 2:** Вычислить *b* \* *d* = 78 \* 34 = 2652.

Следующие два шага еще более непостижимые.

**Шаг 3:** Вычислить (*a* + *b*) \* (*c* + *d*) = 134 \* 46 = 6164.

**Шаг 4:** Вычесть результаты первых двух шагов из результата третьего шага: 6164 – 672 – 2652 = 2840.

Наконец, мы суммируем результаты шагов 1, 2 и 4, но только после добавления четырех конечных нулей к ответу на шаге 1 и двух конечных нулей к ответу на шаге 4.

**Шаг 5:** Вычислить 672 \* 104 + 2840 \* 102 + 2652 = 6 720 000 + 284 000 + 2652 = 70 066 552.

Что мы видим? Это точно такой же (правильный) результат, который получился у нас при применении алгоритма умножения в столбик!

Я не требую от вас сразу интуитивно понять то, что только что произошло.

Скорее, я надеюсь, что вы ощущаете некое сочетание озадаченности и интриги, и оценили тот факт, что, похоже, действительно существуют алгоритмы целочисленного умножения, принципиально отличающиеся от того, с которым вы познакомились еще в школе. Как только вы поймете, насколько сфера алгоритмизации разнообразна, вы начнете задумываться над вопросом: может быть, существует что-то более интересное, чем алгоритм умножения в столбик? Разве приведенный выше алгоритм не позволил вам продвинуться в этом понимании?

# 4 Бинарный поиск

Предположим, вы ищете фамилию человека в телефонной книге (какая древняя технология!). Она начинается с буквы «К». Конечно, можно начать с самого начала и перелистывать страницы, пока вы не доберетесь до буквы «К». Но скорее всего для ускорения поиска лучше раскрыть книгу на середине: ведь буква «К» должна находиться где­то ближе к середине телефонной книги.

Или предположим, что вы ищете слово в словаре, и оно начинается с буквы «0». И снова лучше начать с середины.

Теперь допустим, что вы вводите свои данные при входе на Facebook. При этом Facebook необходимо проверить, есть ли у вас учетная запись на сайте. Для этого ваше имя пользователя нужно найти в базе данных. Допустим, вы выбрали себе имя пользователя «karlrnageddon». Facebook может начать с буквы А и проверять все подряд, но разумнее будет начать с середины.

Перед нами типичная задача поиска. И во всех этих случаях для решения задачи можно применить один алгоритм: бинарный поиск.

Бинарный поиск - это алгоритм; на входе он получает отсортированный список элементов (позднее я объясню, почему он должен быть отсортирован). Если элемент, который вы ищете, присутствует в списке, то бинарный поиск возвращает ту позицию, в которой он был найден. В противном случае бинарный поиск возвращает null.

Например: Рассмотрим пример того, как работает бинарный поиск. Сыграем в простую игру: я загадал число от 1 до 100.



Вы должны отгадать мое число, использовав как можно меньше попыток. При каждой попытке я буду давать один из трех ответов: «мало», «много» или «угадал».

Предположим, вы начинаете перебирать все варианты подряд: 1, 2, 3, 4 …. Вот как это будет выглядеть.







Это пример простого поиска (возможно, термин «тупой поиск» был бы уместнее). При каждой догадке исключается только одно число. Если я загадал число 99, то, чтобы добраться до него, потребуется 99 попыток!

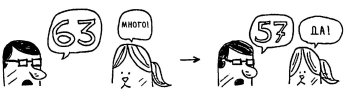
## Более эффективный поиск

Существует другой, более эффективный способ. Начнем с 50.

Слишком мало... но вы только что исключили половину чисел! Теперь вы знаете, что все числа 1-50 меньше загаданного. Следующая попытка: 75.

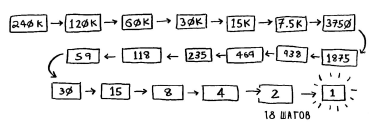


На этот раз перелет. .. Но вы снова исключили половину оставшихся чисел! С бинарным поиском вы каждый раз загадываете число в середине диапазона и ис'КЛючаете половину оставшихся чисел. Следующим будет число 63 (по середине между 50 и 75).



Так работает бинарный поиск. А вы только что узнали свой первый алгоритм! Попробуем поточнее определить, сколько чисел будет исключаться каждый раз.Какое бы число я ни задумал, вы гарантированно сможете угадать его не более чем за 7 попыток, потому что с каждой попыткой исключается половина оставшихся чисел!

Предположим, вы ищете слово в словаре с 240 ООО словами. Как вы думаете, сколько попыток вам понадобится в худшем случае? При простом поиске может потребоваться 240 ООО попыток, если искомое слово находится на самой последней позиции в книге. С каждым шагом бинарного поиска количество слов сокращается вдвое, пока не останется только одно слово.

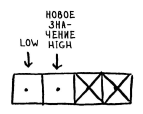


Итак, бинарный поиск потребует 18 шагов - заметная разница! В общем случае для списка из *n* элементов бинарный поиск выполняется за *log*2*n* шагов, тогда как простой поиск будет выполнен за *n* шагов.

Таким образом:

Функция получает отсортированный массив и значение. Если значение присутствует в массиве, то функция возвращает его позицию. При этом мы должны следить за тем, в какой части массива проводится поиск.

Если названное число было слишком мало, то переменная low обновляется на середину отрезка А если догадка была слишком велика, то обновляется переменная high.



Упражнения

1. Имеется отсортированный список из 128 имен, и вы ищете в нем значение методом бинарного поиска. Какое максимальное количество проверок для этого может потребоваться?

2 Предположим, размер списка увеличился вдвое. Как изменится максимальное количество проверок?