Содержание

Введение

Программирование под Windows всегда было достаточно сложной задачей. Интерфейс прикладного программирования (Application Programming Interface – API) Windows предоставляет в ваше распоряжение набор мощных, но не всегда безопасных инструментов для разработки приложений. Эти инструменты в некотором смысле можно сравнить с огромной и тяжелой машиной, при помощи которой удается добиться поразительных результатов, но если водитель неосторожен или не владеет соответствующими навыками, дело, скорее всего, закончится только разрушениями и убытками.

С появлением Delphi ситуация изменилась. С помощью интерфейса для быстрой разработки приложений (Rapid Application development – RAD) Delphi позволяет быстро и легко выполнять подобную работу. Используя Delphi, можно создавать и тестировать приложения со сложным пользовательским интерфейсом без прямого использования функций API. Освобождая программиста от проблем, связанных с применением API, Delphi позволяет сконцентрироваться непосредственно на приложении.

Несмотря на то, что Delphi упрощает создание пользовательского интерфейса, писать остальную часть приложения – код для обработки действий пользователя и отображения результатов – предоставляется программисту. И здесь потребуются алгоритмы.

Алгоритмы – это формальные команды, необходимые для выполнения на компьютере сложных задач. Например, с помощью алгоритма поиска можно найти конкретную информацию в базе данных, состоящей из 10 млн записей. В зависимости от качества используемых алгоритмов искомые данные могут быть обнаружены за секунды, часы или вообще не найдены.

В этой книге не только подробно рассказывается об алгоритмах, написанных на Delphi, но и приводится много готовых мощных алгоритмов. Здесь также анализируются методы управления структурами данных, такими как списки, стеки, очереди и деревья; описываются алгоритмы для выполнения типичных задач – сортировки, поиска и хеширования.

Для того чтобы успешно использовать алгоритмы, недостаточно просто скопировать код в свою программу и запустить ее на выполнение. Необходимо знать, как различные алгоритмы ведут себя в разных ситуациях. В конечном итоге именно эта информация определяет выбор наиболее подходящего варианта.

Книга написана на достаточно простом языке. Здесь рассматривается поведение алгоритмов как в типичных, так наихудших случаях. Это позволит понять, чего вы вправе ожидать от определенного алгоритма, вовремя распознать возможные

трудности и при необходимости переписать или удалить алгоритм. Даже самый лучший алгоритм не поможет в решении задачи, если использовать его неправильно.

Все алгоритмы представлены в виде исходных текстов на Delphi, которые вы можете включать в свои программы без каких-либо изменений. Тексты кода и примеры приложений находятся на сайте издательства «ДМК Пресс» www.dmkpress.ru. Они демонстрируют характерные особенности работы алгоритмов и их использование в различных программах.

Назначение книги

Данная книга содержит следующий материал:

- *полное введение в теорию алгоритмов*. После прочтения книги и выполнения приведенных примеров вы сможете использовать сложные алгоритмы в своих проектах и критически оценивать новые алгоритмы, написанные вами или кем-то еще;
- *большую подборку исходных текстов*. С помощью текстов программ, имеющихся на сайте издательства «ДМК Пресс», вы сможете быстро добавить готовые алгоритмы в свои приложения;
- *готовые примеры программ* позволят вам проверить алгоритмы. Работая с этими примерами, изменяя и совершенствуя их, вы лучше изучите принцип работы алгоритмов. Кроме того, вы можете использовать их как основу для создания собственных приложений.

Читательская аудитория

Книга посвящена профессиональному программированию в Delphi. Она не предназначена для обучения. Хорошее знание основ Delphi позволит вам сконцентрировать внимание на алгоритмах вместо того, чтобы погружаться в детали самого языка.

Здесь изложены важные принципы программирования, которые могут с успехом применяться для решения многих практических задач. Представленные алгоритмы используют мощные программные методы, такие как рекурсия, разбиение на части, динамическое распределение памяти, а также сетевые структуры данных, что поможет вам создавать гибкие и сложные приложения.

Даже если вы еще не овладели Delphi, вы сможете выполнить примеры программ и сравнить производительность различных алгоритмов. Более того, любой из приведенных алгоритмов будет нетрудно добавить к вашим проектам.

Совместимость версий Delphi

Выбор наилучшего алгоритма зависит от основных принципов программирования, а не от особенностей конкретной версии языка. Тексты программ в этой книге были проверены с помощью Delphi 3, 4 и 5, но благодаря универсальности свойств языка они должны успешно работать и в более поздних версиях Delphi.

Языки программирования, как правило, развиваются в сторону усложнения и очень редко в противоположном направлении. Яркий тому пример – оператор `goto` в языке C. Этот неудобный оператор является потенциальным источником ошибок, он почти не используется большинством программистов на C, но сохранился в синтаксисе языка еще с 70-х годов. Оператор даже был встроен в C++ и позднее в Java, хотя создание нового языка было хорошим предлогом избавиться от ненужного наследия.

Аналогично в старших версиях Delphi наверняка появятся новые свойства, но вряд ли исчезнут стандартные блоки, необходимые для реализации алгоритмов, описанных в этой книге. Независимо от того, что добавлено в 4-й, 5-й, и будет добавлено в 6-й версии Delphi, классы, массивы, и определяемые пользователем типы данных останутся в языке. Большая часть, а может быть, и все алгоритмы из этой книги не будут изменяться еще в течение многих лет. Если вам понадобится обновить алгоритмы, то их можно будет найти на сайте www.vb-helper. com/da.htm.

Содержание глав

В **главе 1** рассматриваются те основы, которые вам необходимо изучить, прежде чем приступать к анализу сложных алгоритмов. Здесь описываются методы анализа вычислительной сложности алгоритмов. Некоторые алгоритмы, теоретически обеспечивающие высокую производительность, в реальности дают не очень хорошие результаты. Поэтому в этой главе обсуждаются и практические вопросы, например, рассматривается обращение к файлу подкачки.

В **главе 2** рассказывается, как можно сформировать различные виды списков с помощью массивов и указателей. Эти структуры данных применяются во многих программах, что продемонстрировано в следующих главах книги. В главе 2 также показано, как обобщить методы, использованные для построения связанных списков, для создания других, более сложных структуры данных, например, деревьев и сетей.

В **главе 3** рассматриваются два специализированных вида списков – стеки и очереди, использующиеся во многих алгоритмах (некоторые их них описываются в последующих главах). В качестве практического примера приведена модель, сравнивающая производительность двух типов очередей, которые могли бы использоваться в регистрационных пунктах аэропортов.

Глава 4 посвящена специальным типам массивов. Треугольные, неправильные и разреженные массивы позволяют использовать удобные представления данных для экономии памяти.

В **главе 5** рассматривается мощный, но довольно сложный инструмент – рекурсия. Здесь рассказывается, в каких случаях можно использовать рекурсию и как ее можно при необходимости удалить.

В **главе 6** многие из представленных выше алгоритмов, такие как рекурсия и связанные списки, используются для изучения более сложного вопроса – деревьев. Рассматриваются различные представления деревьев – с помощью полных узлов и нумерации связей. Здесь содержатся также некоторые важные алгоритмы, например, обход узлов дерева.

В **главе 7** затронута более широкая тема. Сбалансированные деревья обладают некоторыми свойствами, которые позволяют им оставаться уравновешенными и эффективными. Алгоритмы сбалансированных деревьев просто описать, но довольно трудно реализовать в программе. В этой главе для построения сложной базы данных используется одна из наиболее мощных структур – Б+ дерево.

В **главе 8** рассматриваются алгоритмы, которые предназначены для поиска ответа в дереве решений. Даже при решении маленьких задач эти деревья могут быть поистине огромными, поэтому становится насущным вопрос эффективного поиска нужных элементов. В этой главе сравнивается несколько различных методов подобного поиска.

Глава 9 посвящена наиболее сложному разделу теории алгоритмов. Алгоритмы сортировки интересны по нескольким причинам. Во-первых, сортировка – это общая задача программирования. Во-вторых, различные алгоритмы сортировки имеют свои достоинства и недостатки, и нет единого универсального алгоритма, который бы работал одинаково в любых ситуациях. И наконец, в алгоритмах сортировки используется множество разнообразных методов, таких как рекурсия, бинарные деревья, применение генератора случайных чисел, что уменьшает вероятность выпадения наихудшего случая.

Глава 10 посвящена вопросам сортировки. Как только список отсортирован, программе может потребоваться найти в нем какой-либо элемент. В этой главе сравниваются наиболее эффективные методы поиска элементов в сортированных списках.

В **главе 11** приводятся более быстрые, чем использование деревьев, способы сортировки и поиска, методы сохранения и размещения элементов. Здесь описывается несколько методов хеширования, включая использование блоков и связанных списков, а также некоторые типы открытой адресации.

В **главе 12** обсуждается другая категория алгоритмов – сетевая. Некоторые из подобных алгоритмов, например, вычисление кратчайшего пути, непосредственно применяются в физических сетях. Они могут косвенно использоваться для решения других проблем, которые на первый взгляд кажутся не относящимися к сетям. Например, алгоритм поиска кратчайшего пути может делить сеть на районы или находить критические точки в сетевом графике.

Глава 13 посвящена объектно-ориентированным алгоритмам. В них используются объектно-ориентированные способы реализации нетипичного для традиционных алгоритмов поведения.

В **приложении 1** описывается содержание архива примеров, который находится на сайте издательства «ДМК Пресс» www.dmkpress.ru.

В **приложении 2** содержатся все программы примеров, имеющихся в архиве. Для того чтобы найти, какая из программ демонстрирует конкретные алгоритмические методы, достаточно обратиться к этому списку.

Архив примеров

Архив примеров, который вы можете загрузить с сайта издательства «ДМК Пресс» www.dmkpress.ru, содержит исходный код в Delphi 3 для алгоритмов и примеров программ, описанных в книге.

Описанные в каждой главе примеры программ содержатся в отдельных подкаталогах. Например, программы, демонстрирующие алгоритмы, которые рассматриваются в главе 3, сохранены в каталоге `\Ch3\`. В приложении 2 перечисляются все приведенные в книге программы.

Аппаратные требования

Для освоения примеров необходим компьютер, конфигурация которого удовлетворяет требованиям работы с Delphi, то есть почти каждый компьютер, работающий с любой версией Windows.

На компьютерах с различной конфигурацией алгоритмы выполняются с неодинаковой скоростью. Компьютер с процессором Pentium Pro с частотой 200 МГц и объемом оперативной памяти 64 Мб, безусловно, будет работать быстрее, чем компьютер на базе процессора Intel 386 и объемом памяти 4 Мб. Вы быстро определите предел возможностей ваших аппаратных средств.

Как пользоваться этой книгой

В главе 1 дается базовый материал, поэтому необходимо начать именно с этой главы. Даже если вам уже известны все тонкости теории алгоритмов, все равно необходимо прочесть эту главу.

Следующими нужно изучить главы 2 и 3, поскольку в них рассматриваются различные виды списков, используемых программами в следующих главах книги.

В главе 6 обсуждаются понятия, которые используются затем в главах 7, 8, и 12. Перед тем как заняться изучением этих глав, вы должны ознакомиться с главой 6. Остальные главы можно читать в произвольном порядке.

В табл. 1 приведены три примерных плана работы с материалом. Вы можете выбрать один из них, руководствуясь тем, насколько глубоко вы хотите изучить алгоритмы. Первый план предполагает освоение основных методов и структур данных, которые вы можете успешно использовать в собственных программах. Второй план помимо этого включает в себя работу с фундаментальными алгоритмами, такими как алгоритмы сортировки и поиска, которые могут вам понадобиться для разработки более сложных программ.

Последний план определяет порядок изучения всей книги. Несмотря на то, что главы 7 и 8 по логике должны следовать за главой 6, она гораздо сложнее, чем более поздние главы, поэтому их рекомендуется прочесть позже. Главы 7, 12 и 13 наиболее трудные в книге, поэтому к ним лучше обратиться в последнюю очередь. Конечно, вы можете читать книгу и последовательно – от самой первой страницы до последней.

Таблица 1. Планы работы

Изучаемый материал	Главы												
Основные методы	1	2	3	4									
Базовые алгоритмы	1	2	3	4	5	6	9	10	13				
Углубленное изучение	1	2	3	4	5	6	9	10	11	8	12	7	13

Обозначения, используемые в книге

В книге используются следующие шрифтовые выделения:

- *курсивом* помечены смысловые выделения в тексте;
- **полужирным шрифтом** выделяются названия элементов интерфейса: пунктов меню, пиктограмм и т.п.;
- `моноширинным шрифтом` выделены листинги (программный код).

Глава 1. Основные понятия

В этой главе представлен базовый материал, который необходимо усвоить перед началом более серьезного изучения алгоритмов. Она открывается вопросом «Что такое алгоритмы?». Прежде чем погрузиться в детали программирования, стоит вернуться на несколько шагов назад для того, чтобы более четко определить для себя, что же подразумевается под этим понятием.

Далее приводится краткий обзор формальной *теории сложности алгоритмов* (complexity theory). При помощи этой теории можно оценить потенциальную вычислительную сложность алгоритмов. Такой подход позволяет сравнивать различные алгоритмы и предсказывать их производительность в различных условиях работы. В данной главе также приведено несколько примеров применения теории сложности для решения небольших задач.

Некоторые алгоритмы на практике работают не так хорошо, как предполагалось при их создании, поэтому в данной главе обсуждаются практические вопросы разработки программ. Чрезмерное разбиение памяти на страницы может сильно уменьшить производительность хорошего в остальных отношениях приложения.

Изучив основные понятия, вы сможете применять их ко всем алгоритмам, описанным в книге, а также для анализа собственных программ. Это позволит вам оценить производительность алгоритмов и предупреждать различные проблемы еще до того, как они приведут к катастрофе.

Что такое алгоритмы

Алгоритм – это набор команд для выполнения определенной задачи. Если вы объясняете кому-то, как починить газонокосилку, вести автомобиль или испечь пирог, вы создаете алгоритм действий. Подобные ежедневные алгоритмы можно с некоторой точностью описать такого рода выражениями:

```
Проверьте, находится ли автомобиль на стоянке.
Убедитесь, что он поставлен на ручной тормоз.
Поверните ключ.
И т.д.
```

Предполагается, что человек, следующий изложенным инструкциям, может самостоятельно выполнить множество мелких операций: отпереть и открыть двери, сесть за руль, пристегнуть ремень безопасности, найти ручной тормоз и т.д.

Если вы составляете алгоритм для компьютера, то должны все подробно описать заранее, в противном случае машина вас не поймет. Словарь компьютера (язык программирования) очень ограничен, и все команды должны быть сформулированы на доступном машине языке. Поэтому для написания компьютерных алгоритмов следует использовать более формализованный стиль.

Увлекательно писать формализованный алгоритм для решения какой-либо бытовой, ежедневной задачи. Например, алгоритм вождения автомобиля мог бы начинаться примерно так:

```
Если дверь заперта, то:
Вставьте ключ в замок
Поверните ключ
Если дверь все еще заперта, то:
Поверните ключ в другую сторону
Потяните за ручку двери
и т.д.
```

Эта часть кода описывает только открывание двери; здесь даже не проверяется, та ли дверь будет открыта. Если замок заклинило или автомобиль оснащен противоугонной системой, алгоритм открывания двери может быть гораздо сложнее.

Алгоритмы были формализованы еще тысячи лет назад. Еще в 300 году до н.э. Евклид описал алгоритмы для деления углов пополам, проверки равенства треугольников и решения других геометрических задач. Он начал с небольшого словаря аксиом, таких как «параллельные линии никогда не пересекаются», и создал на их основе алгоритмы для решения более сложных задач.

Формализованные алгоритмы данного типа хорошо подходят для решения математических задач, где нужно доказать истинность каких-либо положений или возможность каких-нибудь действий, при этом скорость алгоритма не имеет значения. При решении реальных задач, где необходимо выполнить некоторые инструкции, например сортировку на компьютере записей о миллионе покупателей, эффективность алгоритма становится критерием оценки алгоритма.

Анализ скорости выполнения алгоритмов

Теория сложности изучает сложность алгоритмов. Существует несколько способов измерения сложности алгоритма. Программисты обычно сосредотачивают внимание на скорости алгоритма, но не менее важны и другие показатели – требования к объему памяти, свободному месту на диске. Использование быстрого алгоритма не приведет к ожидаемым результатам, если для его работы понадобится больше памяти, чем есть у вашего компьютера.

Память или время

Многие алгоритмы предлагают выбор между объемом памяти и скоростью. Задачу можно решить быстро, используя большой объем памяти, или медленнее, занимая меньший объем.

Типичным примером в данном случае служит алгоритм поиска кратчайшего пути. Представив карту города в виде сети, можно написать алгоритм для определения кратчайшего расстояния между любыми двумя точками в этой сети. Чтобы не вычислять эти расстояния всякий раз, когда они вам нужны, вы можете вывести кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда вам понадобится узнать кратчайшее расстояние между двумя заданными точками, вы можете взять готовое значение из таблицы.

Результат будет получен практически мгновенно, но это потребует огромного объема памяти. Карта улиц большого города, такого как Бостон или Денвер, может содержать несколько сотен тысяч точек. Таблица, хранящая всю информацию о кратчайших расстояниях, должна иметь более 10 млрд ячеек. В этом случае выбор между временем исполнения и объемом требуемой памяти очевиден: используя дополнительные 10 Гб памяти, можно сделать выполнение программы более быстрым.

Из этой особенной зависимости между временем и памятью проистекает идея *объемо-временной сложности*. При таком способе анализа алгоритм оценивается как с точки зрения скорости, так и с точки зрения используемой памяти. Таким образом находится компромисс между этими двумя показателями.

В данной книге основное внимание уделяется временной сложности, но также указываются и некоторые особые требования к объемам памяти для некоторых алгоритмов. Например, *сортировка слиянием* (mergesort), рассматриваемая в главе 9, требует очень больших объемов оперативной памяти. Для других алгоритмов, например *пирамидальной сортировки* (heapsort), которая также описывается в главе 9, достаточно обычного объема памяти.

Оценка с точностью до порядка

При сравнении различных алгоритмов важно понимать, как их сложность зависит от сложности решаемой задачи. При расчетах по одному алгоритму сортировка тысячи чисел занимает 1 с, сортировка миллиона чисел – 10 с, в то время как на те же расчеты по другому алгоритму уходит 2 с и 5 с соответственно. В подобных случаях нельзя однозначно сказать, какая из этих программ лучше. Скорость обработки зависит от вида сортируемых данных.

Хотя интересно иметь представление о точной скорости каждого алгоритма, но важнее знать различие производительности алгоритмов при выполнении задач различной сложности. В приведенном примере первый алгоритм быстрее сортирует короткие списки, а второй – длинные.

Скорость алгоритма можно оценить по порядку величины. Алгоритм имеет сложность O(f (N)) (произносится «O большое от F от N»), функция F от N, если с увеличением размерности исходных данных N время выполнения алгоритма возрастает с той же скоростью, что и функция f (N). Например, рассмотрим следующий код, который сортирует N положительных чисел:

```
for i := 1 to N do
begin
   // Нахождение максимального элемента списка.
   MaxValue := 0;
   for j := 1 to N do
      if (Value[j]>MaxValue) then
      begin
         MaxValue := Value[J];
         MaxJ := J;
      end;
```

```
    // Печать найденного максимального элемента.
    PrintValue(MaxValue);
    // Обнуление элемента для исключения его из дальнейшего поиска.
    Value[MaxJ] := 0;
end;
```

В этом алгоритме переменная `i` последовательно принимает значения от 1 до N. При каждом изменении `i` переменная `j` также изменяется от 1 до N. Во время каждой из N-итераций внешнего цикла внутренний цикл выполняется N раз. Общее количество итераций внутреннего цикла равно N * N или N^2. Это определяет сложность алгоритма $O(N^2)$ (пропорциональна N^2).

Оценивая порядок сложности алгоритма, необходимо использовать только ту часть уравнения рабочего цикла, которая возрастает быстрее всего. Предположим, что рабочий цикл алгоритма представлен формулой $N^3 + N$. В таком случае его сложность будет равна $O(N^3)$. Рассмотрение быстро растущей части функции позволяет оценить поведение алгоритма при увеличении N.

При больших значениях N для процедуры с рабочим циклом $N^3 + N$ первая часть уравнения доминирует и вся функция сравнима со значением N^3. Если N = 100, то разница между $N^3 + N = 1\,000\,100$ и $N^3 = 1\,000\,000$ равна всего лишь 100, что составляет 0,01%. Обратите внимание на то, что это утверждение истинно только для больших N. При N = 2 разница между $N^3 + N = 10$ и $N^3 = 8$ равна 2, что составляет уже 20%.

При вычислении значений «большого O» можно не учитывать постоянные множители в выражениях. Алгоритм с рабочим циклом $3 * N^2$ рассматривается как $O(N^2)$. Таким образом, зависимость отношения O(N) от изменения размера задачи более очевидна. Если увеличить N в 2 раза, эта двойка возводится в квадрат (N^2) и время выполнения алгоритма увеличивается в 4 раза.

Игнорирование постоянных множителей также облегчает подсчет шагов выполнения алгоритма. В приведенном ранее примере внутренний цикл выполняется N^2 раз. Сколько шагов делает каждый внутренний цикл? Чтобы ответить на этот вопрос, вы можете вычислить количество условных операторов `if`, потому что только этот оператор выполняется в цикле каждый раз. Можно сосчитать общее количество инструкций внутри условного оператора `if`. Кроме того, внутри внешнего цикла есть инструкции, не входящие во внутренний цикл, такие как команда `PrintValue`. Нужно ли считать и их?

С помощью различных методов подсчета можно определить, какую сложность имеет алгоритм N^2, $3 * N^2$, или $3 * N^2 + N$. Оценка сложности алгоритма по порядку величины даст одно и то же значение $O(N^2)$, поэтому неважно, сколько точно шагов имеет алгоритм.

Определение сложности

Наиболее сложными частями программы обычно является выполнение циклов и вызовов процедур. В предыдущем примере весь алгоритм выполнен с помощью двух циклов.

Если одна процедура вызывает другую, то необходимо более тщательно оценить сложность последней. Если в ней выполняется определенное число инструкций,

например, вывод на печать, то на оценку порядка сложности она практически не влияет. С другой стороны, если в вызываемой процедуре выполняется O(N) шагов, то функция может значительно усложнять алгоритм. Если процедура вызывается внутри цикла, то влияние может быть намного больше.

В качестве примера возьмем программу, содержащую медленную процедуру `Slow` со сложностью порядка $O(N^3)$ и быструю процедуру `Fast` со сложностью порядка $O(N^2)$. Сложность всей программы зависит от соотношения между этими двумя процедурами.

Если при выполнении циклов процедуры `Fast` всякий раз вызывается процедура `Slow`, то сложности процедур перемножаются. Общая сложность равна произведению обеих сложностей. В данном случае сложность алгоритма составляет $O(N^2) * O(N^3)$ или $O(N^3 * N^2) = O(N^5)$. Приведем соответствующий фрагмент кода:

```
procedure Slow;
var
   i, j, k : Integer;
begin
   for i := 1 to N do
      for j := 1 to N do
         for k := 1 to N do
            // Выполнение каких-либо действий.
end;

procedure Fast;
var
   i, j : Integer;
begin
   for i := 1 to N do
      for j := 1 to N do
         Slow; // Вызов процедуры Slow.
end;

procedure RunBoth;
begin
   Fast;
end;
```

С другой стороны, если основная программа вызывает процедуры отдельно, их вычислительная сложность складывается. В этом случае итоговая сложность по порядку величины равна $O(N^3) + O(N^2) = O(N^3)$. Следующий фрагмент кода имеет именно такую сложность:

```
procedure Slow;
var
   i, j, k : Integer;
begin
   for i := 1 to N do
      for j := 1 to N do
         for k := 1 to N do
            // Выполнение каких-либо действий.
end;
```

```
procedure Fast;
var
   i, j : Integer;
begin
   for i := 1 to N do
      for j := 1 to N do
         // Выполнение каких-либо действий.
end;

procedure RunBoth;
begin
   Fast;
   Slow;
end;
```

Сложность рекурсивных алгоритмов

Рекурсивные процедуры (recursive procedure) – это процедуры, которые вызывают сами себя. Их сложность определяется очень тонким способом. Сложность многих рекурсивных алгоритмов зависит именно от количества итераций рекурсии. Рекурсивная процедура может казаться достаточно простой, но она может очень серьезно усложнять программу, многократно вызывая саму себя.

Следующий фрагмент кода описывает процедуру, которая содержит только две операции. Тем не менее, если задается число N, то эта процедура выполняется N раз. Таким образом, вычислительная сложность данного алгоритма равна O(N).

```
procedure CountDown(N : Integer);
begin
   if (N<=0) then exit;
   CountDown(N-1);
end;
```

Многократная рекурсия

Рекурсивный алгоритм, вызывающий себя несколько раз, называется *многократной рекурсией* (multiple recursion). Процедуры множественной рекурсии сложнее анализировать, чем однократные алгоритмы, кроме того, они могут сделать алгоритм гораздо сложнее.

Следующая процедура аналогична процедуре `CountDown`, только она вызывает саму себя дважды.

```
procedure DoubleCountDown(N : Integer);
begin
   if (N<=0) then exit;
   DoubleCountDown(N-1);
   DoubleCountDown(N-1);
end;
```

Поскольку процедура вызывается дважды, можно было бы предположить, что ее рабочий цикл будет вдвое больше, чем цикл процедуры `CountDown`. При этом

сложность была бы равна 2 * O(N) = O(N). В действительности ситуация гораздо сложнее.

Если количество итераций процедуры при входном значении N равно T(N), то легко заметить, что T(0) равно 1. Если процедура вызывается с параметром 0, то программа просто закончит свою работу с первого шага.

Для больших значений N процедура запускается дважды с параметром N – 1. Количество ее итераций при этом равно 1 + 2 * T(N – 1). В табл. 1.1 приведены некоторые значения сложности алгоритма в соответствии с уравнениями T(0) = 1 и T(N) = 1 + 2 * T(N – 1). При внимательном рассмотрении этих значений можно заметить, что если $T(N) = 2^{(N+1)} - 1$, то рабочий цикл процедуры будет равен $O(2^N)$. Несмотря на то, что процедуры `CountDown` и `DoubleCountDown` выглядят почти одинаково, `DoubleCountDown` выполняется гораздо дольше.

Таблица 1.1. Значения длительности рабочего цикла для процедуры DoubleCountDown

N	0	1	2	3	4	5	6	7	8	9	10
T(N)	1	3	7	15	31	63	127	255	511	1023	2047

Косвенная рекурсия

Рекурсивная процедура может выполняться косвенно, вызывая вторую процедуру, которая, в свою очередь, вызывает первую. *Косвенную рекурсию* даже сложнее анализировать, чем многократную. Алгоритм кривых Серпинского, рассматриваемый в главе 5, включает в себя четыре процедуры, которые являются одновременно и многократной и косвенной рекурсией. Каждая из этих процедур вызывает себя и три другие процедуры до четырех раз. Такой значительный объем работы выполняется в течение времени $O(4^N)$.

Объемная сложность рекурсивных алгоритмов

Для некоторых рекурсивных алгоритмов особенно важна *объемная сложность*. Очень просто написать рекурсивный алгоритм, который запрашивает небольшой объем памяти при каждом вызове. Объем занятой памяти может увеличиваться в процессе последовательных рекурсивных вызовов. По этой причине необходимо провести хотя бы поверхностный анализ объемной сложности рекурсивных процедур, чтобы убедиться, что программа не исчерпает при выполнении все доступные ресурсы.

Следующая процедура выделяет больше памяти при каждом вызове. После 100 или 200 рекурсивных обращений процедура займет всю свободную память компьютера, и программа аварийно остановится, выдав сообщение об ошибке Out of Memory (Недостаточно памяти).

```
procedure GobbleMemory;
var
   x : Pointer;
begin
```

```
   GetMem(x,100000); // Выделяет 100000 байт.
   GobbleMemory;
end;
```

В главе 5 вы найдете более подробную информацию о рекурсивных алгоритмах.

Средний и наихудший случай

Оценка сложности алгоритма до порядка является верхней границей сложности алгоритмов. Если программа имеет больший порядок сложности, это не означает, что алгоритм будет действительно выполняться так долго. При задании правильных данных выполнение многих алгоритмов занимает гораздо меньше времени, чем можно предположить на основании порядка их сложности. Например, следующий код иллюстрирует простой алгоритм, который определяет расположение элемента в списке.

```
function LocateItem(target : Integer) : Integer;
var
   i : Integer;
begin
   for i := 1 to N do
   if (Value[i]=target) then
   begin
      Result := i;
      break;
   end;
end;
```

Если искомый элемент находится в конце списка, то программе придется исследовать все N элементов списка, чтобы обнаружить нужный. Это займет N шагов, и сложность алгоритма будет равна O(N). В данном, так называемом *наихудшем случае* (worst case) время работы алгоритма будет максимальным.

С другой стороны, если искомый число расположено в самом начале списка, алгоритм завершит работу почти сразу же. Он выполнит несколько шагов, прежде чем найдет искомый номер и остановится. Это *наилучший случай* (best case) со сложностью порядка O(1). Строго говоря, подобный случай не очень интересен, поскольку он вряд ли произойдет в реальной жизни. Интерес представляет *средний* или *ожидаемый вариант* (expected case) поведения алгоритма.

Если номера элементов в списке изначально беспорядочно смешаны, то искомый элемент может оказаться в любом месте списка. В среднем потребуется исследовать N/2 элементов для того, чтобы найти требуемый. Значит, сложность этого алгоритма в усредненном случае будет порядка O(N/2), или O(N), если убрать постоянный множитель.

Для некоторых алгоритмов наихудший случай сильно отличается от ожидаемого случая. Например, алгоритм быстрой сортировки, описанный в главе 9, имеет наихудший случай поведения $O(N^2)$, а ожидаемое поведение равно O(N * log(N)), что гораздо быстрее. Алгоритмы, подобные алгоритму быстрой сортировки, иногда

делают очень длинными, чтобы исключить возникновение наихудшего случая поведения.

Общие функции оценки сложности

В табл. 1.2 приведены некоторые функции, которые наиболее часто используются для вычисления сложности. Функции перечислены в порядке возрастания сложности. Это значит, что алгоритмы со сложностью, вычисляемой с помощью функций, которые помещены вверху таблицы, будут выполняться быстрее алгоритмов, сложность которых вычисляется с помощью ниже расположенных функций.

Таблица 1.2. Общие функции оценки сложности

Функция	Примечание
$f(N) = C$	C – константа
$f(N) = \log(\log(N))$	
$f(N) = \log(N)$	
$f(N) = N^C$	C – константа между 0 и 1
$f(N) = N$	
$f(N) = N*\log(N)$	
$f(N) = N^C$	C – константа больше 1
$f(N) = C^N$	C – константа больше 1
$f(N) = N!$	т.е. 1 * 2 * ... * N

Таким образом, уравнение сложности, которое содержит несколько этих функций, при приведении в систему оценки сложности по порядку величины будет сокращаться до функции, расположенной ниже в таблице. Например, $O(\log(N) + N^2)$ – это то же самое, что и $O(N^2)$.

Сможет ли алгоритм работать быстрее, зависит от того, как вы его используете. Если вы запускаете алгоритм раз в год для решения задач с достаточно малыми объемами данных, то вполне приемлема производительность $O(N^2)$. Если же алгоритм выполняется под наблюдением пользователя в интерактивном режиме, оперируя большими объемами данных, то может быть недостаточно и производительности $O(N)$.

Обычно алгоритмы со сложностью $N * \log(N)$ работают с очень хорошей скоростью. Алгоритмы со сложностью N^C при небольших значениях C, например N^2, применяются, когда объемы данных ограничены. Вычислительная сложность алгоритмов, порядок которых определяется функциями C^N и $N!$ очень велика, поэтому эти алгоритмы пригодны только для решения задач с очень малым объемом перерабатываемой информации.

Один из способов рассмотрения относительных размеров этих функций заключается в определении времени, которое требуется для решения задач различных размеров. Табл. 1.3 показывает, как долго компьютер, осуществляющий миллион

операций в секунду, будет выполнять некоторые медленные алгоритмы. Из таблицы видно, что только небольшие задачи можно решить с помощью алгоритмов со сложностью $O(C^N)$, и самые маленькие – с помощью алгоритмов со сложностью O(N!). Для решения задач порядка O(N!), где N = 24, потребовалось бы больше времени, чем существует вселенная.

Таблица 1.3. Время выполнения сложных алгоритмов

	N = 10	N = 20	N = 30	N = 40	N = 50
N^3	0,001 с	0,008 с	0,027 с	0,064 с	0,125 с
2^N	0,001 с	1,05 с	17,9 мин	1,29 дней	35,7 лет
3^N	0,059 с	58,1 мин	6,53 лет	$3{,}86 * 10^5$ лет	$2{,}28 * 10^{10}$ лет
N!	3,63 с	$7{,}71 * 10^4$ лет	$8{,}41 * 10^{18}$ лет	$2{,}59 * 10^{34}$ лет	$9{,}64 * 10^{50}$ лет

Логарифмы

Прежде чем продолжить изложение материала, необходимо рассмотреть логарифмы, так как они играют важную роль во многих алгоритмах. Логарифм числа N по основанию B – это степень P, в которую нужно возвести число B, чтобы выполнялось равенство $B^P = N$. Например, выражение $\log_2 8$ следует читать «степень, в которую необходимо возвести 2, чтобы получилось 8». В этом случае, $2^3 = 8$ или $\log_2 8 = 3$.

Преобразовывать логарифмы от одного основания к другому можно с помощью зависимости $\log_B N = \log_C N / \log_C B$. Если вы хотите преобразовать $\log_2 8$ к основанию 10, то это будет выглядеть так: $\log_{10} N = \log_2 N / \log_2 10$. Значение $\log_2 10$ – константа, которая приблизительно равна 3,32. Поскольку постоянные множители при оценке по порядку сложности можно опустить, допускается не учитывать член $\log_2 10$.

Для любого основания B значение $\log_2 B$ – константа. Это означает, что для оценки по порядку сложности основание логарифма не имеет значения. Другими словами, $O(\log_2 N)$ равно $O(\log_{10} N)$ или $O(\log_B N)$ для любого B. Поскольку основание логарифмов не имеет значения, часто просто пишут, что сложность алгоритма составляет O(log N).

В программировании используется двоичная система счисления, поэтому логарифмы, используемые при анализе сложности алгоритмов, обычно имеют основание 2. Для того чтобы упростить выражения, мы везде будем писать log N, подразумевая $\log_2 N$. Если используется другое основание, это будет обозначено особо.

Скорость работы алгоритма в реальных условиях

Несмотря на то, что малые члены и постоянные множители отбрасываются при изучении сложности алгоритмов, часто их необходимо учитывать для фактического написания программ. Эти числа становятся особенно важными, когда размер задачи мал, а константы большие.