

Ага! Алгоритмы



Изучение теории алгоритмов может оказаться очень полезным программисту-практику. Курс лекций по этому предмету снабжает студентов функциями для решения важных задач и учит их находить подходы к решению новых задач. В следующих главах мы увидим, каким образом более совершенные алгоритмы могут влиять на программное обеспечение, сокращая время на разработку и повышая эффективность программ.

Алгоритмы играют существенную роль в решении обычных задач программирования. В своей книге «Ага! Озарение» (Martin Gardner, Aha! Insight), откуда я бессовестно украл заглавие, Мартин Гарднер описывает то, что я имею в виду: «Кажущаяся сложной задача может иметь простое неожиданное решение». В отличие от более совершенных методов, «ага!-алгоритмы» не требуют высшего образования; они могут прийти в голову любому программисту, готовому серьезно задуматься, до написания программы, в процессе ее кодирования или после этого.

2.1. Три задачи

Довольно общих слов! Эта глава посвящена трем небольшим задачам. Попробуйте решить их самостоятельно, прежде чем продолжить ознакомление с этой главой.

1. Пусть есть некий файл, содержащий не более четырех миллиардов 32-битных целых чисел в случайном порядке. Нужно найти число, которое отсутствует в файле (а хотя бы одно число будет отсутствовать наверняка — почему?). Как бы вы решили эту задачу при наличии неограниченной оперативной памяти? Как вы решите ее, если оперативная память ограничена парой сотен килобайт, но разрешается использование нескольких временных файлов?
2. Осуществите циклический сдвиг одномерного массива из n элементов на i позиций влево. Например, если $n=8$, а $i=3$, вектор `abcdefgh` превращается

в defghabc. Простейшая программа использует n -элементный вспомогательный массив и выполняет задачу за n шагов. Можете ли вы сдвинуть массив за время, пропорциональное n , используя лишь несколько десятков байт под дополнительные переменные?

3. Дан словарь английского языка. Требуется найти все наборы анаграмм. Например, слова pots, stop и tops являются анаграммами друг для друга, поскольку каждое из этих слов может быть получено перестановкой букв любого другого.

2.2. Вездесущий двоичный поиск

Я задумываю целое число от 1 до 100, а вы его угадываете. 50? Мало. 75? Много. И так игра продолжается до тех пор, пока вы не угадаете задуманное мною число. Если число взято из диапазона $1..n$, то оно может быть наверняка угадано за $\log_2 n$ попыток. Если $n=1000$, достаточно будет 10 попыток, а если $n=1\,000\,000$, потребуется не более 20 попыток.

Этот пример иллюстрирует метод, позволяющий решить множество задач программирования, — *двоичный поиск*. В начальный момент мы знаем, что объект находится в заданной области, а операция выбора и проверки значения сообщает нам, где находится объект: в текущей позиции, выше или ниже ее. При двоичном поиске местоположение объекта обнаруживается с помощью повторяющегося выбора элемента из середины текущей области. Если выбранный элемент отличается от искомого, текущая область делится пополам, после чего процесс выбора и сравнения повторяется. Поиск закончен, если обнаруживается искомым элемент или пустая область.

Самое обычное применение двоичного поиска — поиск элемента в отсортированном массиве. При поиске числа 50 будут проверены следующие элементы (рис. 2.1).

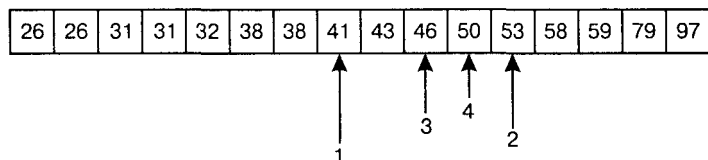


Рис. 2.1. Двоичный поиск в отсортированном массиве

Существует мнение, что программу двоичного поиска тяжело написать без ошибок. Подробно мы изучим ее в главе 4.

Программа последовательного поиска выполняет в среднем около $n/2$ сравнений при поиске в наборе из n элементов, тогда как двоичный поиск всегда делает не более $\log_2 n$. Этот факт может очень сильно повлиять на производительность системы. Вот пример, относящийся к системе бронирования билетов авиакомпании TWA:

«Одна из программ осуществляла последовательный поиск в огромном объеме памяти примерно 100 раз в секунду. По мере роста сети среднее время обработки сообщения возросло до 0,3 мс, что оказалось для нас слишком много. Мы установили,

что проблема заключается в использовании процедуры последовательного поиска, заменили ее процедурой двоичного поиска, и проблема исчезла».

Я часто сталкивался с этим и в других системах. Программисты начинают с простой структуры данных и последовательного поиска, который поначалу оказывается достаточно быстрым. Когда производительности подобной программы начинает не хватать, можно отсортировать таблицу и воспользоваться двоичным поиском, что устраняет проблему.

Однако поле применения двоичного поиска не ограничивается быстрым поиском в отсортированных массивах. Рой Вейл (Roy Weil) применил этот метод для нахождения некорректной строки во входном файле, содержащем более 1000 строк. К сожалению, строку эту нельзя было обнаружить по ее внешнему виду; узнать о ее наличии можно было, лишь обработав файл (начальную часть до произвольного места) некой программой и получив некорректный ответ, на что требовалось несколько минут. Его предшественники, пытавшиеся найти ошибку, пробовали обрабатывать несколько строк за один проход, благодаря чему продвигались к цели со скоростью улитки. Попробуйте догадаться, как удалось Вейлу найти ошибку за десять попыток?

Теперь, когда мы слегка размялись, можно обратиться к задаче 1. На вход поступает последовательный файл (записанный на каком-либо носителе или на диске — хотя к диску и можно обращаться произвольно, последовательное чтение обычно осуществляется с гораздо большей скоростью). Этот файл содержит не более четырех миллиардов 32-битных целых чисел в произвольном порядке, а нам необходимо найти число, отсутствующее в этом файле. Такое число обязательно есть, поскольку 32-битных целых чисел всего 2^{32} , или 4 294 967 296. При наличии неограниченного объема памяти мы могли бы воспользоваться битовым массивом, описанным в главе 1, и выделить 536 870 912 байт (по 8 бит каждый) под все возможные целые числа. Однако нам предлагается найти недостающее число, используя лишь несколько сот байт оперативной памяти и несколько вспомогательных файлов последовательного доступа. Чтобы реализовать двоичный поиск в этой ситуации, прежде всего нужно определить диапазон, представление элементов диапазона и метод проверки, позволяющий определить, в какой половине этого диапазона не хватает числа.

Диапазоном у нас будет набор целых чисел, в котором заведомо недостает по крайней мере одного элемента, а представлять мы его будем в виде файла, содержащего все числа этого диапазона. И вот: озарение! Мы можем проверять диапазон, подсчитывая количество элементов, меньших и больших значения середины диапазона. Либо в верхней, либо в нижней половине диапазона будет не более половины общего количества чисел в диапазоне. Поскольку хотя бы один элемент во всем диапазоне обязательно отсутствует, в одной из половин диапазона также будет отсутствовать по крайней мере один элемент, и в этой половине будет меньшее число элементов. Вот основные составляющие алгоритма двоичного поиска. Попробуйте соединить их вместе самостоятельно, прежде чем подсмотреть ответ Эда Рейнгольда (Ed Reingold).

В этих двух примерах мы лишь слегка коснулись необъятной области применения двоичного поиска в программировании. При поиске корней уравнения

с одной переменной можно использовать метод *бисекции* — последовательного деления интервала, содержащего корень, на равные части. Алгоритм в решении задачи 9 из главы 11 выбирает случайный элемент, а затем рекурсивно вызывается для области по одну сторону от выбранного элемента. Такой алгоритм называется *случайным* (иногда его называют «*рандомизованным*») *двоичным поиском*. Другие приложения двоичного поиска включают деревья (структуры данных) и отладку программ. (Если программа не выводит сообщения об ошибке при внезапном завершении работы, как вы будете вести поиск ошибки?) Во всех подобных случаях программисту может помочь понимание того факта, что все подобные программы являются всего лишь версиями базового алгоритма двоичного поиска.

2.3. Мощь элементарного

Мы нашли решение: двоичный поиск. Теперь попробуем поискать задачи для этого решения. Рассмотрим задачу, которая может иметь несколько решений. Задача 2 заключается в циклическом сдвиге массива x из n элементов влево на i позиций за время, пропорциональное n , причем доступно лишь несколько десятков байт оперативной памяти. В некоторых языках программирования операция циклического сдвига является элементарной (то есть выполняется одним оператором). Для нас важно, что циклический сдвиг соответствует обмену соседних блоков памяти разного размера: при перемещении фрагмента текста с помощью мыши из одного места файла в другое осуществляется именно эта операция. Ограничения по времени и объему памяти существенны для многих подобных приложений.

Можно попытаться решить задачу, копируя первые i элементов массива x во временный массив, сдвигая оставшиеся $n-i$ элементов влево на i позиций, а затем копируя данные из временного массива обратно в основной массив на последние i позиций. Однако данная схема использует i дополнительных переменных, что требует дополнительной памяти. Другой подход заключается в том, чтобы определить функцию, сдвигающую массив влево на один элемент (за время, пропорциональное n), а потом вызывать ее i раз, но такой алгоритм будет отнимать слишком много времени.

Решение проблемы с указанными ограничениями на использование ресурсов потребует написания более сложной программы. Одним из вариантов решения будет введение дополнительной переменной. Элемент $x[0]$ помещается во временную переменную t , затем $x[i]$ помещается в $x[0]$, $x[2i]$ — в $x[i]$ и так далее (перебираются все элементы массива x с индексом по модулю n), пока мы не возвращаемся к элементу $x[0]$, вместо которого записывается содержимое переменной t , после чего процесс завершается. Если $i = 3$, а $n = 12$, этот этап проходит следующим образом (рис. 2.2).

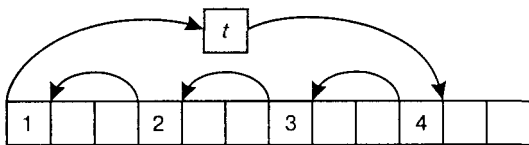


Рис. 2.2. Вариант решения задачи с циклическим сдвигом

Если при этом не были переставлены все имеющиеся элементы, процедура повторяется, начиная с $x[1]$ и так далее, до достижения конечного результата. В задании 3 вам предлагается воплотить это решение в виде кода. Будьте аккуратны.

Можно предложить и другой алгоритм, который возникает из рассмотрения задачи с другой точки зрения. Циклический сдвиг массива x сводится фактически к замене ab на ba , где a — первые i элементов x , а b — оставшиеся элементы. Предположим, что a короче b . Разобьем b на b_1 и b_2 , где b_1 содержит i элементов (столько же, сколько и a). Поменяем местами a и b_1 , получим $b_1 a$. При этом a окажется в конце массива — там, где и полагается. Поэтому можно сосредоточиться на перестановке b_1 и b_2 . Эта задача сводится к начальной, поэтому алгоритм можно вызывать рекурсивно. Программа, реализующая этот алгоритм, будет достаточно красивой (в решении к заданию 3 описывается итерационное решение Гриса (Gries) и Миллса (Mills)), но она требует аккуратного написания кода, а оценить ее эффективность не просто.

Задача кажется сложной, пока вас не осенит озарение («ага!»): итак, нужно преобразовать массив ab в ba . Предположим, что у нас есть функция `reverse`, переставляющая элементы некоторой части массива в противоположном порядке. В исходном состоянии массив имеет вид ab . Вызвав эту функцию для первой части, получим $a b$. Затем вызовем ее для второй части: получим $a b$. Затем вызовем функцию для всего массива, что даст $(a b)$, в это в точности соответствует ba . Посмотрим, как будет такая функция действовать на массив $abcdefgh$, который нужно сдвинуть влево на три элемента:

```
reverse(0, 1-1) /* cbadefgh */
reverse(1, n-1) /* cbahgfed */
reverse(0, n-1) /* defghabc */
```

Дуг Макилрой (Doug McIlroy) предложил наглядную иллюстрацию циклического сдвига массива из десяти элементов вверх на пять позиций (рис. 2.3); начальное положение: обе руки ладонями к себе, левая над правой.

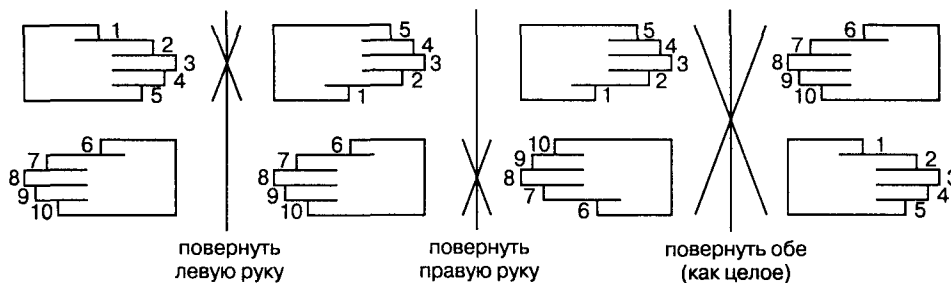


Рис. 2.3. Иллюстрация циклического сдвига

Код, использующий функцию переворота, оказывается эффективным и малотребовательным к памяти, и настолько короток и прост, что при его реализации сложно ошибиться. Б. Керниган и П. Дж. Плуджер пользовались именно этим методом для перемещения строк в текстовом редакторе в своей книге (B. Kernighan, P. J. Plauger, *Software Tools in Pascal*, 1981). Керниган пишет, что эта функция за-

работала правильно с первого же запуска, тогда как их предыдущая версия, использовавшая связный список, содержала несколько ошибок. Этот же код используется в некоторых текстовых редакторах, включая тот, в котором я впервые набрал настоящую главу. Кен Томпсон (Ken Thompson) написал этот редактор с функцией reverse в 1971 году, и он утверждает, что она уже тогда была легендарной.

2.4. Соберем все вместе: сортировка

Вернемся к задаче 3. Дается словарь английского языка (по одному слову в каждой строке входного файла), нужно найти все возможные анаграммы. Существует несколько причин, по которым стоит заняться этой задачей. Во-первых, решение будет красивой комбинацией правильного подхода и правильных методов. Важность второй причины в некоторых ситуациях неоспорима: вам же не хочется оказаться единственным человеком на вечеринке, который не знает, что deposit, dopiest, posited и topside — анаграммы? А если этих причин недостаточно, в задаче 6 будет описана аналогичная проблема, возникающая в реальной жизни.

Множество предложенных подходов к решению задачи анаграмм оказываются на практике неэффективными и сложными. Любой метод, рассматривающий все перестановки букв каждого слова, обречен. Слово cholecystoduodenostomy (являющееся анаграммой duodenocholecystostomy из моего словаря) дает 22 (!) перестановки, что приблизительно равно $1,124 \times 10^{21}$. Даже если предположить, что проверки будут выполняться со скоростью самого быстродействующего компьютера, выполняющего 10^9 перестановок в секунду, — обработка одного слова займет $1,1 \times 10^9$ секунд. Одно из правил большого пальца¹ (см. раздел 7.1) гласит, что « y секунд равно 1 нановеку», то есть $1,1 \times 10^9$ секунд — это почти тридцать пять лет! Любой метод, сравнивающий все пары слов, потребует по меньшей мере целую ночь работы на моем компьютере. В словаре, который я использовал, содержится 230 000 слов, а даже простейшее сравнение анаграмм требует не меньше 1 мкс, поэтому полное время работы составит приблизительно:

$230\,000 \text{ слов} * 230\,000 \text{ сравнений/слово} * 1 \text{ мкс/сравнение} = 52\,900 * 10^6 \text{ микросекунд} = 52\,900 \text{ секунд} = 14.7 \text{ часа}$

Можно ли найти способ обойти эти вычислительные ловушки?

Озарение «ага!»: придумать способ вычисления сигнатуры слова таким образом, чтобы все слова, принадлежащие к одному классу анаграмм, имели одинаковую сигнатуру, затем собрать эти слова вместе. При этом исходная задача разделяется на две подзадачи: выбор сигнатуры и объединение всех слов с одинаковой сигнатурой. Подумайте над этим самостоятельно, прежде чем читать дальше.

Для решения первой подзадачи воспользуемся сортировкой: упорядочим буквы слова в алфавитном порядке². Сигнатурой слова deposit будет deiopst, причем

¹ Эмпирические правила приближенных вычислений. — *Примеч. перео.*

² Этот алгоритм нахождения анаграмм был независимо открыт несколькими программистами приблизительно в середине 60-х годов.

это же сочетание букв является также сигнатурой слова *dopiest* и любого другого слова из этого класса анаграмм. Для решения второй подзадачи упорядочим слова так, чтобы их сигнатуры располагались в алфавитном (лексикографическом) порядке. Лучшая иллюстрация была предложена Томом Каргиллом (Tom Cargill): сортировать нужно сначала в этом порядке (взмах рукой слева направо), затем в этом (сверху вниз). В разделе 2.8 описана реализация этого алгоритма.

2.5. Принципы Сортировка

Наиболее очевидная цель применения сортировки — получение отсортированного результата, ценного либо уже самого по себе, либо в качестве промежуточного массива для дальнейшей обработки (например, двоичного поиска). В задаче об анаграммах важным было не столько упорядочение при сортировке, сколько группировка одинаковых элементов (сигнатур). Получение сигнатур является еще одной из возможных целей сортировки: упорядочение букв в слове дает его каноническую форму. Добавление ключей к записям и последующая сортировка по этим ключам дают возможность использовать сортировку для упорядочения данных в дисковых файлах. В разделе 3 мы еще несколько раз вернемся к сортировке.

Двоичный поиск

Алгоритм поиска элемента в упорядоченном наборе оказывается на редкость эффективным и может применяться к данным, находящимся в памяти или на диске. Его единственным недостатком является то, что доступной и заранее отсортированной должна быть вся таблица. Идея, на которой основывается этот простой алгоритм, используется во многих приложениях.

Сигнатуры

Когда отношение эквивалентности используется для определения классов, оказывается полезным определить сигнатуру, отличающую эти классы друг от друга. Сортировка букв слова дает одну из возможных сигнатур для анаграмм, другие варианты могут быть получены, к примеру, путем сортировки и последующей замены повторяющихся букв цифрами, указывающими их количество в слове. Сигнатурой слова *mississippi* в этом случае будет *i4m1p2s4* или *i4mp2s4* (если единицу по умолчанию не указывать). Можно также хранить массив из 26 целых чисел (для каждого слова), который будет содержать информацию о количестве повторяющихся букв. Другие области применения сигнатур включают используемый ФБР метод индексации отпечатков пальцев и эвристические алгоритмы Soundex для объединения имен, одинаково звучащих, но по-разному пишущихся (табл. 2.1).

Кнут описывает метод Soundex в главе 6 третьего тома книги «Искусство программирования для ЭВМ, сортировка и поиск».

Таблица 2.1. Сигнатуры одинаково звучащих имен

Имя	Сигнатура Soundex
Smith	s530
Smythe	s530
Schultz	s243
Shultz	s432

Постановка задачи

В главе 1 было наглядно показано, что доскональное выяснение требований заказчика является основополагающей частью процесса написания программы. Тема этой главы — следующая стадия процесса написания программы, а именно поиск ответа на вопрос: «Какими базовыми средствами может быть реализована программа?» Во всех примерах суть озарения состояла в использовании новой базовой операции, заметно упрощающей задачу.

Перспективы для программиста

Лень — двигатель прогресса. Хорошие программисты всегда немножко ленивы. Они сидят и ждут озарения, вместо того, чтобы воплощать в жизнь первую пришедшую им в голову идею. Лень, однако, должна компенсироваться вовремя возникающим желанием писать программу. Самое сложное — это определить, когда же наступает это должное время. Помочь тут может только опыт решения задач и анализа решений.

2.6. Задачи

1. Решите задачу нахождения всех анаграмм одного слова. Как бы вы ее решили, если бы вам дали только это слово и словарь? Что, если бы у вас была возможность обработать словарь, прежде чем искать анаграммы?
2. Дан последовательный файл, содержащий 4 300 000 000 32-битных целых чисел. Как найти целое число, которое встречается дважды?
3. Мы кратко упомянули о двух алгоритмах циклического сдвига массива, требовавших аккуратного кодирования. Реализуйте их. Как в этих программах используется наибольший общий делитель i и n ?
4. Несколько читателей обратили внимание на то, что, хотя все три предложенных алгоритма работают за время, пропорциональное n , алгоритм обмена (см. раздел 2.3) оказывается примерно вдвое быстрее алгоритма реверсирования: он сохраняет во временный файл и возвращает обратно каждый элемент лишь один раз, в то время как алгоритм реверсирования делает это дважды. Поэкспериментируйте с функциями, чтобы проверить скорость их работы на реальных машинах; при этом обязательно нужно учитывать, к какой области памяти происходит обращение.

2.6. Задачи 37

5. Функции сдвига массива меняют ab на ba . Как бы вы преобразовали вектор abc к виду cba ? Эта задача моделирует обмен областей памяти, не прилегающих друг к другу.
6. В конце 70-х в лабораториях Белл была разработана программа «Справочной службы, управляемой пользователем», которая позволяла предпринимателям находить номера в телефонном справочнике, используя стандартный телефон с кнопочным набором (рис. 2.4).

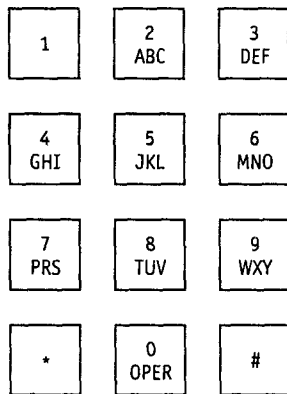


Рис. 2.4. Клавиатура кнопочного телефона

- Чтобы найти номер разработчика системы М. Леска (Mike Lesk), нужно было набрать номер LESK*M* (то есть 5375*6*), и система озвучивала его номер. Сейчас подобные службы распространены повсеместно. Проблема может возникнуть, если несколько имен имеют один и тот же код; когда такое случалось в системе Леска, его программа запрашивала у пользователя дополнительную информацию. Если дан большой список имен наподобие обычного телефонного справочника, как бы вы выделили подобные совпадения? Леек провел подобный эксперимент и выяснил, что количество совпадений составляло всего лишь 0,2%. Как бы вы реализовали функцию, возвращающую список имен, задаваемых кодом, подаваемым на ее вход?
7. В начале 60-х Вик Высоцкий (Vic Vissotski) работал с программистом, которому нужно было транспонировать матрицу 4000 на 4000 элементов, хранившуюся на магнитной ленте. Все записи имели один и тот же формат и были длиной порядка нескольких десятков байт. Первому варианту программы, который был предложен этим программистом, потребовалось бы около 50 часов на выполнение задачи. Как удалось Высоцкому уменьшить это время до получаса?
8. [J. Ullman] Дан набор из n вещественных чисел, вещественное число t , целое k . Как можно быстро определить существование k -элементного поднабора, сумма элементов которого не превышает t ?

9. Последовательный поиск выполняется медленно, но не требует предварительной обработки, тогда как двоичный поиск работает очень быстро, но требует наличия отсортированного набора. Сколько нужно выполнить двоичных поисков на n -элементном массиве, чтобы окупить время, затраченное на его предварительную сортировку?

10. Когда к Томасу Эдисону обратился очередной претендент на должность помощника, Эдисон предложил ему вычислить объем пустой колбы от лампочки. Через несколько часов вычислений с кронциркулем в руках претендент вернулся с результатом — 150 см³. После нескольких секунд вычислений Эдисон ответил «ближе к 155». Как он это сделал?

2.7. Дополнительная литература

В разделе 8.8 главы 8 приведены ссылки на несколько хороших книг по теории алгоритмов.

2.8. Реализация поиска анаграмм

Для решения этой задачи я написал три небольшие программы, соединив их в канал обработки. При этом выводимый одной из программ текст поступал на вход второй программы. Первая программа определяет сигнатуры слов, вторая — сортирует, а третья — объединяет слова одного класса в одной строке. Вот пример обработки словаря из шести слов (рис. 2.5).

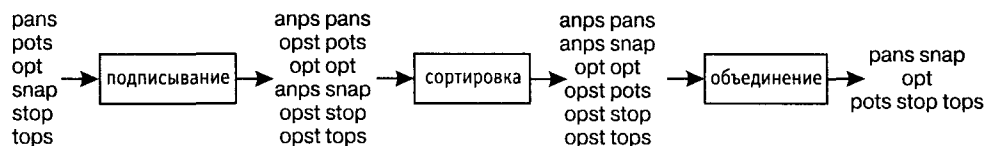


Рис. 2.5. Пример обработки словаря из 6 слов

В результате получаем три класса анаграмм.

В приведенной ниже программе на С (см. листинг 2.1) предполагается, что ни одно слово не состоит более чем из 100 букв и входной файл содержит только строчные буквы и символы перевода строки. (Поэтому перед ее запуском пришлось пропустить словарь через программу, приводившую все символы к нижнему регистру).

Листинг 2.1. Программа подписывания анаграмм

```

int charcomp(char *x, char *y) { return *x - *y; }
#define WORDMAX 100
int main(void)

```

```

{
    char word[WORDMAX], sig[WORDMAX];
    while (scanf("%s", word) != EOF) {
        strcpy(sig, word);
        qsort(sig, strlen(sig), sizeof(char), charcomp);
        printf("%s %s\n", sig, word);
    }
    return 0;
}

```

Цикл `while` считывает по одной строке в переменную `word`, пока не доберется до конца файла. Функция `strcpy` копирует строку в переменную `sig`, символы которой затем сортируются с помощью стандартной библиотечной функции `qsort` (аргументы: сортируемый массив, его длина, длина элемента, указатель на функцию сравнения двух элементов). Наконец, `printf` выводит сигнатуру и слово в файл, завершая строку символом перевода строки.

Для сортировки используется системная программа сортировки, а написанная мной программа `squash` объединяет анаграммы одного класса в одной строке.

Листинг 2.2. Объединение анаграмм одного класса в одной строке

```

int main(void)
{
    char word[WORDMAX], sig[WORDMAX], oldsig[WORDMAX];
    int linenum=0;
    strcpy(oldsig, "");
    while (scanf("%s %s", sig, word) != EOF) {
        if (strcmp(oldsig, sig)!=0 && linenum > 0)
            printf("\n");
        strcpy(oldsig, sig);
        linenum++;
        printf("%s", word);
    }
    printf("\n");
    return 0;
}

```

Основная работа выполняется вторым оператором `printf` — он выводит второе поле каждой строки и разделитель (пробел). Оператор `if` определяет момент смены сигнатуры: если `sig` отличается от `oldsig` (предыдущего значения сигнатуры), в выходной файл записывается символ перевода строки (если эта запись не первая в файле). Последний оператор `printf` записывает в конечный файл символ перевода строки.

После проверки этих небольших программ на тестовых файлах я получил список анаграмм из большого словаря, набрав:

```
sign <dictionary | sort squash >gramlist
```

Эта команда скармливает файл `dictionary` программе `sign`, направляет ее вывод на вход программы `sort`, вывод последней направляется на вход `squash`, а результаты ее работы сохраняются в файле `gram list`. Программа выполнялась всего 18 секунд: 4 на `sign`, 11 на `sort` и 3 на `squash`.

Я применил эту программу к словарю, содержащему 230 000 слов. Он, однако, не включал многие окончания типа -s и -ed. А вот самые интересные классы анаграмм:

```
subessential suitability  
canter creant cretan nectar recant tanrec trance  
caret carte cater crate creat creta react recta trace  
destain instead sainted satined  
adroitly dilatory idolatry  
least setal slate stale steal stela tales  
reins resin rinse risen serin siren  
constitutionalism misconstitutional
```