Раздел 4 Тексты

Компьютеры и тексты

Вначале было слово.

Так говорит история человечества. В истории компьютеров вначале было число. Долгое время вместо термина «компьютер» использовались аббревиатуры «ЭВМ» (Электронная Вычислительная Машина) и «ЦВМ» (Цифровая Вычислительная Машина), что подчеркивало цифровую сущность первых компьютеров. И использовались они тогда в отраслях, связанных с военными применениями, в зарождающейся космической отрасли, в физике, - в тех областях, где господствовала цифра. Тогда в почете были физики, а не лирики с их, казалось бы, ненужными текстами.

Ситуация начала меняться в 60-х годах. И в немалой степени способствовали переменам сами компьютеры, развитие которых потребовало создания языков программирования. Нотацию для записи программ на уровне более высоком, чем система команд компьютера, стали называть «языком». Если первый язык программирования Fortran (Formula Translator) сохранил в своем названии приверженность к формулам, то последующий языки – AlgoL, CoboL, SnoboL, PL/I несли в своем имени букву L от слова «язык» (Language). Программы для компьютеров, написанные на языках программирования, стали восприниматься как тексты. И одной из главных задач, порожденных внутренними потребностями самого программирования, стала задача трансляции. К тому времени языкотворчество стало модным, число языков исчислялось сотнями. Для каждого языка нужен был транслятор - программа, осуществляющая перевод текста с одного (формального) языка в другой формальный язык (язык системы команд компьютера).

В первых языках программирования – Фортране и Алголе по-прежнему практически отсутствовали средства представления текстовой информации и работы с ней. В сборнике упражнений по Алголу [9], подготовленном на факультете ВМК МГУ и вышедшем в 1975 году, нет ни одного упражнения по работе с текстовой информацией, все упражнения предназначены для работы с числами. Приведу еще цитату из книги [8], вышедшей у нас в 1980 году и посвященной обзору расплодившихся тогда языков программирования: «Можно сказать, что для «научных» языков программирования характерно полное или почти полное отсутствие средств для работы со строками литер».

Однако время господства цифры уже прошло и ей пришлось уступить символу, занявшему законное первое место в компьютерных программах. Этому во многом способствовали успехи математической лингвистики, теории и практики разработки трансляторов, теории формальных грамматик и теории автоматов. Появились языки логического программирования. В японском проекте машин пятого поколения делалась ставка на эти языки в надежде на существенное повышение «интеллекта» программ, немыслимое без работы с текстами. Возникающие здесь задачи оказались ничуть не проще, а во многом сложнее задач вычислительной математики. Надежды решить их с помощью компьютера оправдались лишь в малой степени. Например, задача перевода с одного естественного языка на другой до сих пор решена лишь частично. До сих пор приемлемый перевод технических текстов требует вмешательства человека. Для художественных текстов компьютерный перевод может быть полезен лишь как подстрочник. Но не все так плохо, и несомненные успехи в этой области достигнуты. Интеллектуальные роботы, автомобили, дома - становятся частью нашего мира. Мой хороший друг, прекрасный математик и шахматист Леонид Дмитриевич Иванов полагал, что компьютеры никогда не смогут одолеть шахматного мастера. Он ошибался. Сегодня чемпион мира еще может соревноваться с компьютером, но завтра справиться с компьютером в этой игре человеку будет невозможно. Компьютеры считают лучше. Они быстрее просчитывают варианты, они могут обработать в короткое время огромные массивы разнородной информации – числа, тексты, картинки и звуки. Что же остается человеку? Многое. Действительно новую информацию создает человек. Он создает «настоящие» тексты. Компьютеры могут лишь их обрабатывать.

Появление персональных компьютеров в каждом доме, а затем и появление компьютерных сетей, создало новую реальность – информационный мир. Ежедневно миллионы людей создают новые тексты, размещая их в Интернете – этом громадном хранилище текстов. Денно и нощно поисковые машины перелопачивают эту груду, индексируя их, наводя хоть какой-то порядок, позволяющий по запросу найти нужный текст. Без людей, создающих тексты, и без компьютеров, обрабатывающих эти тексты, Интернет, как хранилище информации, был бы бесполезным.

Здесь есть еще одна невидимая сторона дела – алгоритмическая сложность задач, решаемых в процессе поиска. Пользователям Интернета, далеким от понимания алгоритмов, может казаться совершенно естественным, что на их запрос уже через секунды выдается большое число ссылок на тексты с запрашиваемой информацией. Пользователи могут жаловаться, что ссылок слишком много, не все из них действительно соответствуют запросу, но в целом система работает удовлетворительно. У специалиста, представляющего, какие объемы текстов следует просмотреть для получения ответов, работоспособность системы должна вызывать изумление и уважение. Примитивные алгоритмы работы с текстами не смогли бы привести к успеху поиска.

Интернет далеко не единственная область, где подобные алгоритмы играют важнейшую роль. Молекулярная биология (и ее раздел - биоинформатика) является сегодня бурно развивающейся научной областью. Как ни странно, а может быть вполне естественно, что при анализе структур ДНК и РНК, при расшифровке генома человека работа с текстами играет определяющую роль. В книге Дэна Гансфилда [Строки], подробно рассматриваются алгоритмы работы с текстами, как необходимый инструментарий решения задач вычислительной биологии. Приведу из нее некоторые цитаты, поясняющие, как биологическая информация представляется в виде текста: «Можно получить биологически осмысленные результаты, рассматривая ДНК как одномерную строку символов… Аналогичное, но более сильное предположение делается и о белках… Информация, которая лежит за биохимией, клеточной биологией и развитием, может быть представлена обычной строкой, составленной из 4-х символов G, A, T и C. Для биологии организмов эта строка является исходной структурой данных».

(В одной из задач, приводимых ниже, поясняется смысл используемых символов).

Символьные данные

Дадим формальное определение строки символов и других важных понятий.

Пусть $T = \{t_1, t_2, \ldots t_n\}$ – конечное множество, которое будем называть алфавитом, а его элементы – символами алфавита. Строкой s (словом, цепочкой) в алфавите T будем называть последовательность подряд записанных символов из T – ($s = c_1c_2...c_m$). Число символов в слове – m будем называть длиной слова. Пустое слово длины 0 будем обозначать символом ε или константой вида `""`. Если s – строка длины m ($s = c_1c_2...c_m$), то подстрокой s[i, j], где $1 <= i <= j <= m$, ,будем называть часть строки из s, начиная с i-го символа и заканчивая символом j - ($c_ic_{i+1}\ldots c_j$). Подстрока s[1, j] называется префиксом слова, а подстрока s[i, m] – суффиксом слова.

Алфавит T можно рассматривать как множество слов длины 1. Рассмотрим операцию конкатенации над множествами, так, что конкатенация алфавита T с самим собой дает множество всех слов длины 2. Обозначается конкатенация TT как T^2. Обозначим через

- T^k – множество слов в алфавите T длины k, - его можно рассматривать как k-кратную конкатенацию алфавита T;
- T^+ - множество всех непустых слов в алфавите T произвольной длины. Это множество получается, как результат объединения множеств T^k по всем возможным значениям k;
- T^* - множество всех возможных слов в алфавите T. Это множество называется итерацией алфавита и его можно рассматривать как объединение пустого слова с множеством T^+.

Рассмотрим примеры некоторых алфавитов. Алфавит $T_2 = \{0, 1\}$ – любые данные, хранимые в памяти компьютера, можно рассматривать, как слова в этом алфавите. Алфавит $T_{10} = \{0, 1, \ldots 9\}$ – целые числа без знака в десятичной системе счисления являются словами в этом алфавите. Четырехбуквенные алфавиты $T_{DNK} = \{A, T, G, C\}$ и $T_{DNK} = \{A, U, G, C\}$ используются для строк, задающих структуру молекул ДНК и РНК. Каждая молекула ДНК и РНК представляет комбинацию 4-х нуклеотидов: Аденин (A), тимин (T), цитозин (C) или гуанин (G) в ДНК; аденин (A), урацил (U), цитозин (C) или гуанин (G) в РНК.

В основе каждого языка лежит некоторый алфавит T. Если под языком G(T) понимать множество всех слов данного языка в алфавите T, то G(T) является подмножеством T*. Задать конкретный язык можно разными способами. Для языков программирования широко применяется аппарат формальных грамматик, позволяющий описать синтаксически корректные программы.

Определим класс языков, задаваемых регулярными множествами. **Регулярное множество** определяется рекурсивно следующими правилами:

- пустое множество, а также множество, содержащее пустое слово, и одноэлементные множества, содержащие символы алфавита, являются регулярными базисными множествами;
- если множества P и Q являются регулярными, то множества, построенные применением операций объединения, конкатенации и итерации – $P \bigcup Q$, PQ, P^*, Q^* – тоже являются регулярными.

Регулярные выражения представляют удобный способ задания регулярных множеств. Аналогично множествам, они определяются рекурсивно:

- регулярные базисные выражения задаются символами и определяют соответствующие регулярные базисные множества, например, выражение f задает одноэлементное множество {f} при условии, что f — символ алфавита T;
- если p и q – регулярные выражения, то операции объединения, конкатенации и итерации – p + q, pq, p*, q* — являются регулярными выражениями, определяющими соответствующие регулярные множества.

По сути, регулярные выражения – это более простой и удобный способ записи регулярных множеств в виде обычной строки. Каждое регулярное множество, а, следовательно, и каждое регулярное выражение задает некоторый язык L(T) в алфавите T. Этот класс языков достаточно мощный, с его помощью можно описать интересные языки, но устроены они довольно просто – их можно определить также с помощью простых грамматик, например, правосторонних грамматик. Более важно, что для любого регулярного выражения можно построить конечный автомат, который распознает, принадлежит ли заданное слово языку, порожденному регулярным выражением. На этом основана практическая ценность регулярных выражений.

С точки зрения практика регулярное выражение задает образец поиска. После чего можно проверить, удовлетворяет ли заданная строка или ее подстрока данному образцу. В языках программирования синтаксис регулярного выражения существенно обогащается, что дает возможность более просто задавать сложные образцы поиска. Такие синтаксические надстройки, хотя и не меняют сути регулярных выражений, крайне полезны для практиков, избавляя их от ненужных сложностей.

Какие операции над строками лежат в основе преобразований текста? Как и для арифметического типа, это операции отношения, позволяющие сравнивать строки, и некоторый набор операций, позволяющих из заданных строк получать новые строки.

Строковый тип, также как и арифметический тип, считается упорядоченным. Вначале задается порядок на символах алфавита, например как в кириллице от «А» до «Я». Порядок на алфавите порождает порядок на словах, называемый лексикографическим порядком. Не определяя его формально, скажу, что этот порядок задает расположение слов в словарях. Поэтому на строках определены операции сравнения не только на равенство, но и на «больше» и «меньше».

Пока не выработаны устоявшиеся обозначения для базисных операций над строками, подобные «+» или «-» для арифметического типа. Тем не менее, любой язык программирования, позволяющий работать с текстами, включает такие операции над строками как: конкатенация, поиск, вставка, удаление и замена подстрок. Эти операции можно считать базисными. С их помощью один текст можно преобразовать в другой текст, например сделать из «мухи» «слона», или осуществить перевод текста с одного языка на другой язык.

Представление символьных данных в C#

Для работы с текстами на языке C# библиотека классов FCL предлагает целый набор разнообразных классов, сосредоточенных в разных пространствах имен этой библиотеки. Классы для работы с текстами находятся как в основном пространстве имен System, так и в пространствах System.Text и System.Text.RegularExpression. Основные, но далеко не все из этих классов, подробно рассмотрены в учебнике [1]. Они же будут встречаться в задачах этого раздела.

Классы C# для работы с текстами

Рассмотрим несколько основных классов, позволяющих работать с текстовой информацией, - char, char[], string, StringBuilder. Класс System.Char (синоним класса char) определяет множество символов – алфавит языка. Объекты этого класса задают символы, используемые при работе с текстами в программах C#. Класс относится к значимым типам и реализуется в виде структуры Char.

Для символов класса Char используется двухбайтная кодировка Unicode, определяющая как мощность алфавита, включающего 2^{16} =65536 различных символов, так и порядок на алфавите. Если код символа s_1 меньше кода символа s_2, то символ s_1 предшествует в алфавите символу s_2 и справедливо неравенство $s_1 < s_2$. По символу нетрудно получить его код, поскольку существует неявное преобразование из типа Char в целочисленный тип int. Обратное преобразование также существует, но должно быть явным. Вот простой пример взаимных преобразований:

```
//преобразования int <-> char
Char s1 = 'Я', s2, s3;
int codes1, codes2, codes3;
codes1 = s1;
codes2 = codes1 - 1; codes3 = codes1 + 1;
s2 = (char)codes2; s3 = (char)codes3;
Console.WriteLine("Codes: {0}, {1}, {2}",
              codes1, codes2, codes3);
Console.WriteLine("Symbols: {0}, {1}, {2}",
              s1, s2, s3);
```

В результате работы этого фрагмента будут напечатаны символы «Я», «Ю», «а» и их коды – 1071, 1070, 1072.

Множество символов класса Char велико, что позволило включить в него большинство алфавитов естественных языков – латиницу, кириллицу, алфавиты иероглифических языков и другие. Для символов алфавитов естественных языков, как правило, используется плотная кодировка, сохраняющая принятый на алфавите порядок. Для кириллицы исключением являются буквы «Ё» и «ё», коды которых 1025 и 1105 находятся вне интервала, задающего коды для других букв алфавита.

Из символов класса Char строятся строки языка C#. Для построения строк предоставляется ряд взаимосвязанных классов – Char[], string, StringBuilder и другие. Если s – строка - объект одного из этих классов, то определен и объект s[i], принадлежащий классу Char, задающий i-й символ строки s. Каждый из объектов этих классов можно рассматривать как коллекцию символов. Приведу пример работы с объектами разных классов:

```
string str1 = "рококо";
StringBuilder str2 = new StringBuilder(str1);
char[] str3 = new char[6] {'р','о','к','о','к','о'};
char ch1 = str1[1], ch2 = str2[3], ch3 = str3[5];
Console.WriteLine( "строки: {0}, {1}, {2}",
       str1, str2.ToString(), new string(str3));
Console.WriteLine( "символы 2 - 4 - 6: {0}, {1}, {2}",
       ch1, ch2, ch3);
foreach (char ch in str1)
      Console.Write(ch.ToString());
Console.WriteLine();
foreach (char ch in str3)
      Console.Write(ch.ToString());
// foreach (char ch in str2)
//     Console.Write(ch.ToString());
```

В результате работы этого фрагмента все строки будут иметь значение «рококо», а все символы – «о». К массиву символов и к строке string можно применять цикл типа foreach, явно работая со строкой, как с коллекцией символов. К объектам StringBuilder этот цикл не применим.

В языке C# строку можно задавать массивом символов – Char[]. Этот способ хорош тогда, когда операции над строкой могут использовать преимущества массива – получать быстрый доступ к любому символу строки – читать и изменять его. Он удобен при работе со строками постоянной длины. Для такого представления отсутствуют встроенные операции над строками. Преодолеть этот недостаток можно, написав собственный класс CharArray, реализовав в нем все базисные операции. Из рассмотренных нами операций над строками только поиск вхождения подстроки и замена одной подстроки на другую, равную ей по длине, реализуется без всяких проблем. Все остальные операции – конкатенация, вставка, удаление, замена подстрок - требуют изменения размера массива, что требует в C# создания нового массива. Проще всего при реализации таких операций создавать новый массив, представляющий результат операции. Задача облегчается тем, что C# допускает динамические массивы, потому создание новой строки – массива требуемого размера не вызывает трудностей.

Основным способом представления строк в C# является класс string. В этом классе реализованы все базисные операции над строками. Он позволяет работать со строками переменной длины. Следует сказать, что ограничения, накладываемые на операции над строками класса string, являются довольно строгими – ни одна из операций не может изменять содержимое строки. Невозможно даже в строке s изменить значение символа s[i] – допускается только чтение символа, но не его замена. Класс string относится к так называемым неизменяемым (immutable) классам - все операции, требующие изменения строки, создают новую строку. Поскольку при объявлении строк string не требуется указывать их длину, то для конечного пользователя эти ограничения не являются обременительными. Другое дело, что при этом может происходить потеря эффективности. Если, например, изменения в строке состоят в том, что на каждом шаге цикла изменяется один символ в строке, то при больших циклах накладно каждый раз создавать новую строку.

В ситуациях, требующих многократного изменения значения строки, язык C# предлагает использовать специально спроектированный класс StringBuilder. Строки этого класса могут менять свои размеры в процессе работы с ними, не требуя создания новых объектов.

Классы C#, используемые для представления строк – Char[], string, StringBuilder, связаны между собой, и из объекта одного класса нетрудно получить объект другого класса. Конструктору класса string можно передать массив символов, создав тем самым объект класса string. Для обратного преобразования из string в Char[] следует вызвать метод ToCharArray, которым обладают объекты класса string. Достаточно вызвать метод ToString объекта StringBuilder для преобразования объекта класса StringBuilder в класс string. Обратное преобразование можно выполнить, передавая конструктору класса StringBuilder объект string. Приведенный выше пример демонстрирует некоторые из этих преобразований.

В зависимости от того, какие операции выполняются над строкой, следует использовать то или иное ее представление, переходя при необходимости от одного представления к другому. Какие же методы предлагают соответствующие классы?

Основная группа методов класса Char предназначена для классификации символа – определения его категории, выяснения, является ли символ цифрой, буквой, разделителем и так далее.

Класс Char[] является массивом. Он содержит методы, общие для массивов. Часть из них полезна при работе со строками. Например, метод Reverse позволяет обратить строку, а Sort отсортировать символы. Методы IndexOf и LastIndexOf позволяют найти первое и последнее вхождение символа в массив (строку). Если необходимы другие специальные операции, то, как уже говорилось, их следует реализовать самостоятельно.

Наиболее мощный набор методов для работы со строками предоставляет класс string. В частности этот класс реализует все базисные операции: вставки подстроки (Insert), удаления подстроки (Remove), замены подстроки (Replace), выделения подстроки (Substring), определения индекса вхождения (IndexOf). Весьма полезные взаимообратные операции реализуются методами Split и Join. Первый из них позволяет создать массив строк, разделяя исходную строку текста на фрагменты. Для разделения текста используются соответствующие разделители, стоящие между фрагментами. Например, если строка задает текст некоторой процедуры на языке программирования, то методом Split можно получить массив, элементы которого являются операторами этой процедуры. Предложение естественного языка методом Split можно разделить на отдельные слова. Метод Join выполняет обратную операцию, склеивая массив строк в одну строку с добавлением разделителей. Поскольку при расщеплении используется множество разделителей, а при сборке только один из них, то восстановление исходного текста в первоначальном виде не всегда возможно, но и не всегда требуется.

Класс StringBuilder, как уже отмечалось, выполняет более эффективно базисные операции над строками, требующие изменения размера строки. Но набор возможных операций со строкой у него значительно меньше, чем у объектов string.

Благодаря взаимным преобразованиям между классами для одного и того же текста можно иметь различные представления, и в зависимости от типа операции использовать то или иное представление. Более подробно познакомиться с классами, позволяющими работать с текстами, можно в соответствующих главах учебника [1]. А сейчас перейдем к задачам.

Задачи

4.1 Напишите процедуру, подсчитывающую частоту использования группы символов в заданном тексте. Проведите исследование произведений двух поэтов, подсчитав частоты использования частоты использования гласных и согласных, глухих и звонких согласных. Для представления текстов используйте класс Char[].

4.2 Напишите процедуру, подсчитывающую частоту использования группы символов в заданном тексте. Проведите исследование произведений двух поэтов, подсчитав частоты использования частоты использования гласных и согласных, глухих и звонких согласных. Для представления текстов используйте класс string.

4.3 Напишите процедуру, подсчитывающую частоту использования группы символов в заданном тексте. Проведите исследование произведений двух поэтов, подсчитав частоты использования частоты использования гласных и согласных, глухих и звонких согласных. Для представления текстов используйте класс StringBuilder.

4.4 Напишите процедуру, разделяющую исходный текст на предложения. Для представления текстов используйте класс Char[].

4.5 Напишите процедуру, разделяющую исходный текст на предложения. Для представления текстов используйте класс string.

4.6 Напишите процедуру, разделяющую исходный текст на предложения. Для представления текстов используйте класс StringBuilder.

4.7 Исходный текст представляет собой предложение. Напишите процедуру, разделяющую исходный текст на слова. Для представления текстов используйте класс Char[].

4.8 Исходный текст представляет собой предложение. Напишите процедуру, разделяющую исходный текст на слова. Для представления текстов используйте класс string.

4.9 Исходный текст представляет собой предложение. Напишите процедуру, разделяющую исходный текст на слова. Для представления текстов используйте класс StringBuffer.

4.10 Напишите процедуру IsIder, проверяющую является ли исходный текст правильно построенным идентификатором. Для представления текста используйте класс Char[].

4.11 Напишите процедуру IsIder, проверяющую является ли исходный текст правильно построенным идентификатором. Для представления текста используйте класс string.

4.12 Напишите процедуру IsIder, проверяющую является ли исходный текст правильно построенным идентификатором. Для представления текста используйте класс StringBuffer.

4.13 Напишите процедуру IsInt, проверяющую является ли исходный текст правильно построенным целым числом. Для представления текста используйте класс Char[].

4.14 Напишите процедуру IsInt, проверяющую является ли исходный текст правильно построенным целым числом. Для представления текста используйте класс string.

4.15 Напишите процедуру IsInt, проверяющую является ли исходный текст правильно построенным целым числом. Для представления текста используйте класс StringBuffer.

4.16 Напишите процедуру IsFloat, проверяющую является ли исходный текст правильно построенным числом с плавающей точкой. Для представления текста используйте класс Char[].

4.17 Напишите процедуру IsFloat, проверяющую является ли исходный текст правильно построенным числом с плавающей точкой. Для представления текста используйте класс string.

4.18 Напишите процедуру IsFloat, проверяющую является ли исходный текст правильно построенным числом с плавающей точкой. Для представления текста используйте класс StringBuffer.

4.19 Напишите процедуру IsNumber, проверяющую является ли исходный текст правильно построенным числом. Для представления текста используйте класс Char[].

4.20 Напишите процедуру IsNumber, проверяющую является ли исходный текст правильно построенным числом. Для представления текста используйте класс string.

4.21 Напишите процедуру IsNumber, проверяющую является ли исходный текст правильно построенным числом. Для представления текста используйте класс StringBuffer.

4.22 Исходный текст представляет описание класса на C#. Напишите процедуру, выделяющую из этого текста заголовки методов класса с предшествующими им тегами summary. Для представления текстов используйте класс Char[].

4.23 Исходный текст представляет описание класса на C#. Напишите процедуру, выделяющую из этого текста заголовки методов класса с предшествующими им тегами summary. Для представления текстов используйте класс string.

4.24 Исходный текст представляет описание класса на C#. Напишите процедуру, выделяющую из этого текста заголовки методов класса с предшествующими им тегами summary. Для представления текстов используйте класс StringBuffer.

4.25 Исходный текст представляет описание класса на C#. Напишите процедуру, удаляющую из этого текста комментарии. Для представления текстов используйте класс Char[].

4.26 Исходный текст представляет описание класса на C#. Напишите процедуру, удаляющую из этого текста комментарии. Для представления текстов используйте класс string.

4.27 Исходный текст представляет описание класса на C#. Напишите процедуру, удаляющую из этого текста комментарии. Для представления текстов используйте класс StringBuffer.

4.28 Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из методов класса. Для представления текстов используйте класс Char[].

4.29 Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из методов класса. Для представления текстов используйте класс string.

4.30 Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из методов класса. Для представления текстов используйте класс StringBuffer.

4.31 Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из полей класса. Для представления текстов используйте класс Char[].

4.32 Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из полей класса. Для представления текстов используйте класс string.

4.33 Исходный текст представляет описание класса на C#. Напишите процедуру, создающую массив строк, каждая из которых содержит описание одного из полей класса. Для представления текстов используйте класс StringBuffer.

4.34 Исходный текст задает оператор языка C#. Напишите процедуру, определяющую тип оператора. Для представления текстов используйте класс Char[].

4.35 Исходный текст задает оператор языка C#. Напишите процедуру, определяющую тип оператора. Для представления текстов используйте класс string.

4.36 Исходный текст задает оператор языка C#. Напишите процедуру, определяющую тип оператора. Для представления текстов используйте класс StringBuffer.

4.37 Напишите процедуру «Строгий Палиндром», определяющую является ли заданный текст палиндромом. Напомню, палиндромом называется симметричный текст, одинаково читаемый как слева направо, так и справа налево.

4.38 Напишите процедуру «Палиндром», определяющую является ли заданный текст палиндромом. При анализе текста:
- пробелы не учитываются;
- регистр не учитывается;
- буквы «е» и «ё», «и» и «й» считаются одинаковыми.
Фраза, которую Мальвина диктовала Буратино, - «А роза упала на лапу Азора» считается палиндромом.

4.39 Напишите процедуру «Слог», разбивающую слово на слоги. Предложите свой алгоритм. За основу возьмите следующие правила:
- две подряд идущие гласные рассматриваются как одна гласная;
- число слогов определяется числом гласных букв (с учетом предыдущего правила);
- Если n – число согласных между двумя соседними гласными, то n/2 согласных относятся к предыдущему слогу, а оставшиеся к следующему. Вот примеры нескольких разбиений в соответствии с этим алгоритмом: «слог», «сло - во», «прог - ноз», «транс – крип - ция», «зоо – ма – га – зин».

Проекты

4.40 Создайте класс CharArray для представления строк и интерфейс для работы с ним. Методы класса должны включать набор методов класса string. Внутреннее представление строки должно задаваться массивом символов – Char[]. Методы, изменяющие размер строки должны реализовываться функциями, как в классе string, создавая новый объект.

4.41 Создайте класс CharArray для представления строк и интерфейс для работы с ним. Методы класса должны включать набор методов класса string. Внутреннее представление строки должно задаваться массивом символов – Char[]. Методы, изменяющие размер строки должны реализовываться процедурами, как в классе StringBuffer.

4.42 Создайте класс MyText для работы с текстом. Методы этого класса должны выполнять должны различные операции над текстом. Примеры некоторых операций даны в задачах этого раздела. Операции над текстом должны, например, позволять получать коллекции абзацев, предложений, слов текста, получать абзац, предложение, слово по его номеру, разбивать слово на слоги.

4.43 Создайте класс MyProgramText для работы с текстом программ на языке C#. Методы этого класса должны выполнять должны различные операции над текстом программы. Примеры некоторых операций даны в задачах этого раздела.

Поиск и Сортировка

Задачи поиска и сортировки возникают в самых разных контекстах. Рассмотрим задачу поиска в следующей постановке. Дан массив Items с элементами типа (класса) T и элемент pattern типа T, называемый образцом. Необходимо определить, встречается ли образец в массиве и, если да, определить индекс его вхождения. Задача сортировки состоит в том, чтобы отсортировать массив Items. Предполагается, что тип T является упорядоченным типом, так что его элементы можно

сравнивать. Задачу можно конкретизировать, полагая, например, что T – это тип string, и рассматривать поиск и сортировку строковых массивов. Поскольку алгоритмы поиска и сортировки практически не зависят от типа T, то отложим конкретизацию типа настолько, насколько это возможно.

Поиск

Рассмотрим три классических алгоритма поиска – линейный поиск, линейный поиск с барьером, бинарный поиск в упорядоченном массиве.

Линейный поиск

Алгоритм линейного поиска предельно ясен. В цикле по числу элементов сравнивается очередной элемент массива с образцом. При нахождении элемента, совпадающего с образцом, поиск прекращается. Если цикл завершается без нахождения совпадений, то это означает, что в массиве нет искомого элемента. Время работы такого алгоритма линейно. В худшем случае придется сравнить образец со всеми элементами, в лучшем – с одним, в среднем – число сравнений равно n/2, где n – число элементов массива. У линейного поиска есть один недостаток. Если образец не присутствует в массиве, то без принятия предохранительных мер, поиск может выйти за границы массива, вследствие чего может возникнуть исключительная ситуация. В классическом варианте линейного поиска приходится на каждом шаге дополнительно проверять корректность значения текущего индекса.

Чтобы эта простая задача смотрелась интереснее, рассмотрим параметризованный алгоритм с параметром T, задающим тип элементов, и его реализацию на языке C#. Построим универсальный класс (класс с родовыми параметрами):

```
public class Service<T> where T:IComparable<T>
{
}
```

Класс Service имеет параметр T, на который наложено ограничение – класс T должен быть наследником интерфейса IComparable, а следовательно, реализовать метод CompareTo этого интерфейса. Содержательно это означает, что T является упорядоченным классом.

Класс Service будем рассматривать, как сервисный класс, предоставляющий другим клиентским классам некоторые сервисы, в частности возможность осуществлять поиск в массивах любого типа. Добавим в этот класс два статических метода, реализующих алгоритм линейного поиска:

```
/// <summary>
        /// Линейный поиск образца в массиве
        /// </summary>
        /// <param name="massiv">искомый массив</param>
        /// <param name="pattern">образец поиска</param>
        /// <returns>
        /// индекс первого элемента, совпадающего с образцом
        /// или -1, если образец не встречается в массиве
        /// </returns>
        public static int SearchPattern(T[] massiv, T pattern)
        {
            for (int i = 0; i < massiv.Length; i++)
               if (massiv[i].CompareTo(pattern)==0) return (i);
           return (-1);
        }
        /// <summary>
        /// Вариация линейного поиска образца в массиве
        /// </summary>
```

```
        /// <param name="massiv">искомый массив</param>
        /// <param name="pattern">образец поиска</param>
        /// <returns>
        /// индекс первого элемента, совпадающего с образцом
        /// или -1, если образец не встречается в массиве
        /// </returns>
        public static int SearchPattern1(T[] massiv, T pattern)
        {
            int i = 0;
            while((i<massiv.Length)&& (massiv[i].CompareTo(pattern)!=0))
                i++;
            if (i == massiv.Length) return (-1); else return (i);
        }
```

Две вариации линейного поиска отличаются лишь деталями. В первой из них проще условие цикла, но зато в тело цикла встроен оператор if, при выполнении условия которого завершается не только цикл, но и сам метод. В другой вариации усложнено условие цикла, но тело цикла совсем простое. В принципе тело цикла можно сделать пустым в этом варианте, внеся увеличение индекса во второе условие цикла. Но это уже трюк, снижающий ясность понимания программы. Трюкачество я не приветствую. Какую из эквивалентных версий выбирать – это дело программистского вкуса.

Поиск с барьером

Алгоритм линейного поиска можно упростить, избавившись от проверки дополнительного условия, если быть уверенным, что в массиве обязательно присутствует элемент, совпадающий с образцом. Иногда истинность этого условия следует из знания того, как строился массив и образец поиска. Но можно добиться выполнения этого условия принудительно, соорудив в массиве «барьер», препятствующий выходу поиска за границы массива. С этой целью массив расширяется на один элемент и в качестве последнего элемента записывается «барьер» - образец поиска. В этом случае поиск всегда найдет образец. Если образца нет среди «родных» элементов массива, то он встретится в конце в виде «барьера».

Для упрощения больше подходит вторая версия алгоритма линейного поиска. Приведу реализацию этой схемы:

```
    /// <summary>
        /// Линейный поиск с барьером
        /// Предусловие: В массиве существует элемент,
        /// совпадающий с образцом pattern
        /// </summary>
        /// <param name="massiv">искомый массив</param>
        /// <param name="pattern">образец поиска</param>
        /// <returns>
        /// индекс первого элемента, совпадающего с образцом
        /// </returns>
        public static int SearchBarrier(T[] massiv, T pattern)
        {
            int i = 0;
            while (massiv[i].CompareTo(pattern) != 0)
                i++;
```

```
            return (i);
        }
```

Заметьте, сам метод никаких барьеров не строит. Он лишь формулирует предусловие, требующее существование барьерного элемента в массиве. Ответственность за выполнения предусловия лежит на клиенте. Тот, кто вызывает метод, тот и должен заботиться о выполнении предусловия. Таковы принципы проектирования по контракту. Конечно, можно построить другую реализацию, где ответственность за построение барьера берет на себя сам метод.

Бинарный поиск

У этого метода поиска много синонимичных названий – метод деления пополам, двоичный или бинарный поиск, метод дихотомии. Все эти названия отражают тот приятный факт, что в упорядоченном массиве сравнение с одним элементом позволяет вдвое уменьшить число кандидатов. Для этого достаточно сравнить образец с элементом массива, стоящим в середине. Если образец совпадает с этим элементом, то элемент найден и поиск завершается. Если образец меньше срединного элемента, то размеры области поиска сокращаются вдвое - элемент может находиться лишь в первой половине массива. Если образец больше срединного элемента, он находится во второй половине массива. Введение двух параметров – start и finish, задающих границы области поиска, позволяет достаточно просто описать схему алгоритма. Алгоритм бинарного поиска намного эффективнее линейного поиска в особенности для больших массивов. Нетрудно понять, что для отсортированного массива из n элементов он требует не более чем $\log_2(n)$ сравнений образца с элементами массива. Вот его возможная реализация:

```
/// <summary>
/// Бинарный поиск образца в упорядоченном массиве
/// Предусловие: Массив упорядочен
/// </summary>
/// <param name="massiv">искомый массив</param>
/// <param name="pattern">образец поиска</param>
/// <returns>
/// индекс элемента, совпадающего с образцом,
/// но не обязательно индекс первого вхождения,
/// -1, если образец не встречается в массиве
/// </returns>
public static int BinSearch(T[] massiv, T pattern)
{
  int start = 0, finish = massiv.Length-1, mid = (start+finish)/2;
  while (start <= finish)
  {
     if (massiv[mid].CompareTo(pattern) == 0) return (mid);
     if(massiv[mid].CompareTo(pattern) == 1)
       finish = mid-1;
     else
      start = mid+1;
     mid = (start+finish)/2;
   }
   return(-1);
 }
```

Как клиентский класс может пользоваться сервисами универсального класса Service? Приведу три примера работы с методами класса. Наш первый клиент имеет массив целых чисел. Перед вызовом метода поиска он гарантирует упорядоченность массива и потому вполне законно вызывает метод бинарного поиска:

```
static void Test5()
        {
            Random rnd = new Random();
            const int n = 100;
            int[] ar1 = new int[n];
            for (int i = 0; i < n; i++)
                ar1[i] = rnd.Next(1, 1000);
            Array.Sort(ar1);
            for (int i = 0; i < n; i++)
                Console.Write(ar1[i].ToString() + ", ");
            int pat1 = rnd.Next(1, 10);
            int k = Service<int>.BinSearch(ar1, pat1);
            if (k != -1)
                Console.WriteLine("Образец pat1 = {0} найден в массиве!" +
                    "\nЭто элемент ar[{1}] = {2} ", pat1, k, ar1[k]);
            else
                Console.WriteLine("Образец pat1 ={0} не найден!", pat1);
        }
```

Второй клиент работает с массивом символов. Клиент гарантирует, что элементы массива и образец поиска принадлежат одному и тому же классу - классу char. Никаких других ограничений не накладывается. Для поиска вызывается обычный вариант линейного поиска:

```
static void Test6()
        {
            Random rnd = new Random();
            const int n = 100;
            const int st = 1072; //код символа "а"
            const int fin = 1103; //код символа "я"
            Char[] ar1 = new Char[n];
            for (int i = 0; i < n; i++)
                ar1[i] = (Char)rnd.Next(st, fin+1);
            for (int i = 0; i < n; i++)
                Console.Write(ar1[i].ToString() + ", ");
            char pat1 = (char)rnd.Next(st, fin);
            int k = Service<char>.SearchPattern(ar1, pat1);
            if (k != -1)
                Console.WriteLine("Образец pat1 = {0} найден в массиве!" +
                    "\nЭто элемент ar[{1}] = {2} ", pat1, k, ar1[k]);
            else
                Console.WriteLine("Образец pat1 ={0} не найден!", pat1);
```

```
}
```

Третий клиент работает с массивом строк класса string. Он предпочитает использовать метод поиска с барьером и сам заботится об организации барьера до вызова метода:

```
static void Test7()
        {
            int n;
            Console.WriteLine("Введите n - число элементов массива");
            n = Convert.ToInt32(Console.ReadLine());
            string[] ar1 = new string[n+1];
            for (int i = 0; i < n; i++)
            {
                Console.WriteLine("Введите строку - элемент" +
                " массива с номером {0}", i);
                ar1[i] = Console.ReadLine();
            }
            string pat1;
            Console.WriteLine("Введите строку - образец поиска");
            pat1 = Console.ReadLine();
            ar1[n] = pat1;
            //Выполнено условие метода поиска с барьером
            int k = Service<string>.SearchBarrier(ar1, pat1);
            if (k != n)
                Console.WriteLine("Образец pat1 = {0} найден в массиве!" +
                    "\nЭто элемент ar[{1}] = {2} ", pat1, k, ar1[k]);
            else
                Console.WriteLine("Образец pat1 ={0} не найден!", pat1);
        }
```

Для более подробного знакомства с универсальными классами рекомендую прочесть главу 22 учебника [1].

Задачи

4.44 Напишите три процедуры поиска (линейного, линейного с барьером, бинарного) для работы с классом double.

4.45 Напишите три процедуры поиска (линейного, линейного с барьером, бинарного) для работы с классом StringBuilder.

4.46 Напишите три процедуры поиска (линейного, линейного с барьером, бинарного) для работы с классом int.

4.47 Напишите три процедуры поиска (линейного, линейного с барьером, бинарного) для работы с классом string.

4.48 Напишите три процедуры поиска (линейного, линейного с барьером, бинарного) для работы с классом Person. Класс Person определите сами.

4.49 На основе приведенного описания класса Service создайте собственный универсальный класс, включающий различные варианты метода поиска. Создайте Windows-интерфейс для работы с этим классом.

4.50 Создайте DLL на основе класса Service и постройте проекты – консольный и Windows, в которых есть классы, являющиеся клиентами класса Service.

Сортировка

Задача сортировки формулируется достаточно просто. Дан массив Ar с элементами типа T. Тип (класс) T является упорядоченным типом, так что для него определена операция сравнения элементов. Отсортировать массив можно по возрастанию или по убыванию. В первом случае для всех элементов массива выполняется условие Ar[i] <= Ar[i+1], во-втором – справедливо условие Ar[i] >= Ar[i+1]. Упорядочив массив по возрастанию, можно вызвать затем метод Reverse для изменения порядка сортировки. Порядок сортировки можно задавать как параметр метода, что сказывается лишь на операции сравнения элементов - «больше» или «меньше».

Методов сортировки великое множество. Классическим трудом является третий том «Искусства программирования» Д. Кнута [Кнут], который так и называется «Сортировки». Одним из основных критериев классификации методов сортировки является сложность метода сортировки – временная и емкостная – T(n) и P(n). В первом случае нас интересует время сортировки произвольного массива из n элементов, во-втором - дополнительная память, требуемая в процессе сортировки. Говоря о времени сортировки, можно рассматривать минимальное, максимальное или среднее время сортировки. Обычно под временем сортировки подразумевается число требуемых операций, которые в свою очередь разделяются на операции сравнения элементов и операции обмена элементами, когда два элемента Ar[i] и Ar[j] обмениваются местами.

Сортировка за линейное время

За линейное время можно сортировать массивы, элементы которых принадлежат фиксированному числу видов. Рассмотрим алгоритмы сортировки отдельно для случая двух, трех и четырех видов. На практике очень часто возникает необходимость делить некоторую совокупность на две части – «красных» и «белых», «мужчин» и «женщин». Случаю трех видов посвящена известная задача Дейкстры о «голландском национальном флаге». Вот как Дейкстра формулирует эту задачу [Структурное]. Элементы в массиве принадлежат трем видам – красные, белые и синие. Требуется отсортировать массив в порядке следования этих цветов во флаге Голландии. Поскольку цвета флагов России и Голландии совпадают, то для нас приятнее сортировать массив в порядке следования этих цветов во флаге России - белые, синие, красные элементы. Задачи, где в массиве 4 вида элементов, встречаются не менее часто. Например молекула ДНК, как уже упоминалось, представляется строкой (массивом char) в алфавите из четырех символов. Этот массив иногда требуется отсортировать в некотором заданном порядке следования символов.

С помощью рис. 4_1 поясним идею эффективного алгоритма сортировки, не требующего дополнительной памяти и сортирующего массив при числе видов m, равном 2, 3 или 4, за один проход по массиву.

Рис. 4.1 Инвариантное расположение зон массива при m = 2, 3, 4

Рассмотрим вначале случай m =2, когда в массиве есть элементы только двух видов. Для простоты будем называть их белыми и красными. Все множество индексов элементов массива разделим на три непересекающихся подмножества – 0-зона, содержащая только белые элементы, 1-зона для красных элементов и непроверенная зона для тех элементов, чей цвет не установлен. Инвариантом, поддерживаемым в проектируемом алгоритме, будет расположение зон, показанное на рис. 4_1. Массив отсортирован, когда непроверенная зона становится пустой. В этом идея алгоритма - поддерживать истинность инварианта, сокращая непроверенную зону. В начальном состоянии 0-зона и 1-зона пусты, а непроверенная зона занимает все множество индексов, так что ее начальная граница Start = 0, а граница Finish = n-1. В начальном состоянии инвариант считается истинным. Основной и единственный проход по циклу выполняется по непроверенной зоне до тех пор, пока эта зона не станет пустой (или состоять из одного элемента). Проверка элементов начинается с левого конца непроверенной зоны. До тех пор, пока очередной элемент является белым, расширяется 0-зона и соответственно сокращается непроверенная зона – значение границы Start увеличивается на 1. В тот момент, когда встречается красный элемент, проверка прекращается и запускается аналогичный цикл, но теперь уже с правого конца непроверенной зоны. Когда на правом конце обнаруживается белый элемент, происходит обмен значениями на двух концах непроверенной зоны. Обмен восстанавливает истинность инварианта. По завершении цикла непроверенная зона становится пустой, так что 1-зона с красными элементами следует сразу за 0-зоной с белыми элементами и массив отсортирован.

Алгоритм легко обобщается на случай 3-х и 4-х видов элементов. В случае трех элементов появляется дополнительная зона для синих элементов. Инвариантная ситуация расположения зон показана на рис. 4_1. Внешний цикл устроен аналогичным образом, как и в случае m = 2. Когда на левом конце обнаруживается элемент, не принадлежащий 1-зоне, то анализируются две возможные ситуации. Если элемент белый, то он меняется местами с синим элементом, стоящим на границе между белой и синей зонами. Инвариант восстанавливается и продолжается проверка элементов на левом конце непроверенной зоны. Если же на левом конце обнаруживается красный элемент, принадлежащий зоне 3, то непроверенная зона начинает анализироваться с правого конца. Красная зона расширяется до тех пор, пока не встретится элемент – синий или белый. Синий элемент потребует одного обмена, а белый – двух обменов для поддержания инварианта.

Случай 4-х элементов восстанавливает симметрию расположения зон – по две зоны справа и слева от непроверенной зоны. В этом случае анализ на левом и правом конце непроверенной зоны выполняется одинаковым образом.

Приведу пример реализации алгоритма сортировки для случая классификации двух видов. Главная цель примера не столько в том, чтобы продемонстрировать сам алгоритм – он достаточно прост, а в том, чтобы показать, как на C# написать универсальный алгоритм для элементов любого типа и с разными названиями видов. Хочется, чтобы алгоритм можно было использовать для классификации элементов 0 и 1, «мужчин» и «женщин», «красных» и «белых».

Добавим в ранее построенный класс Service с родовым параметром T два поля:

```
//Поля и методы сортировки видов
        /// <summary>
        /// число видов
        /// </summary>
        static int m;
        /// <summary>
        /// массив, задающий возможные виды элементов
        /// </summary>
        static T[] Spacimen;
```

Прежде, чем вызывать метод сортировки, клиентский класс должен будет позаботиться об уведомлении класса Service, какие виды элементов могут встретиться в сортируемом массиве. Для передачи этой информации в классе Service есть специальный метод:

```
    /// <summary>
        /// Задать информацию, необходимую методам видовой сортировки
        /// </summary>
        /// <param name="m1"> число видов, которым принадлежат
        /// элементы в сортируемом массиве</param>
        /// <param name=" Spacimen1"> массив значений видов </param>
        public static void InitSpacimen(int m1, T[]Spacimen1)
        {
            m = m1;
            Spacimen = Spacimen1;
        }
```

Метод InitSpacimen должен вызываться один раз и предшествовать любому вызову метода сортировки. Приведем теперь текст самого метода сортировки:

```
 /// <summary>
/// Сортировать массив,
/// Предусловие: Элементы массива принадлежат двум видам,
```

```
/// заданным в массиве Spacimen
/// </summary>
/// <param name="ar">сортируемый массив</param>
public static void SortTwoKinds(T[] ar)
{
   int start = 0, finish = ar.Length - 1;
   T val1 = Spacimen[0], val2 = Spacimen[1];
   while (start < finish)
   {
        while ((start < finish) && (ar[start].CompareTo(val1) == 0))
            start++;
        while ((start < finish) && (ar[finish].CompareTo(val2) == 0))
            finish--;
        //обмен
        T temp = ar[start]; ar[start]= ar[finish];
        ar[finish] = temp;
        start++; finish--;
    }
}
```

Тот факт, что элементы сортируемого массива могут быть экземплярами произвольного класса T, обеспечивается тем, что класс Service является универсальным классом с параметром T. Тот факт, что виды элементов могут иметь произвольные значения, обеспечивается тем, что классу Service предварительно передается массив Spacimen, хранящий информацию о возможных видах.

Покажем теперь, как клиентский класс может вызывать метод сортировки в конкретной ситуации:

```
static void Test8()
        {
            //Два вида элементов
            int m = 2;
            string[] cand = new string[m];
            cand[0] = "red"; cand[1] = "white";
            Service<string>.InitSpacimen(m, cand);
            //Моделирование массива ar
            Random rnd = new Random();
            const int n = 10;
            string[] ar = new string[n];
            for (int ind, i = 0; i < n; i++)
            {
               ind = rnd.Next(0, m);
               ar[i] = cand[ind];
            }
            for (int i = 0; i < n; i++)
```

```
            Console.Write(ar[i] + " ");
        Console.WriteLine();
        //Сортировка массива ar
        Service<string>.SortTwoKinds(ar);
        for (int i = 0; i < n; i++)
            Console.Write(ar[i]+ " ");
        Console.WriteLine();
    }
```

Рассмотренный алгоритм вряд ли целесообразно обобщать на случай, когда число видов более четырех. Тем не менее, можно построить эффективный алгоритм, когда число видов m известно и оно заведомо меньше n – числа элементов в массиве. В этом случае можно построить алгоритм сортировки, работающий за время O(n*log(m)). Идея алгоритма достаточно прозрачна. За один проход по сортируемому массиву посчитаем, сколько элементов каждого вида находится в массиве. Для хранения этой информации потребуется дополнительный массив Counts размерности m, элемент которого Counts[i] задает число элементов вида Spacimen[i] в сортируемом массиве. Используя этот массив и массив Spacimen можно за время O(n) заполнить сортируемый массив элементами, следующими в нужном порядке. Основное время алгоритма уходит на формирование массива Counts, поскольку для каждого элемента нужно определить какому виду он принадлежит, что требует проведения поиска в массиве Spacimen. Для поиска можно использовать метод SearchBarrier, на что уйдет время порядка O(m). Время можно сократить до $O(\log_2(m))$, если использовать бинарный поиск, предварительно отсортировав массив Spacimen. Заметьте, на сортировку и хранение отсортированного массива понадобится дополнительное время порядка $O(m*\log_2(m))$ и дополнительная память.

Когда число видов m сравнимо по порядку с числом элементов n, то алгоритм становится эквивалентным по сложности классическим алгоритмам сортировки. Этот способ сортировки иногда называют сортировкой «черпаками». Если в процессе сортировки нужно хранить не только ключи, но и связанную с ними информацию, например указатели на объекты, то тогда действительно нельзя обойтись подсчетом числа элементов одного вида, поскольку для каждого из элементов нужно сохранять связанную с ним информацию. В этом случае для каждого вида элементов нужно иметь свой «черпак» - массив, хранящий элементы данного вида. Алгоритм сортировки, как и в выше описанном случае, состоит из двух этапов. На первом – заполняются черпаки, на втором – данные из черпаков сливаются в общий массив.

В выше приведенном примере разбиение элементов массива на виды осуществлялось с помощью представителей – к одному виду относились элементы с одним и тем же значением. Общий способ классификации состоит в задании классифицирующей функции – int Classification(int m, T item), которая для каждого элемента item возвращает число, задающее его вид. Аргумент m этой функции указывает максимальное число видов для этой функции классификации. Обычно предполагается, что значение, возвращаемое функцией, является целым числом в диапазоне [0, m-1], задавая номер вида.

В алгоритме быстрой сортировки Хоара требуется все множество элементов отсортировать на два подмножества. Вначале располагаются элементы, меньшие некоторого выбранного элемента, затем все оставшиеся. Для решения этой задачи применяется алгоритм SortTwoKinds с простой функцией классификации. Понятно, что в конкретных задачах могут встречаться достаточно сложные функции классификации, так что эту функцию полезно рассматривать как параметр функции сортировки. Алгоритм сортировки можно построить так, чтобы он не зависел ни от типа сортируемых элементов (это параметр алгоритма), ни от типа функции, задающей классификацию элементов, - это еще один параметр алгоритма.

Рекомендую, прежде чем решать задачи, прочитать лекцию 22 учебника [1], посвященную универсальности – классам с родовыми параметрами, и лекцию 20, в которой рассматриваются функциональные типы, делегаты и функции высших порядков. Если мы хотим передавать функцию классификации в качестве параметра функции (процедуре) сортировки, то последняя задается функцией высшего порядка.

Задачи

4.51 Напишите процедуру Reverse, меняющую порядок элементов массива.

4.52 Напишите функцию FReverse, возвращающую массив с обратным порядком следования элементов массива, заданного в качестве аргумента.

4.53 Напишите процедуру CreateSpecimen, создающую по массиву ar массив представителей. Массив представителей отличается от исходного массива тем, что в нем нет повторяющихся элементов. В том случае, когда нет повторений в исходном массиве, оба массива будут совпадать.

4.54 Напишите функцию FCreateSpecimen с аргументом, заданным массивом ar, возвращающую массив представителей. Массив представителей отличается от исходного массива ar тем, что в нем нет повторяющихся элементов. В том случае, когда нет повторений в исходном массиве, оба массива будут совпадать.

4.55 Дан массив ar и массив его представителей Spacimen. Напишите процедуру HowMany, вычисляющую целочисленный массив той же размерности, что и массив Spacimen. Элемент этого массива с индексом k должен быть равным числу элементов вида Spacimen[k], содержащихся в массиве ar.

4.56 Дан массив ar и массив его представителей Spacimen. Напишите функцию FHowMany, возвращающую целочисленный массив той же размерности, что и массив Spacimen. Элемент возвращаемого массива с индексом k должен быть равным числу элементов вида Spacimen[k], содержащихся в массиве ar.

4.57 Дан массив ar и функция классификации – int Classification(int m, T item). Напишите процедуру HowMany, вычисляющую целочисленный массив. Элемент этого массива с индексом k должен быть равным числу элементов вида k, содержащихся в массиве ar. Функция классификации должна передаваться процедуре HowMany в качестве параметра.

4.58 Дан массив ar и функция классификации – int Classification(int m, T item). Напишите функцию HowMany, вычисляющую целочисленный массив. Элемент этого массива с индексом k должен быть равным числу элементов вида k, содержащихся в массиве ar. Функция классификации должна передаваться функции HowMany в качестве параметра.

4.59 Дан массив представителей Spacimen - массив без повторяющихся элементов и массив Counts, элементы которого задают для каждого представителя число его повторений. Напишите процедуру SortAr, создающую массив с повторениями, где каждый представитель повторяется заданное число раз. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов создаваемого массива.

4.60 Дан массив представителей Spacimen - массив без повторяющихся элементов и массив Counts, элементы которого задают для каждого представителя число его повторений. Напишите функцию FSortAr, возвращающую массив с повторениями, где каждый представитель повторяется заданное число раз. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов создаваемого массива.

4.61 Напишите процедуру SortTwoKinds – процедуру сортировки массива типа string, содержащего элементы двух видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.62 Напишите процедуру SortTwoKinds – процедуру сортировки массива типа string, содержащего элементы двух видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Деление элементов на два вида задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.63 Напишите процедуру SortThreeKinds – процедуру сортировки массива типа string, содержащего элементы трех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.64 Напишите процедуру SortThreeKinds – процедуру сортировки массива типа string, содержащего элементы трех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Деление элементов на три вида задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.65 Напишите процедуру SortFourKinds – процедуру сортировки массива типа string, содержащего элементы четырех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.66 Напишите процедуру SortFourKinds – процедуру сортировки массива типа string, содержащего элементы четырех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Деление элементов на четыре вида задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.67 Напишите процедуру SortMKinds1 – процедуру сортировки массива типа string, содержащего элементы m видов. Алгоритм должен выполняться за время порядка O(n*m), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.68 Напишите процедуру SortMKinds2 – процедуру сортировки массива типа string, содержащего элементы m видов. Алгоритм должен выполняться за время порядка $O(n*\log_2(m))$, где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.69 Напишите процедуру SortMKinds – процедуру сортировки массива типа string, содержащего элементы m видов. Деление элементов на m видов задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.70 Напишите процедуру SortKinds – процедуру сортировки массива типа string. Процедура должна вначале определить число представителей в сортируемом массиве. Затем в зависимости от полученного значения m вызвать одну из подходящих процедур сортировки.

4.71 Напишите процедуру SortKinds1 – процедуру сортировки массива типа string, которой в качестве параметра передается функция классификации. Аргумент m функции классификации определяет число видов сортируемого массива. В зависимости от значения m следует вызвать одну из подходящих процедур сортировки.

4.72 Напишите универсальную (с параметром типа T) процедуру SortTwoKinds – процедуру сортировки массива типа T, содержащего элементы двух видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.73 Напишите универсальную (с параметром типа T) процедуру SortTwoKinds – процедуру сортировки массива типа T, содержащего элементы двух видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Деление элементов на два вида задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.74 Напишите универсальную (с параметром типа T) процедуру SortThreeKinds – процедуру сортировки массива типа T, содержащего элементы трех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.75 Напишите универсальную (с параметром типа T) процедуру SortThreeKinds – процедуру сортировки массива типа T, содержащего элементы трех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Деление элементов на три вида задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.76 Напишите универсальную (с параметром типа T) процедуру SortFourKinds – процедуру сортировки массива типа T, содержащего элементы четырех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.77 Напишите универсальную (с параметром типа T) процедуру SortFourKinds – процедуру сортировки массива типа T, содержащего элементы четырех видов. Алгоритм должен выполняться за время порядка O(n), где n – это число элементов массива. Деление элементов на четыре вида задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.78 Напишите универсальную (с параметром типа T) процедуру SortMKinds1 – процедуру сортировки массива типа T, содержащего элементы m видов. Алгоритм должен выполняться за время порядка O(n*m), где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.79 Напишите универсальную (с параметром типа T) процедуру SortMKinds2 – процедуру сортировки массива типа T, содержащего элементы m видов. Алгоритм должен выполняться за время порядка $O(n*\log_2(m))$, где n – это число элементов массива. Виды элементов задаются массивом представителей.

4.80 Напишите универсальную (с параметром типа T) процедуру SortMKinds – процедуру сортировки массива типа T, содержащего элементы m видов. Деление элементов на m видов задается соответствующей функцией классификации, передаваемой процедуре сортировки в качестве параметра.

4.81 Напишите универсальную (с параметром типа T) процедуру SortKinds – процедуру сортировки массива типа T. Процедура должна вначале определить число представителей в сортируемом массиве. Затем в зависимости от полученного значения m вызвать одну из подходящих процедур сортировки.

4.82 Напишите универсальную (с параметром типа T) процедуру SortKinds1 – процедуру сортировки массива типа T, которой в качестве параметра передается функция классификации. Аргумент m функции классификации определяет число видов сортируемого массива. В зависимости от значения m следует вызвать одну из подходящих процедур сортировки.

Проекты

4.83 На основе рассмотренного в этом разделе класса Service постройте собственный класс, расширив его методами сортировки для разных значений числа видов m. Постройте Windows-интерфейс, позволяющий клиентам класса Service вызывать его методы для массивов разных типов.

4.84 Постройте проект, позволяющий сравнивать время, затрачиваемое компьютером на сортировку массива, когда применяются методы универсального класса Service и аналогичные методы, написанные для конкретного типа данных. Цель проекта – показать возможные потери эффективности, как плата за универсальный характер методов сортировки.

4.85 Постройте проект, позволяющий сравнивать время, затрачиваемое компьютером на сортировку массива с элементами двух, трех, четырех и m видов, при использовании методов класса Service и метода быстрой сортировки Хоара, встроенной в библиотеку FCL.

Методы сортировки за время порядка $O(n^2)$

В ситуациях, когда приходится сортировать массивы небольшой размерности, разумно пользоваться простыми методами сортировки. Нетрудно доказать, что в тех случаях, когда на элементы массивов не накладываются никакие дополнительные ограничения кроме упорядоченности, то наилучшие методы сортировки в среднем требуют времени работы порядка $O(n* \log_2(n))$. Существует целый набор таких эффективных по порядку методов сортировки, различающихся сложностью реализации. Простые и естественные способы сортировки требуют, как правило, времени работы порядка $O(n^2)$. Эти методы сортируют массивы небольшой размерности быстрее, чем их соперники - более эффективные по порядку, но и более сложные методы сортировки. Но понятно, что у каждого из

квадратичных методов сортировки есть свой предел, то максимальное значение n, после которого эффективные методы со сложностью $O(n*\log_2(n))$ начинают работать быстрее.

Рассмотрим алгоритмы сортировки с квадратичной сложностью, начиная с простейших, интуитивно понятных.

Сортировка SortMin (SortMax) на основе нахождения минимального (максимального) элемента массива

Две сортировки минимумами и максимумами являются вариациями алгоритма, называемого часто «простым выбором». Идея алгоритма прозрачна и состоит в том, чтобы найти минимальный (максимальный) элемент массива и поставить его на первое (последнее) место. Затем применить тот же прием к массиву без первого (последнего) элемента, повторяя эту схему, пока оставшаяся часть массива не будет состоять из одного элемента.

Сортировка SortMinMax

Эта сортировка является слегка улучшенным вариантом предыдущей сортировки, когда минимальный и максимальный элементы находятся одновременно. Они и меняются местами с первым и соответственно последним элементами текущей части массива. Повышение эффективности достигается за счет того, что одновременный поиск максимума и минимума можно выполнить быстрее, чем при раздельном их поиске.

Сортировка SortBubble (SortBall) – пузырьковая сортировка и сортировка «тяжелыми шариками»

Эти две вариации одного алгоритма сортировки относят к классу «обменных сортировок». В каждом алгоритме сортировки присутствуют операции сравнения элементов и обмена элементов. Но алгоритмы могут отличаться тем, какие операции превалируют в реализации алгоритма. В сортировках прямого выбора минимумами и максимумами обмен выполняется только после того, как сделан выбор нужного элемента, требующий многократных проверок. В обменных сортировках обмен элементов является основной операцией в процессе сортировки. И те и другие методы имеют свои достоинства и соответственно недостатки. Операции обмена обычно более дорогие (требуют больше времени), чем операции сравнения. В этом преимущество методов прямого выбора. Но в обменных сортировках за один проход не только один элемент становится на свое место, но и другие элементы стремятся занять свои места, что позволяет ускорить сортировку.

Идея алгоритма пузырьковой сортировки, принадлежащей классу обменных сортировок, состоит в том, чтобы, начиная с конца массива, сравнивать два соседних элемента и, если нарушается упорядоченность, производить обмен элементами – более легкий элемент меняется местами со своим соседом. Очевидно, что при первом проходе массива минимальный элемент, как самый легкий всплывет наверх, подобно пузырьку воздуха, и станет на первое место. Важно то, что при этом будут всплывать, приближаясь к своим законным местам и другие легкие элементы. Обменные сортировки хорошо работают на почти упорядоченных массивах. Достоинство алгоритма еще и в том, что он позволяет собрать важную информацию - на каждом проходе можно подсчитывать число обменов. Если оно равно 0, то массив уже упорядочен, и сортировку можно прекращать.

Алгоритм «тяжелого шарика» является симметричной вариацией пузырьковой сортировки. Работа начинается с начала массива и в процессе обмена вниз опускаются тяжелые элементы, так что на первом проходе максимальный элемент станет на последнее место.

Сортировка SortShaker – шейкерная сортировка

Эта сортировка является слегка улучшенным вариантом предыдущей сортировки, когда на одном проходе применяется алгоритм пузырьковой сортировки, на следующем – алгоритм тяжелого шарика. Поочередное применение приводит к тому, что подъем легких элементов и опускание тяжелых выполняется равномерно, что в ряде случаев способствует ускорению процесса сортировки. Хотя сама идея красивая, но трудно найти какое либо математическое обоснование эффективности шейкерной сортировки в сравнении с обычным «пузырьком».

Сортировка SortInsert – сортировка вставками

Сортировка вставками – это еще один класс простых методов сортировки. Рассмотрим простейший вариант этого способа сортировки. Чтобы описать идею алгоритма, предположим вначале, что массив уже упорядочен за исключением последнего элемента. Тогда задача сводится к тому, чтобы

вставить этот элемент в нужную позицию. Это можно сделать двояко. Во-первых, можно применить алгоритм, подобный «пузырьку», выполняя обмен, пока последний элемент не «всплывет» на свое место. В лучшем случае не придется делать ни одного обмена, если последний элемент – это максимальный элемент и стоит уже на своем месте. В худшем случае придется сделать n сравнений и n обменов, если последний элемент – это минимальный элемент массива. В среднем – истина посредине. Другой способ состоит в том, чтобы воспользоваться упорядоченностью массива. В этом случае место вставки, используя алгоритм бинарного поиска, можно найти значительно быстрее за log(n) операций. К сожалению, нельзя избежать сдвига всех элементов массива ниже точки вставки.

Понятно, как идею вставки распространить на весь массив. Рассматриваем начальную часть массива, как уже упорядоченную. Поскольку часть массива, состоящая из одного первого элемента упорядочена по определению, то вначале вставляем в эту упорядоченную часть второй элемент массива, затем третий, пока не дойдем до последнего.

Сортировка SortShell – улучшенный вариант сортировки вставками

Сортировка, предложенная Шеллом, сложнее в реализации, чем ранее рассмотренные простые методы. Более того, интуитивно она наименее понятна и при знакомстве с ней кажется странным, что она может давать хорошие результаты. Но эта неочевидность характерна и для других эффективных методов сортировки. Та же быстрая сортировка Хоара далеко не очевидна, особенно когда появилась ее первоначальная версия, не использующая рекурсию.

В чем идея алгоритма сортировки Шелла? Зададим последовательность чисел:

$h_k, h_{k-1}\ldots, h_1$

Эта последовательность должна быть убывающей и заканчиваться значением $h_1 = 1$. Любая последовательность чисел с такими свойствами является подходящей и гарантирует сортировку массива. До сих пор неизвестно, какая последовательность является наилучшей. Желательным свойством последовательности является взаимная простота чисел h_i. Другое свойство требует, чтобы каждое из них примерно в два раза было меньше предыдущего. Хорошим выбором считается последовательность чисел, в которой $h_i = 2^i - 1$. Первый член последовательности h_k подбирается так, чтобы он был примерно равен n/2, где n – размерность массива. При n = 1000 последовательность может быть такой: 511, 255, 127, 63, 31, 15, 7, 3, 1.

Внешний цикл в сортировке Шелла – это цикл по последовательности h_i. Каждый член этой последовательности делит элементы массива на группы, состоящие из элементов массива, отстоящих друг от друга на расстоянии i. Для выбранной нами последовательности первый член последовательности создает большое число групп – h_k групп, в каждой из которых не более двух элементов. На следующем шаге число групп уменьшается, а число элементов в них увеличивается. На последнем шаге при $h_1 = 1$ возникает одна группа, в которую входят все элементы. Суть алгоритма в том, что к каждой возникающей группе независимо применяется обычный алгоритм вставки, сортирующий каждую группу. В чем же суть алгоритма. Ведь на последнем этапе ко всему массиву применяется обычный алгоритм вставки. За счет чего же достигается эффективность алгоритма? Дело в том, что к последнему этапу массив будет «почти упорядочен», а на таких массивах алгоритм вставки работает крайне быстро. Действительно, если массив упорядочен, то алгоритм SortInsert с «пузырьковым обменом» выполнит всего лишь n операций сравнения и ему вообще не потребуются операции обмена.

Сортировка Шелла хороша еще и тем, что она является прекрасным примером, требующим изощренного программирования. При написании реализации этого метода сортировки крайне полезно выписать инварианты циклов и использовать приемы доказательного программирования. Рекомендую прочесть лекцию 10 учебника [1] и разобрать пример быстрой сортировки Хоара, где реализация метода сортировки сопровождается записью инвариантов цикла и неформальным доказательством корректности реализации.

Задачи

4.86 Напишите процедуры SortMin и SortMax для некоторого конкретного типа элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.87 Напишите процедуру SortMinMax для некоторого конкретного типа элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.88 Напишите процедуру SortBubble и SortBall для некоторого конкретного типа элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.89 Напишите процедуру SortShaker для некоторого конкретного типа элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.90 Напишите процедуру SortInsert для некоторого конкретного типа элементов массива. Для вставки элемента используйте идею «пузырькового» обмена. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.91 Напишите процедуру SortInsert1 для некоторого конкретного типа элементов массива. Для вставки элемента используйте идею бинарного поиска. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.92 Напишите процедуру SortShell для некоторого конкретного типа элементов массива. Сопроводите реализацию явным выписыванием инвариантов для всех циклов и неформальным доказательством корректности алгоритма сортировки. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.93 Напишите универсальные процедуры SortMin и SortMax для произвольного типа T элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.94 Напишите универсальную процедуру SortMinMax для произвольного типа T элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.95 Напишите универсальные процедуры SortBubble и SortBall для произвольного типа T элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.96 Напишите универсальную процедуру SortShaker для произвольного типа T элементов массива. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.97 Напишите универсальную процедуру SortInsert для произвольного типа T элементов массива. Для вставки элемента используйте идею «пузырькового» обмена. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.98 Напишите универсальную процедуру SortInsert1 для произвольного типа T элементов массива. Для вставки элемента используйте идею бинарного поиска. Постройте график времени работы процедуры в зависимости от размерности массива – n.

4.99 Напишите универсальную процедуру SortShell для произвольного типа T элементов массива. Сопроводите реализацию явным выписыванием инвариантов для всех циклов и неформальным доказательством корректности алгоритма сортировки. Постройте график времени работы процедуры в зависимости от размерности массива – n.

Проекты

4.100 Постройте класс Sorting, содержащий методы сортировки, и класс Analyze, позволяющий анализировать время работы методов сортировки на одних и тех же массивах. Интерфейс проекта должен поддерживать представление результатов анализа в виде графиков.

4.101 Постройте универсальный класс Sorting, содержащий методы сортировки массивов произвольного типа, и класс Analyze, позволяющий анализировать время работы методов сортировки на одних и тех же массивах. Интерфейс проекта должен поддерживать представление результатов анализа в виде графиков.

Рекурсивные методы сортировки за время порядка $O(n*\log_2(n))$

Большинство эффективных методов сортировки описываются в виде рекурсивных алгоритмов и реализуются как рекурсивные процедуры. Хотя реализация рекурсивных процедур требует от разработчиков трансляторов использования стековой памяти, но эта техника сегодня настолько отработана, что потери на организацию рекурсии становятся незначительными в сравнении с теми преимуществами, которые дают рекурсивные алгоритмы. Для небольших массивов конечно квадратичные алгоритмы требуют меньше времени, но чем больше размер массива, тем эффективнее становится применение рекурсивных методов.

Рекурсивное описание алгоритма, как правило, значительно короче нерекурсивного описания. Но оно всегда оставляет впечатление некоторого трюка. Казалось бы умеем решать задачу только для самого простого случая, затем делаем некоторый фокус и задача оказывается решенной и для сложных случаев. Описать рекурсивное решение просто, гораздо сложнее повторить вычисления согласно рекурсивному алгоритму. Но, как известно, описанием алгоритмов занимается человек, а выполнением - компьютер. При анализе алгоритма, отладке соответствующей процедуры и человеку, конечно же, приходится выполнять рекурсивные вычисления.

Разберем несколько рекурсивных алгоритмов сортировки.

Сортировка слиянием

Задача сортировки массива решается совсем просто, когда массив состоит из одного элемента. В этом случае ничего делать не нужно, - такой массив считается отсортированным по определению. Это позволяет написать следующую схему рекурсивного алгоритма сортировки слиянием – RSortMerge(start, finish):

```
if(n > 1)
{
  RSortMerge(start, mid); //Отсортируй первую половину массива
  RSortMerge(mid+1, finish); //Отсортируй вторую половину массива
  Merge();  //Слей две половины в единый массив, сохраняя упорядоченность
}
```

Задача сортировки свелась к значительно более простой задаче слияния двух упорядоченных массивов (частей массива) в единый массив.

Нетрудно доказать, что этот алгоритм будет выполняться за время порядка O(n*log(n)). Вспомним базисную теорему о рекурсии, приведенную в главе 3 (теорема 3.1). Она утверждает, что, если исходную задачу размерности n удается разбить на две подзадачи размерности n/2, а затем из решений подзадач за линейное время получить решение исходной задачи, то сложность решения будет иметь порядок O(n*log(n)). На этом и построен алгоритм сортировки слиянием. Разбиение исходной задачи на две подзадачи одинаковой размерности не вызывает никаких сложностей – массив делим пополам. Чтобы выполнить слияние за линейное время, процедуре Merge необходима дополнительная память. Во избежание дополнительных программистских сложностей сразу же после слияния можно отсортированный массив переписать в исходный. Обратная перепись сохраняет линейное время работы слияния.

Алгоритм процедуры слияния Merge достаточно прост. Обычно вводят три целочисленные переменные – l, r, u, - представляющие индексы левого и правого массивов и массива слияния. Затем в цикле сравниваются элементы левого и правого массива с текущими значениями индексов l и r. Если элемент левого массива меньше или равен элемента правого массива, то он переписывается в массив слияния и происходит сдвиг по левому массиву – увеличивается индекс l. В противном случае аналогичные операции выполняются над элементом правого массива. Цикл завершается, когда один из массивов полностью переписан – соответствующий индекс l или r достигнет предельного значения. После этого в «хвост» массива слияния дописывается остаток массива, чей индекс не достигнул своего предела.

Еще одно небольшое замечание к реализации алгоритма. Обычно рекурсивная процедура имеет дополнительные аргументы, позволяющие разбивать задачу на подзадачи. В нашем случае у процедуры RSortMerge заданы два аргумента, характеризующие начало и конец сортируемой части массива. Чтобы не требовать у клиента задания этих аргументов при вызове процедуры, обычно пишут нерекурсивную процедуру без аргументов, единственное назначение которой состоит в вызове рекурсивной процедуры с исходными значениями аргументов:

```
public void SortMerge()
{ RSortMerge(0, n-1);}
```

Быстрая сортировка Хоара

И в этом алгоритме исходная задача разбивается на две подзадачи, объединение решений которых дает общее решение. Разбиение задачи на две подзадачи одинаковой размерности в этом случае делается следующим образом. Найдем медиану сортируемого массива – x. По определению элемент, стоящий посредине сортируемого массива является его медианой. Поэтому элемент x позволяет разбить исходное множество элементов на два равных подмножества – элементов, меньших или равных x, и элементов, больших или равных x. Если каждое из подмножеств будет отсортировано, то будет отсортирован и весь массив, так что на объединение решений не требуется никаких затрат. Применяя рекурсивно этот алгоритм к каждому из подмножеств, в конечном итоге приходим к множествам размерности 1, отсортированным по определению.

Чтобы такой способ сортировки работал за время O(n*log(n)), необходимо, уметь находить медиану массива за линейное время и разбивать множество на два подмножества за линейное время. Алгоритм решения второй из этих задач за один проход по массиву мы разбирали подробно – его вариация реализуется в процедуре SortTwoKinds. Для нахождения медианы массива существуют алгоритмы, укладывающиеся в линейное время. Сам Хоар предложил эффективный алгоритм поиска медианы, требующий в среднем времени O(n), но в худших случаях требующий времени порядка $O(n^2)$.

В алгоритме, получившем название «быстрой сортировки» Хоара, поиск медианы массивы вообще не проводится. Вместо этого выбирается случайным образом любой элемент массива. Он и используется в качестве барьера, разбивающего исходное множество на два подмножества, которые в этом случае могут иметь разную размерность. В таком варианте сложность алгоритма имеет порядок O(n*log(n)) только в среднем, а в худшем случае будет иметь сложность $O(n^2)$. На практике оказывается, что именно такой алгоритм является наиболее быстрым в большинстве случаев. Поэтому он и используется в качестве встроенного алгоритма сортировки для большинства программных систем. В частности он используется и при работе на C# в методе Sort класса Array из библиотеки FCL.

Подробный разбор возможной реализации этого алгоритма с неформальным доказательством его корректности дан в лекции 10 учебника [1].

Рассмотрим идею алгоритма поиска медианы, точнее, его общий вариант – нахождение квантили массива порядка k. Квантилью порядка k называется элемент для которого в массиве существует k элементов, меньших или равных квантили, и n - k элементов, больших или равных квантили. Медиана – это квантиль порядка n/2.

Идея алгоритма основана на возможности разбиения множества на два подмножества относительно некоторого барьера. Как уже говорилось, эта идея лежит в основе алгоритма быстрой сортировки. Она же используется и в алгоритме поиска квантили. Опишем ее по отношению к поиску медианы. Возьмем элемент с номером k = n/2, используем его в качестве барьера x, разобьем множество элементов на два подмножества. Пусть i и j задают границы этих подмножеств. Так как в каждое из подмножеств могут входить элементы, равные x, то границы задают некоторый интервал. Если окажется, что номер k принадлежит интервалу [i, j], то задача решена и медиана найдена. Если же номер k лежит вне этого интервала, то задача разбиения повторяется на той части интервала, которой принадлежит номер k. Поскольку длина интервала в среднем уменьшается в два раза, то в среднем потребуется линейное время для нахождения квантили.

Приведу реализацию этого алгоритма для случая целочисленных массивов:

```
/// <summary>
        /// Поиск за линейное (в среднем) время квантили
        /// массива порядка k.
        /// При k = n/2 квантиль задает медиану массива.
        /// </summary>
        /// <param name="k">порядок квантили</param>
        /// <returns> квантиль порядка k</returns>
        public int Find(int k)
```

```
{
    int l = 0, r = n - 1;
    int x = 0;
    while (l < r)
    {
        x = arr[k]; //барьер
        //Разбиение на два подмножества
        int i = l, j = r;
        do
        {
            while (arr[i] < x) i++;
            while (x < arr[j]) j--;
            if (i <= j)
            {
                if (arr[i] != arr[j])
                {//обмен
                    int w = arr[i]; arr[i] = arr[j]; arr[j] = w;
                }
                i++; j--;
            }
        }
        while (i <= j);
        //Изменение границ интервала поиска
        if (j < k) l = i;
        if(k < i) r = j;
    }
    return (arr[k]);
}
```

Пирамидальная сортировка

Этот алгоритм носит разные названия – пирамидальной сортировки, сортировки деревом, иногда его называют сортировкой кучи. Алгоритм рекурсивный, имеет сложность порядка O(n*log(n)) и не требует дополнительной памяти.

Для пояснения идеи алгоритма заметим, что любой массив из n элементов с индексами от 0 до n-1можно рассматривать как запись некоторого бинарного дерева, каждый узел (вершина) которого имеет одного или двух потомков. Первый элемент массива с индексом 0 будем рассматривать как корень дерева. Элементы, начиная с индекса n/2, являются листьями дерева, не имеющими потомков. Потомками элемента с индексом i (i < n/2) являются элементы с индексами 2*i + 1, 2*i + 2. В соответствии с этим определением каждый массив можно рассматривать как бинарное дерево высоты log(n).

Массив, задающий бинарное дерево, называется пирамидой (пирамидальным деревом), если для каждой вершины дерева значение элемента, представленного в вершине, меньше либо равно значений элементов, представленных его потомками. Отсюда в частности следует, что минимальный элемент массива находится в корне пирамидального дерева.

Алгоритм пирамидальной сортировки состоит из двух частей, в первой части массив преобразуется в пирамиду. Во второй части по пирамиде строится отсортированный массив. Обе части

алгоритма можно выполнить за время O(n*log(n)). В основе алгоритма каждой из этих частей лежит рекурсивная процедура просеивания Seed(k). Идея алгоритма просеивания Seed(k) состоит в следующем. Пусть построена пирамида, но в одном ее узле k свойство пирамидальности нарушено. Требуется восстановить пирамидальность. Понятно, что если у узла k нет потомков – он является листом дерева, то ничего делать не нужно, поскольку на листья никаких ограничений не накладывается. Если же у листа имеется один или два потомка, но значение в узле меньше значений его потомков, то тоже ничего делать не нужно, - свойство пирамидальности выполняется. В противном случае нужно найти потомка с наименьшим значением и провести обмен значений между узлом и потомком. После обмена нужно рекурсивно вызвать процедуру Seed для нового узла. Поскольку каждый раз происходит спуск по дереву, а высота дерева не превосходит log(n), то время работы этой процедуры пропорционально числу ее вызовов и имеет порядок O(log(n)).

Процедура построения пирамиды выглядит совсем просто. Листья дерева не нарушают пирамидальность. Поэтому достаточно достроить пирамиду для элементов, не являющихся листьями, начиная с конца и двигаясь к вершине пирамиды. Вот ее текст:

```
public void MakePyramid()
        {
            int m = (n-2) / 2;
            //m- индекс последнего элемента, имеющего потомков
            //элементы с индексами, большими m, являются листьями пирамиды
            for (int k = m; k >= 0; k--)
                Seed(k);
        }
```

Понятно, что время работы этой процедуры с учетом времени работы процедуры Seed имеет порядок O(n*log(n)).

Также просто решается задача построения отсортированного массива по пирамиде. Поскольку минимальный элемент массива находится в корне дерева, то он извлекается и меняется местами с последним элементом массива. К элементу, поставленному в вершину, применяется процедура просеивания, предполагая, что размер пирамиды уменьшился на 1. Затем этот процесс повторяется. Понятно, что и в этом случае время работы имеет сложность порядка O(n*log(n)).

Задачи

4.102 Построить процедуру сортировки слиянием SortMerge для массива с конкретным типом элементов (int, string). Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.103 Построить процедуру быстрой сортировки QuickSort для массива с конкретным типом элементов (int, string). Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.104 Построить процедуру нахождения квантили порядка k для массива с конкретным типом его элементов (int, string). Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.105 Используя процедуру нахождения медианы массива, построить вариант процедуры быстрой сортировки QuickSort для массива с конкретным типом элементов (int, string). Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.106 Построить процедуру пирамидальной сортировки PyramidSort для массива с конкретным типом элементов (int, string). Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.107 Построить универсальную процедуру сортировки слиянием SortMerge для массива с элементами произвольного типа T. Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.108 Построить универсальную процедуру быстрой сортировки QuickSort для массива с элементами произвольного типа T. Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.109 Построить универсальную процедуру нахождения квантили порядка k для массива с элементами произвольного типа T. Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.110 Используя универсальную процедуру нахождения медианы массива, построить вариант процедуры быстрой сортировки QuickSort для массива с элементами произвольного типа T. Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

4.111 Построить универсальную процедуру пирамидальной сортировки PyramidSort для массива с элементами произвольного типа T. Привести инварианты циклов, позволяющие доказать корректность алгоритма. Построить интерфейс, позволяющий оценить время работы процедуры.

Проекты

4.112 Постройте класс Sorting, содержащий рекурсивные и нерекурсивные методы сортировки, и класс Analyze, позволяющий анализировать время работы методов сортировки на одних и тех же массивах. Интерфейс проекта должен поддерживать представление результатов анализа в виде графиков.

4.113 Постройте универсальный класс Sorting, содержащий рекурсивные и нерекурсивные методы сортировки массивов произвольного типа, и класс Analyze, позволяющий анализировать время работы методов сортировки на одних и тех же массивах. Интерфейс проекта должен поддерживать представление результатов анализа в виде графиков.

4.114 Постройте проект «Словарь терминов». Проект должен позволять работать со словарем терминов. Элементами словаря являются термины и их определения. Над элементами словаря определены такие операции как поиск, вставка, замена и удаление. В словарь разрешается добавлять новые термины и их определения, искать в словаре по термину его определение (перевод), искать по переводу сам термин. Соответствие при поиске можно задавать как строгое, так и нестрогое. При нестрогом соответствии слово можно искать по его префиксу или суффиксу, предложение можно искать по одному или нескольким словам, входящим в термин или его определение. В качестве основы для интерфейса можно взять известный словарь Abby Lingvo.

Проект «Сортировки»

В заключение приведу текст проекта, содержащего некоторый вариант класса, содержащего реализацию рекурсивных методов сортировки и позволяющего также оценить время сортировки на массивах со случайным заполнением элементов. Покажу также возможный вариант интерфейса для работы с этим классом. Начну с интерфейса. На рис. 4_2 показана форма, позволяющая анализировать время работы трех эффективных методов сортировки.

Рис. 4_2 Интерфейс для работы с рекурсивными методами сортировки

На рис. 4_2 форма показана в процессе работы. Можно видеть, что на данном конкретном примере (массиве размерности 100) быстрая сортировка и пирамидальная сортировка дают примерно одинаковые результаты (516 и 515), а сортировка слиянием дает чуть худшие результаты (594). Эти результаты слегка варьируются от примера к примеру. Так, например, при увеличении размерности массива в 10 раз цифры следующие – 4078 для быстрой сортировки, 5094 – для пирамидальной и 4984

– для сортировки слиянием. Как видите, время увеличилось также в 10 раз даже чуть меньше. Быстрая сортировка оказалась лучше, чем ее соперники, а пирамидальная сортировка на этом примере оказалась чуть хуже сортировки слиянием. Приведу еще цифры для массива размерности 10000 – они соответственно равны: 44156, 65891, 61031. Быстрая сортировка существенно опередила соперников.

Вот как выглядит класс Sorting. Начну с описания полей класса, его конструктора, используемого в классе делегата и метода, заполняющего массив случайными числами:

```
class SortingArrs
{
    //delegate
    public delegate void SortMethod();
    //fields
    int n;  //размер массива
    int Heap_Size;   //размер пирамиды, построенной на массиве
    public int[] arr;  //сортируемый массив
    Random rnd;
    //конструктор класса
    public SortingArrs(int n)
    {
       this.n = n;
       arr = new int[n];
       Heap_Size = n;
       rnd = new Random();
    }
    public void FillRndNumbers()
    {
       for (int i = 0; i < n; i++)
        //arr[i] = rnd.Next(n);
        arr[i] =   rnd.Next(-1000, 1000);
    }
```

Комментарии здесь излишни. А вот как выглядит подробное описание метода быстрой сортировки с инвариантами и доказательством корректности, встроенном в программный текст.

```
/// <summary>
/// Вызывает рекурсивную процедуру QSort,
/// передавая ей границы сортируемого массива.
/// Сортируемый массив arr задается
/// соответствующим полем класса.
/// </summary>
public void QuickSort()
{
    QSort(0, n - 1);
}
/// <summary>
/// Небольшая по размеру процедура содержит три
```

```
/// вложенных цикла while, два оператора if и рекурсивные
/// вызовы. Для таких процедур задание инвариантов и
/// доказательство корректности облегчает отладку.
/// Предусловие: (start <= finish)
/// Постусловие: массив arr отсортирован по возрастанию
/// </summary>
/// <param name="start">начальный индекс сортируемой части
/// массива arr</param>
/// <param name="finish">конечный индекс сортируемой части
/// массива arr</param>

void QSort(int start, int finish)
{
    if (start != finish)
    //если (start = finish), то процедура ничего не делает,
    //но постусловие выполняется, поскольку массив из одного
    //элемента отсортирован по определению.
    //Докажем истинность постусловия для массива с числом элементов >1.
    {
        int ind = rnd.Next(start, finish);
        int item = arr[ind];
        int ind1 = start, ind2 = finish;
        int temp;
    /********
     Введем три непересекающихся множества:
     S1: {arr(i), start <= i =< ind1-1}
     S2: {arr(i), ind1 <= i =< ind2}
     S1: {arr(i), ind2+1 <= i =< finish}
     Введем следующие логические условия,
     играющие роль инвариантов циклов нашей программы:
     P1: объединение S1, S2, S3 = arr
     P2: (S1(i) < item) Для всех элементов S1
     P3: (S3(i) >= item) Для всех элементов S3
     P4: item - случайно выбранный элемент arr
     Нетрудно видеть, что все условия становятся
     истинными после завершения инициализатора цикла.
     Для пустых множеств S1 и S3 условия P2 и P3
     считаются истинными по определению.
     Inv = P1 & P2 & P3 & P4
    ********/
    while (ind1 <= ind2)
```

```
    {
        while ((ind1 <= ind2) && (arr[ind1] < item)) ind1++;
            //(Inv == true) & ~B1 (B1 - условие цикла while)
        while ((ind1 <= ind2) && (arr[ind2] >= item)) ind2--;
            //(Inv == true) & ~B2 (B2 - условие цикла while)
        if (ind1 < ind2)
        //Из Inv & ~B1 & ~B2 & B3 следует истинность:
        //((arr[ind1] >= item)&&(arr[ind2]<item))== true
        //Это условие гарантирует, что последующий обмен элементов
        //обеспечит выполнение инварианта Inv
       {
           temp = arr[ind1]; arr[ind1] = arr[ind2];
            arr[ind2] = temp;
            ind1++; ind2--;
        }
        //(Inv ==true)
    }
    //из условия окончания цикла следует: (S2 - пустое множество)
    if (ind1 == start)
    //В этой точке S1 и S2 - это пустые множества, -> (S3 = arr)
    //Нетрудно доказать, что отсюда следует истинность:(item = min)
    //Как следствие, можно минимальный элемент сделать первым,
    // а к оставшемуся множеству применить рекурсивный вызов.
    {
        temp = arr[start]; arr[start] = item;
        arr[ind] = temp;
        QSort(start + 1, finish);
    }
    else
    //Здесь оба множества S1 и S3 не пусты.
    //К ним применим рекурсивный вызов.
    {
        QSort(start, ind1 - 1);
        QSort(ind2 + 1, finish);
    }
    //Индукция по размеру массива и истинность инварианта
    //доказывает истинность постусловия в общем случае.
  }
}// QuickSort
```

Приведу теперь набор процедур, реализующих пирамидальную сортировку:

```
//HeapSort - SortTree - PyramidSort
```

```
/// <summary>
/// Рекурсивная поцедура просеивания элемента по древу (пирамиде)
/// Основная компонента построения пирамиды для массива
/// и построения отсортированного массива по пирамиде
/// </summary>
/// <param name="i"></param>
public void Seed(int i)
{
//просеивание элемента с индексом i,
//нарушающего свойство пирамидальности массива
    int l = 2 * i+1, r = l + 1;   //индексы левого и правого потомка
    int temp=0;
    if (l > Heap_Size-1) return;  //это лист пирамиды
        int cand = i;
        if (arr[i] < arr[l]) cand = l;
        if((r < Heap_Size)&&(arr[cand] <arr[r]))cand = r;
        if (cand != i) //обмен
        {
            temp = arr[i]; arr[i] = arr[cand]; arr[cand] = temp;
            Seed(cand);     //Просеивание вниз
        }
}
/// <summary>
/// Преобразует массив arr  в пирамиду
/// Массив является пирамидой, если для каждого элемента массива,
/// не являющегося листом пирамиды, его значение меньше или равно
/// значения двух его потомков
/// </summary>
public void MakePyramid()
{
    int m = (n-2) / 2;
    //m- индекс последнего элемента, имеющего потомков
    //элементы с индексами, большими m, являются листьями пирамиды
    for (int k = m; k >= 0; k--)
        Seed(k);
}
/// <summary>
   /// Пирамидальная сортировка
   /// Вначале по массиву строится пирамида
   /// затем по пирамиде строится отсортированнный массив
   /// </summary>
```

```
        public void PyramidSort()
        {
            Heap_Size = arr.Length;
            int temp =0;
            MakePyramid();
            for (int i = 0; i < n - 1; i++)
            {
                temp = arr[0]; arr[0] = arr[Heap_Size-1];
                arr[Heap_Size-1] = temp; Heap_Size--;
                Seed(0);
            }
        }
```

Вот текст еще одного рекурсивного метода сортировки:

```
/// <summary>
/// Сортировка слиянием
/// Вызывает рекурсивную процедуру SortM,
/// передавая ей границы сортируемого массива.
/// Сортируемый массив arr задается
/// соответствующим полем класса.
public void SortMerge()
{
    int[] temp = new int[arr.Length];
    SortM(0, n - 1, temp);
}
/// <summary>
/// Рекурсивная процедура сортировки слиянием
/// </summary>
/// <param name="start">начало сортируемой части массива</param>
/// <param name="finish">конец сортируемой части массива</param>
/// <param name="temp">массив для хранения результатов слияния</param>
void SortM(int start, int finish, int[] temp)
{
    if (start < finish)
    {
        int mid = (start + finish) / 2;
        SortM(start, mid, temp);
        SortM(mid + 1, finish, temp);
        Merge(start, mid, finish, temp);
    }
}
/// <summary>
```

```
/// Слияние отсортированных частей масссива в массив temp
/// После слияния массив переписывается в исходный массив
/// </summary>
/// <param name="start">начало первой части</param>
/// <param name="mid">конец первой части - начало второй</param>
/// <param name="finish">конец второй части</param>
/// <param name="temp">временный массив с результатами слияния</param>
///
void Merge(int start, int mid, int finish, int[] temp)
{
    int topl = start, topr = mid + 1, topt =start;
    //Слияние, пока не завершится одна из частей массива
    while ((topl <= mid) && (topr <= finish))
    {
        if (arr[topl] <= arr[topr])
            temp[topt++] = arr[topl++];
        else
            temp[topt++] = arr[topr++];
    }
    //дописывание остатка незавершенной части массива
    while (topl <= mid)
        temp[topt++] = arr[topl++];
    while (topr <= finish)
        temp[topt++] = arr[topr++];
    //переливаем элементы в исходный массив
    for (int i = start; i <= finish; i++)
        arr[i] = temp[i];
}
```

Добавлю к этим методам сортировки метод, позволяющий вычислять квантили массива:

```
/// <summary>
/// Поиск за линейное время(в среднем)  квантили
/// массива порядка k.
/// При k = n/2 квантиль задает медиану массива.
/// </summary>
/// <param name="k">порядок квантили</param>
/// <returns> квантиль порядка k</returns>
public int Find(int k)
{
    int l = 0, r = n - 1;
    int x = 0;
    while (l < r)
```

```
        {
            x = arr[k]; //барьер
            //Разбиение на два подмножества
            int i = l, j = r;
            do
            {
                while (arr[i] < x) i++;
                while (x < arr[j]) j--;
                if (i <= j)
                {
                    if (arr[i] != arr[j])
                        {//обмен
                            int w = arr[i]; arr[i] = arr[j]; arr[j] = w;
                        }
                    i++; j--;
                }
            }
            while (i <= j);
            //Изменение границ интервала поиска
            if (j < k) l = i;
            if(k < i) r = j;
        }
        return (arr[k]);
    }
```

Закончим рассмотрение методов класса специальным методом, позволяющим вычислить время, затрачиваемое на сортировку массива:

```
    /// <summary>
            /// Подсчет времени сортировки
            /// Сортируется один и тот же массив 10000 раз
            /// Метод сортировки передается как параметр
            /// </summary>
            /// <param name="sort">Метод сортировки</param>
            /// <returns> время сортировки </returns>
            public int HowLong(SortMethod sort)
            {
                const int count = 10000;
                int start, finish;
                start = MS(DateTime.Now);
                for (int i = 1; i <= count; i++)
                {
                    Random rnd = new Random(77);
```

```
            for (int j = 0; j < n; j++)
            {
                arr[j] = rnd.Next(-1000,1000);
            }
            sort();
        }
        finish = MS(DateTime.Now);
        return (finish - start);
}//HowLong
int MS(DateTime dt)
{
   return ((dt.Hour * 60 + dt.Minute) * 60 + dt.Second) * 1000
       + dt.Millisecond;
}
```

Литература

1. В. Биллиг «Основы программирования на C#», М. 2006 г.
2. Б. Мейер «Объектно-ориентированное конструирование программных систем», М., 2005 г.
3. Д. Кнут « Искусство программирования», т. 3 «Сортировки»
4. Р. Грэхем, Д. Кнут, О. Паташник «Конкретная математика», М., 1998 г.
5. Т. Кормен, Ч. Лейзерсон, Р. Ривест «Алгоритмы. Построение и анализ», М., 2001 г.
6. Д. Гасфилд «Строки, деревья и последовательности в алгоритмах», С-Пб., 2003 г.
7. Н. Вирт «Алгоритмы + структуры данных = программы», М. 1985 г.
8. Д. Баррон « Введение в языки программирования», М. 1980г.
9. Н. Трифонов «Сборник упражнений по языку Алгол» М. 1975 г.