Министерство образования и науки Российской Федерации

Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М. Э. Абрамян

Структуры данных в PascalABC.NET

Выпуск 2

Минимумы и максимумы.
Списки, множества, словари, стеки и очереди.
Многомерные структуры

Учебное пособие по курсу
«Основы программирования»
для студентов естественнонаучных
и технических специальностей

Ростов-на-Дону 2016

УДК 004.438.NET
ББК 32.973.202
А 13

Печатается по решению Редакционно-издательского совета
Института математики, механики и компьютерных наук им. И. И. Воровича
Южного федерального университета (протокол № 4 от 1 сентября 2016 г.)

Рецензенты:

доктор технических наук, профессор кафедры «Информатика»
Ростовского государственного университета путей сообщения (РГУПС)
М. А. Бутакова

кандидат физико-математических наук, доцент кафедры алгебры
и дискретной математики Института математики, механики
и компьютерных наук им. И. И. Воровича Южного федерального университета
С. С. Михалкович

Абрамян М. Э.

А 13 Структуры данных в PascalABC.NET. Выпуск 2. Минимумы и максимумы. Списки, множества, словари, стеки и очереди. Многомерные структуры. — Ростов н/Д : Изд-во ЮФУ, 2016. — 118 с.

ISBN

Учебное пособие является вторым в серии пособий, посвященных структурам данных в языке PascalABC.NET (версия 3.1). В нем обсуждаются различные варианты алгоритмов, связанных с нахождением минимальных и максимальных элементов, рассматриваются структуры данных из библиотеки PascalABC.NET (списки, множества, словари, стеки и очереди), а также детально описываются особенности работы с многомерными структурами, в том числе многомерными и невыровненными массивами. Изложение сопровождается многочисленными примерами, причем основная часть примеров представляет собой решения задач из электронного задачника Programming Taskbook, встроенного в систему PascalABC.NET. Пособие снабжено подробным указателем.

Для преподавателей программирования, старшеклассников и студентов.

УДК 004.438.NET
ББК 32.973.202

ISBN

Предисловие

Предлагаемое учебное пособие является вторым в серии пособий, посвященных структурам данных в языке PascalABC.NET. В дополнение к одномерным динамическим массивам и последовательностям, описанным в первом выпуске [1], в данном пособии рассматриваются структуры данных из стандартной библиотеки платформы .NET, а также различные варианты многомерных структур, в том числе многомерные и невыровненные массивы.

Пособие состоит из 8 глав. Глава 1 является вводной; в ней на материале, изложенном в [1], обсуждаются различные варианты алгоритмов, связанных с нахождением минимальных и максимальных элементов. Главы 2–5 посвящены различным типам структур данных, реализованных в стандартной библиотеке .NET в виде обобщенных классов. В главе 2 описываются два варианта списков: список на базе массива List<T> и двусвязный список на базе цепочки узлов LinkedList<T>. В главе 3 рассматриваются типы множеств, доступные в PascalABC.NET: традиционный тип set of T и два класса из стандартной библиотеки .NET: HashSet<T> (множество на базе хеш-таблицы) и SortedSet<T> (отсортированное множество). В главе 4 описываются словари (ассоциативные массивы), реализованные в библиотеке .NET в виде обобщенного класса Dictionary<TKey, TVal>. В главе 5 рассматриваются стеки и очереди — классы Stack<T> и Queue<T>. Главы 6–7 посвящены многомерным структурам. В главе 6 подробно описываются многомерные массивы, а в главе 7 обсуждаются приемы работы с многомерными иерархическими структурами, конструируемыми на основе уже изученных одномерных структур, при этом основное внимание уделяется массивам массивов (невыровненным массивам) и спискам списков. В главе 8 описываются статические методы класса Array — базового класса платформы .NET, на основе которого реализуются массивы в языках данной платформы. Эти методы могут рассматриваться как дополнения к основным методам массивов, описанным в [1].

Каждая глава завершается пунктом, в котором изученные структуры используются для решения задач (глава 1 целиком состоит из таких пунктов). Как и в [1], задачи берутся из электронного задачника Programming Taskbook, интегрированного в систему Programming Taskbook. В главе 1

рассматриваются задачи из группы Minmax, в главах 2–5 и 8 — из группы Array, в главах 6–7 — из группы Matrix. Для большинства задач приводится несколько вариантов их решения, использующих различные средства PascalABC.NET, и обсуждаются их особенности; в частности, исследуется быстродействие реализованных алгоритмов. В целях закрепления изученного материала читателю рекомендуется рассмотреть аналогичные задачи из групп Minmax, Array и Matrix и решить их различными способами. При этом ему могут оказаться полезными указания к большинству задач из этих групп, приведенные в [2].

Приводимые в книге результаты численных экспериментов были получены на компьютере со следующей конфигурацией (в скобках указывается индекс производительности, используемый Windows для оценки компонентов системы):

- процессор AMD A10-6700 3,70 GHz (7,3),
- оперативная память 6 Г (7,3),
- 64-разрядная операционная система Windows 7.

Предполагается, что читатель предварительно изучил первый выпуск данной серии [1], на который в тексте пособия даются многочисленные ссылки. Как и в [1], от читателя не требуется знакомства с объектно-ориентированным программированием, хотя в тексте упоминаются такие понятия ООП, как «класс» и «обобщенный класс», «объект» («экземпляр класса»), «метод» и «статический метод», «свойство». Смысл используемых понятий легко восстанавливается из контекста, в котором они упоминаются. Для более серьезного изучения объектной модели платформы .NET, реализованной в PascalABC.NET, можно рекомендовать пособие [3].

Как и первый выпуск данной серии, пособие снабжено указателем, содержащим ссылки на свойства и методы всех изученных структур и ссылки на все разобранные в книге задачи и варианты их решения.

В книге описываются возможности системы PascalABC.NET по состоянию на июль 2016 г. (версия 3.1). Скачать последнюю версию системы можно на ее сайте **http://pascalabc.net**.

Глава 1. Минимумы и максимумы

Алгоритмы нахождения минимальных и максимальных значений являются одними из наиболее часто используемых, поэтому в современных языках программирования обычно предусматриваются их стандартные реализации. В PascalABC.NET средства для поиска минимумов и максимумов и связанных с ними характеристик реализованы в виде запросов последовательностей (см. [1, гл. 4]), а также методов массивов (см. [1, гл. 5]). В то же время, возможны ситуации, в которых предпочтительнее использовать модификации стандартных алгоритмов поиска минимумов и максимумов, не прибегая к стандартным запросам или методам. В настоящей главе рассматриваются типовые задачи, связанные с минимумами и максимумами, приводятся различные варианты их решений и обсуждается их эффективность. При решении задач активно используются запросы последовательностей и методы массивов, поэтому материал данной главы можно рассматривать как иллюстрацию средств языка PascalABC.NET, рассмотренных в первом выпуске данной серии [1].

Задачи взяты из группы Minmax электронного задачника Programming Taskbook (работа с задачником подробно описывается в [1, гл. 2]).

1.1. Нахождение минимальных и максимальных элементов и их индексов

Начнем с задачи **Minmax1**, в которой требуется найти минимальный и максимальный элемент исходного вещественного массива. При непосредственной реализации алгоритма получаем следующий вариант решения:

```
Task('Minmax1'); // Вариант 1
var a := ReadArrReal;
var min := a[0];
var max := a[0];
for var i := 1 to a.Length - 1 do
  if a[i] < min then
    min := a[i]
  else if a[i] > max then
    max := a[i];
```

```
Write(min, max);
```

Для ввода исходного массива мы используем функцию ReadAddReal из модуля PT4, которая вначале вводит размер массива, затем выделяет память требуемого размера, считывает значения элементов и возвращает полученный массив.

Переменные min и max инициализируются начальным элементом массива, а затем в цикле сравниваются с очередными его элементами. При нахождении элемента с меньшим значением изменяется min, при нахождении элемента с бóльшим значением изменяется max. Заметим, что вместо цикла for можно было использовать цикл foreach [1, п. 3.2]; в этом случае не потребуется обращаться к свойству Length и использовать индексы:

```
foreach var e in a do
  if e < min then
    min := e
  else if e > max then
    max := e;
```

Единственный недостаток использования цикла foreach в данном случае — то, что он обрабатывает и *первый* элемент массива (впрочем, при обработке первого элемента не будет выполнено никаких действий, кроме двух сравнений).

Если воспользоваться запросами Min и Max, доступными для любой коллекции, в том числе и для массивов (см. [1, п. 4.1]), то решение будет состоять всего лишь из двух операторов (не считая оператора вызова процедуры Task, инициализирующей задачу):

```
Task('Minmax1'); // Вариант 2
var a := ReadArrReal;
Write(a.Min, a.Max);
```

Поскольку в данном случае минимальный и максимальный элементы вычисляются независимо друг от друга (и, таким образом, для их вычисления требуется двукратный просмотр массива), можно ожидать, что быстродействие данного варианта будет ниже, чем варианта 1 с явной реализацией алгоритма, использующей единственный цикл. Разумеется, эффект от подобного замедления будет проявляться лишь при обработке массивов *очень* большого размера; тем не менее, информация о сравнительном быстродействии алгоритмов может оказаться полезной при решении вопроса об их использовании в конкретной программе. В дальнейшем мы будем неоднократно проводить численные эксперименты для сравнительного анализа быстродействия алгоритмов, поэтому на примере алгоритма одновременного поиска минимального и максимального элемента опишем такой численный эксперимент подробно.

Понятно, что оценить сравнительное быстродействие программ, использующих задачник Programming Taskbook, нельзя, так как в них обрабатываются массивы небольшого размера. Применение задачника удобно на этапе разработки и тестирования алгоритма, поскольку упрощает ввод и вывод данных и обеспечивает автоматическую проверку правильности.

Для проверки быстродействия мы должны самостоятельно генерировать исходные массивы. В этом нам помогут соответствующие функции-генераторы, описанные в [1, п. 3.3]. Для хранения текущего размера массива будем использовать переменную size. Тогда для создания массива нам будет достаточно вызвать функцию ArrRandomReal(size). При этом элементы массива будут выбираться случайным образом из полуинтервала [0, 10), что нас вполне устраивает.

Чтобы определять время работы части программы, будем использовать функцию MillisecondsDelta [1, п. 5.2]. Напомним, что она возвращает число миллисекунд, прошедших с момента ее предыдущего вызова (или с начала выполнения программы, если функция вызвана впервые).

Будем проверять алгоритмы для массивов, содержащих от 4 до 32 миллионов элементов, выполняя последовательное удвоение их размера. Программа для проведения численного эксперимента принимает следующий вид (для краткости мы указываем только раздел операторов без обрамляющих его слов begin-end):

```
var size := 2000000;
for var j := 1 to 4 do
begin
  size := 2 * size;
  var a := ArrRandomReal(size);
  MillisecondsDelta;
  var min := a[0];
  var max := a[0];
  for var i := 1 to size - 1 do
    if a[i] < min then
      min := a[i]
    else if a[i] > max then
      max := a[i];
  var t1 := MillisecondsDelta;
  var min2 := a.Min;
  var max2 := a.Max;
  var t2 := MillisecondsDelta;
  Writeln(size:8, t1:6, t2:6, min:14:10, min2:14:10,
    max:14:10, max2:14:10);
end;
```

Полужирным шрифтом выделены фрагменты, содержащие реализацию первого и второго варианта алгоритма. Для каждого размера size выводится сам размер, время работы первого и второго варианта алгоритма и найденные ими значения минимального и максимального алгоритма.

Для проверки быстродействия программу следует запускать в *режиме релиза* (см. замечание в [1, п. 3.4]). При запуске в этом режиме может быть получен такой результат:

```
 4000000   11   75 0.0000024308 0.0000024308 9.9999992875 9.9999992875
 8000000   21  144 0.0000005821 0.0000005821 9.9999996880 9.9999996880
16000000   42  293 0.0000011595 0.0000011595 9.9999996461 9.9999996461
32000000   84  575 0.0000002468 0.0000002468 9.9999999488 9.9999999488
```

Итак, вариант с использованием запросов работает примерно в *семь* раз медленнее. Интересно отметить, что при запуске программы в *режиме отладки* разница в результатах будет существенно меньше:

```
 4000000   34   78 0.0000011129 0.0000011129 9.9999919022 9.9999919022
 8000000   67  146 0.0000014063 0.0000014063 9.9999987381 9.9999987381
16000000  131  294 0.0000013877 0.0000013877 9.9999999488 9.9999999488
32000000  265  580 0.0000001350 0.0000001350 9.9999996973 9.9999996973
```

Это связано с тем, что действия по генерации отладочной информации приводят к большему замедлению программ, содержащих явные циклы.

Теперь обратимся к задаче **Minmax4**, посвященной столь же часто возникающей задаче — нахождению *индекса* минимального или максимального элемента. В задаче Minmax4 требуется найти *порядковый номер* минимального элемента вещественного массива. Напомним, что порядковые номера начинаются с 1, тогда как индексы динамических массивов начинаются с 0, и это обстоятельство надо учитывать при решении задачи. Заметим также, что в задачах электронного задачника Programming Taskbook, связанных с обработкой *вещественного* массива, предполагается, что все его элементы являются *различными*, поэтому в нашем случае вещественный массив будет иметь ровно *один* минимальный элемент, и следовательно, его номер будет находиться однозначно.

Вначале приведем реализацию алгоритма, использующую цикл. Отличие от алгоритма нахождения минимума состоит в том, что при изменении переменной min требуется дополнительно изменять связанную с ней переменную k, содержащую *индекс* элемента, записываемого в min:

```
Task('Minmax4'); // Вариант 1
var a := ReadArrReal;
var min := a[0];
var k := 0;
for var i := 1 to a.Length - 1 do
  if a[i] < min then
  begin
```

```
    min := a[i];
    k := i;
  end;
Write(k + 1);
```

При выводе мы прибавляем к переменной k единицу, чтобы перейти от индексов к порядковым номерам.

Среди запросов для последовательностей отсутствует запрос, позволяющий найти индекс минимального элемента, однако подобный метод предусмотрен у массивов: это метод IndexMin, который находит индекс *первого* минимального элемента [1, п. 5.3]. Имеется метод LastIndexMin, находящий индекс последнего минимального элемента, а также аналогичные методы для поиска индексов максимальных элементов. Вариант решения, использующий метод IndexMin, будет очень коротким:

```
Task('Minmax4'); // Вариант 2
var a := ReadArrReal;
var k := a.IndexMin;
Write(k + 1);
```

Этот вариант можно представить в виде *единственного* оператора, причем в программе не потребуется использовать ни одной переменной:

```
Task('Minmax4'); // Вариант 2a
Write(ReadArrReal.IndexMin + 1);
```

Численный эксперимент показывает, что вариант с явной реализацией алгоритма работает быстрее примерно в 2 раза (см. таблицу 1.1). Исходные массивы генерировались так же, как и в программе проверки быстродействия алгоритмов для задачи Minmax1.

Таблица 1.1. *Быстродействие вариантов решения задачи Minmax4*

Размер size	Вариант 1	Вариант 2
4000000	8	18
8000000	16	35
16000000	34	66
32000000	64	132

В задаче **Minmax8** требуется обработать целочисленный массив. Для него надо найти порядковые номера первого и последнего минимального элемента (эти номера могут различаться, так как в целочисленных массивах вполне могут содержаться одинаковые элементы). В варианте, использующем цикл, достаточно выполнить, как обычно, один проход по массиву, заполняя *три* переменные: min, k1 (индекс первого минимального элемента) и k2 (индекс последнего минимального элемента):

```
Task('Minmax8'); // Вариант 1
var a := ReadArrInteger;
```

```
var min := a[0];
var k1 := 0;
var k2 := 0;
for var i := 1 to a.Length - 1 do
 if a[i] < min then
 begin
  min := a[i];
  k1 := i;
  k2 := i;
 end
 else if a[i] = min then
  k2 := i;
Write(k1 + 1, k2 + 1);
```

Для ввода целочисленного массива используется функция ReadArrInteger из модуля PT4. В цикле обрабатываются две ситуации: когда очередной элемент оказывается меньшим, чем min (в этом случае требуется изменить все переменные), и когда очередной элемент совпадает с min (в этом случае надо изменить только переменную k2).

Вариант, применяющий *методы* динамических массивов, состоит всего из двух операторов:

```
Task('Minmax8'); // Вариант 2
var a := ReadArrInteger;
Write(a.IndexMin + 1, a.LastIndexMin + 1);
```

При проверке быстродействия массивы генерировались следующим образом: ArrRandomInteger(size, –size div 10, size div 10); это, с одной стороны, обеспечивало разнообразие значений, а с другой — гарантировало, что многие значения будут повторяться. Проверка показала (см. таблицу 1.2), что быстродействие этих вариантов различается в два раза.

Таблица 1.2. *Быстродействие вариантов решения задачи Minmax8*

Размер size	Вариант 1	Вариант 2
4000000	12	23
8000000	23	43
16000000	45	82
32000000	90	176

1.2. Нахождение условных минимумов и максимумов

В программах часто требуется находить минимумы или максимумы только среди тех элементов, которые удовлетворяют определенным условиям. В этом случае говорят о нахождении *условных минимумов/максимумов*. Примером задачи на нахождение условного минимума является задача **Minmax12**, в которой требуется найти минимальный *положительный*

элемент вещественного массива (если положительные элементы в массиве отсутствуют, то надо вывести особое значение 0). Ее решение представляет собой стандартный алгоритм нахождения условного минимума:

```
Task('Minmax12'); // Вариант 1
var a := ReadArrReal;
var min := real.MaxValue;
for var i := 0 to a.Length - 1 do
  if (a[i] > 0) and (a[i] < min) then
    min := a[i];
if min = real.MaxValue then
  min := 0;
Write(min);
```

Обратите внимание на особенности этого алгоритма по сравнению с алгоритмом нахождения «обычного» минимума. Во-первых, теперь нельзя инициализировать переменную min начальным элементом массива, так как это элемент может не удовлетворять дополнительному условию. В подобной ситуации проще всего инициализировать min каким-либо *большим* значением. В программе переменной min присваивается наибольшее значение вещественного типа; это значение можно получить с помощью свойства MaxValue, имеющегося у *любого* числового типа (аналогичным образом, переменная для хранения максимума может инициализироваться каким-либо *маленьким* значением, например real.MinValue). Во-вторых, в цикле выполняется проверка того, что элемент a[i] удовлетворяет дополнительному условию. И, наконец, в-третьих, после цикла проверяется, содержал ли исходный массив элементы, удовлетворяющие требуемому условию. Если такие элементы не были обнаружены, то переменная min сохранит свое начальное значение; в этом случае по условию задачи в нее надо записать число 0.

Вместо цикла for в варианте 1 можно использовать цикл foreach.

Специальных запросов или методов динамических массивов, предназначенных для нахождения условных минимумов/максимумов, не предусмотрено, однако можно воспользоваться обычным вариантом запроса Min или Max, если предварительно выполнить *фильтрацию* исходных данных с помощью запроса Where [1, п. 4.2]:

```
var min := a.Where(e -> e > 0).Min;
```

Такой вариант будет правильно работать только в случае, если исходный массив a содержит хотя бы один положительный элемент. Если положительных элементов нет, то запрос Where вернет *пустую последовательность*, к которой нельзя применять запрос Min, поскольку его вызов приведет к аварийному завершению программы. Простейшим способом обработки подобной особой ситуации является применение запроса DefaultIf-

Empty [1, п. 4.4], преобразующего пустую последовательность в одноэлементную:

```
var min := a.Where(e -> e > 0).DefaultIfEmpty.Min;
```

Если запрос DefaultIfEmpty вызывается без параметров, то созданная одноэлементная последовательность будет содержать элемент с нулевым значением — именно это и требуется в нашем случае. Таким образом, при отсутствии положительных элементов запрос Where вернет пустую последовательность, эта последовательность будет преобразована в последовательность из единственного нулевого элемента, который и будет возвращен запросом Min в качестве минимального.

Итак, решение задачи Minmax12, использующее запросы, будет иметь следующий вид:

```
Task('Minmax12'); // Вариант 2
var a := ReadArrReal;
var min := a.Where(e -> e > 0).DefaultIfEmpty.Min;
Write(min);
```

Это решение можно записать в виде единственного оператора:

```
Task('Minmax12'); // Вариант 2a
Write(ReadArrReal.Where(e -> e > 0).DefaultIfEmpty.Min);
```

Замечание. Возможен вариант решения этой задачи, основанный на запросах и при этом не требующий применения запросов Where и DefaultIfEmpty: достаточно использовать запрос Min с параметром — лямбда-выражением и затем дополнительно проанализировать результат запроса. Этот вариант описан в [1, п. 4.1] при решении задачи LinqBegin9, в которой положительный минимум надо найти для целочисленного массива.

Решение задачи Minmax12, использующее запросы, уже не является таким же простым, как аналогичные решения предыдущих задач. Численный эксперимент показывает, что работает этот вариант решения более чем в 2 раза медленнее, чем вариант 1 (см. таблицу 1.3).

Таблица 1.3. *Быстродействие вариантов решения задачи Minmax12*

Размер size	Вариант 1	Вариант 2
4000000	28	70
8000000	54	124
16000000	109	249
32000000	219	497

Теперь обратимся к задаче **Minmax13**, в которой требуется найти *порядковый номер* первого максимального нечетного числа в исходном целочисленном массиве (если массив не содержит нечетных чисел, то надо вывести 0). Для решения этой задачи достаточно модифицировать алгоритм поиска условного максимума (аналогичный алгоритму поиска условного

минимума, описанному в варианте 1 решения предыдущей задачи), добавив в него переменную k для хранения индекса:

```
Task('Minmax13'); // Вариант 1
var a := ReadArrInteger;
var max := integer.MinValue;
var k := -1;
for var i := 0 to a.Length - 1 do
  if (a[i] > max) and Odd(a[i]) then
  begin
    max := a[i];
    k := i;
  end;
Write(k + 1);
```

Обратите внимание на то, что мы инициализировали переменную k особым значением –1, чтобы в случае, если нечетные элементы не будут найдены, мы получили при выводе число 0 (единица прибавляется, как обычно, для того чтобы перейти от *индексов* к *порядковым номерам*).

Решить эту задачу с помощью запросов еще сложнее, чем предыдущую, хотя нам известен метод IndexMax, позволяющий найти индекс первого максимального элемента в массиве. Дело в том, что при фильтрации исходного набора элементов мы потеряем информацию об их исходных индексах. Следует так выполнить фильтрацию, чтобы *сохранить значения индексов вместе с отфильтрованными элементами*. Сделать это можно, если предварительно преобразовать исходный набор чисел в набор *кортежей* вида (i, e), где поле i содержит индекс элемента e (по поводу кортежей в языке PascalABC.NET см. [1, п. 1.4]). Для подобного преобразования в PascalABC.NET предусмотрен особый запрос Numerate [1, п. 4.3]. Затем надо применить к полученной последовательности запрос Where, выбрав только те кортежи, у которых второе поле является нечетным числом (напомним, что для доступа ко второму полю кортежа e можно использовать индексную нотацию: e[1]). Наконец, в отфильтрованной последовательности надо найти элемент с наименьшим вторым полем и вывести *первое* поле найденного элемента. Поскольку в данном случае требуется находить максимум не среди самих элементов, а среди набора значений, вычисляемых по этим элементам (такие значения называют *ключами*), необходимо вместо обычного запроса Max использовать специальный запрос MaxBy [1, п. 4.1]. И, разумеется, необходимо предусмотреть обработку особой ситуации, когда в исходном массиве отсутствуют нечетные числа. В результате получаем следующий вариант решения:

```
Task('Minmax13'); // Вариант 2
var a := ReadArrInteger;
```

```
var k := a.Numerate(0).Where(e -> Odd(e[1]))
  .DefaultIfEmpty((-1,0)).MaxBy(e -> e[1])[0];
Write(k + 1);
```

При вызове запроса Numerate мы явно указали параметр 0, чтобы нумерация начиналась от 0 (по умолчанию нумерация в запросе Numerate начинается от 1).

Особая ситуация обрабатывается, как обычно, с помощью запроса DefaultIfEmpty. В данном случае нас не устраивает его вариант без параметров, так как этот вариант вернет последовательность с элементом nil (поскольку кортежи являются ссылочным типом данных, для которого нулевой элемент означает *нулевую ссылку*). Поэтому мы указываем в запросе DefaultIfEmpty параметр, определяющий кортеж, который будет включен в созданную одноэлементную последовательность. Главное в этом кортеже то, что его первое поле равно –1.

К результату, возвращенному запросом MaxBy, мы сразу применяем операцию индексирования [0]. Благодаря этой операции в переменную k будет записан не весь кортеж с максимальным вторым полем, а только его первое поле — индекс первого максимального нечетного числа из исходного массива.

Полученное решение можно представить в виде единственного оператора:

```
Task('Minmax13'); // Вариант 2a
Write(ReadArrInteger.Numerate(0).Where(e -> Odd(e[1]))
  .DefaultIfEmpty((-1,0)).MaxBy(e -> e[1])[0] + 1);
```

Это решение можно сделать еще немного короче, если «разрешить» запросу Numerate выполнять нумерацию от 1. В этом случае нам не потребуется прибавлять 1 к номеру, возвращаемому цепочкой запросов. Необходимо также изменить параметр запроса DefaultIfEmpty на (0, 0):

```
Task('Minmax13'); // Вариант 2b
Write(ReadArrInteger.Numerate.Where(e -> Odd(e[1]))
  .DefaultIfEmpty((0,0)).MaxBy(e -> e[1])[0]);
```

Использование в алгоритме решения нескольких запросов, в том числе запроса, генерирующего последовательность кортежей, естественно, приводит к увеличению времени работы. Тестирование алгоритмов 1 и 2 на массивах большого размера дало результаты, приведенные в таблице 1.4.

Таблица 1.4. *Быстродействие вариантов решения задачи Minmax13*

Размер size	Вариант 1	Вариант 2
4000000	8	281
8000000	17	573
16000000	33	1068

32000000	66	2134

Таким образом, вариант с запросами выполняется медленнее примерно в 30 раз.

1.3. Дополнительные задачи на минимумы и максимумы

Вторая половина группы Minmax содержит задачи, требующие разработки более сложных алгоритмов, чем те, которые связаны с нахождением «обычных» минимумов и максимумов (в том числе условных) или их номеров.

В задаче **Minmax22** требуется найти *два* наименьших элемента вещественного массива (и вывести их в порядке возрастания их значений). Напомним, что при обработке *вещественных* массивов мы предполагаем, что все их элементы имеют *различные* значения.

Для решения задачи достаточно одного прохода исходного массива; требуется лишь использовать две вспомогательные переменные: min, содержащую минимальный из просмотренных элементов, и min2, содержащую элемент, следующий по величине за минимальным. При обработке очередного элемента a[i] в цикле следует проверять, в какой из промежутков $(-\infty, \text{min})$, (min, min2), (min2, $+\infty$) он попадет, и в зависимости от этого изменять значения переменных min и min2. Например, если a[i] < min, то следует записать в min2 прежнее значение min, а в min — значение a[i], а если a[i] окажется в промежутке (min, min2), то достаточно записать в min2 значение a[i].

Важно, чтобы в начале алгоритма для переменных min и min2 выполнялась оценка min ≤ min2; для этого можно, например, в min записать значение real.MinValue, а в min2 — начальный элемент массива.

Полный вариант алгоритма выглядит следующим образом:

```
Task('Minmax22'); // Вариант 1
var a := ReadArrReal;
var min2 := real.MaxValue;
var min := a[0];
for var i := 1 to a.Length -1 do
  if a[i] < min then
  begin
    min2 := min;
    min := a[i];
  end
  else if a[i] < min2 then
    min2 := a[i];
Write(min, min2);
```

Если не задаваться целью решить задачу за один проход массива, то простое двухпроходное решение можно реализовать с помощью запроса Min и метода массива IndexMin:

```
Task('Minmax22'); // Вариант 2
var a := ReadArrReal;
var k := a.IndexMin;
var min := a[k];
a[k] := real.MaxValue;
var min2 := a.Min;
Write(min, min2);
```

Вначале мы определяем индекс минимального элемента. Это, во-первых, позволяет нам определить значение min, а во-вторых, упрощает поиск min2: достаточно «испортить» наименьший элемент (присвоив ему, например, значение real.MaxValue), чтобы с помощью запроса Min найти следующий по величине элемент. Если желательно сохранить содержимое исходного массива, то достаточно после нахождения min2 восстановить испорченный элемент, присвоив ему значение min: a[k] := min.

Еще один вариант двухпроходного решения можно получить, если использовать запрос Where:

```
Task('Minmax22'); // Вариант 3
var a := ReadArrReal;
var min := a.Min;
var min2 := a.Where(x -> x <> min).Min;
Write(min, min2);
```

В этом варианте, в отличие от предыдущего, мы не изменяем исходный массив. Кроме того, его можно использовать и в случае, если массив может содержать одинаковые элементы.

Наконец, ответ можно получить, *отсортировав* исходный массив по возрастанию (используя, например, процедуру Sort, описанную в [1, п. 5.4]). Тогда требуемые значения будут содержаться в первых двух элементах отсортированного массива:

```
Task('Minmax22'); // Вариант 4
var a := ReadArrReal;
Sort(a);
var min := a[0];
var min2 := a[1];
Write(min, min2);
```

Разумеется, это решение будет наименее эффективным, несмотря на то что в стандартной библиотеке реализованы очень быстрые алгоритмы сортировки.

Результаты проверки быстродействия четырех рассмотренных вариантов (при запуске программы в режиме релиза) приведены в таблице 1.5.

Таблица 1.5. *Быстродействие вариантов решения задачи Minmax22*

Размер size	Вариант 1	Вариант 2	Вариант 3	Вариант 4
4000000	11	55	94	457
8000000	22	131	203	966
16000000	42	251	403	2039
32000000	85	507	808	4162

В заключение рассмотрим задачу **Minmax27**, основная сложность которой состоит в определении тех значений, среди которых надо выбрать максимальный. В этой задаче дается целочисленный массив, содержащий только нули и единицы, и требуется найти номер элемента, с которого начинается самая длинная последовательность одинаковых чисел, а также количество элементов этой последовательности. Если таких последовательностей несколько, то надо вывести номер начального элемента *первой* из них. При обсуждении этой задачи будем для краткости называть последовательности подряд идущих одинаковых элементов *сериями*.

Например, для исходного набора

0 1 1 **0 0 0** 1 0 0 0

правильным ответом будут числа 4 и 3 (4 — *порядковый номер* начального элемента первой из серий одинаковых чисел наибольшей длины, 3 — *количество* элементов в этой серии).

Для решения задачи надо в цикле по элементам исходного массива находить *длины всех серий* и использовать их в алгоритме нахождения максимума (кроме того, надо хранить информацию о *начальной позиции каждой серии*). Чтобы распознавать конец серии, достаточно сравнивать *соседние элементы массива*: если они различны, то, следовательно, в данной позиции закончилась одна серия и началась другая. Получаем следующую программу:

```
Task('Minmax27'); // Вариант 1
var a := ReadArrInteger;
var kmax := 1;
var pmax := 0;
var k := 1;
var p := 0;
for var i := 1 to a.Length - 1 do
  if a[i - 1] = a[i] then
    k += 1
  else
  begin
    if k > kmax then
```

```
    begin
      kmax := k;
      pmax := p;
    end;
    k := 1;
    p := i;
  end;
if k > kmax then
begin
  kmax := k;
  pmax := p;
end;
Write(pmax + 1, kmax);
```

В этой программе переменная p содержит индекс начального элемента текущей серии, а переменная k — ее длину. Переменная pmax содержит индекс начального элемента самой длинной из просмотренных серий, а kmax — длину этой серии. В цикле обрабатываются все элементы, начиная со второго. Первый элемент не обрабатывается, он автоматически учитывается как начальный элемент первой серии (поэтому перед циклом переменные k и kmax получают значение 1, а p и pmax — значение 0).

Дополнительная проверка после цикла необходима для правильной обработки *последней* серии одинаковых элементов. Это связано с тем, что обработка каждой серии в цикле выполняется в тот момент, когда серия сменяется следующей, поэтому последняя серия в цикле обработана не будет.

Можно избавиться от необходимости выполнять дополнительную проверку после цикла, если анализировать серию не в момент ее завершения, а *при каждом увеличении ее длины*:

```
Task('Minmax27'); // Вариант 1а
var a := ReadArrInteger;
var kmax := 1;
var pmax := 0;
var k := 1;
var p := 0;
for var i := 1 to a.Length - 1 do
  if a[i - 1] = a[i] then
  begin
    k += 1;
    if k > kmax then
    begin
      kmax := k;
      pmax := p;
```

```
  end;
 end
 else
 begin
  k := 1;
  p := i;
 end;
Write(pmax + 1, kmax);
```

Здесь оператор if с условием k > kmax перенесен из ветки else в ветку then внешнего условного оператора.

Поскольку оба реализованных алгоритма имеют достаточно сложную логику, полезно промоделировать их работу, составив *трассировочную таблицу*. В качестве исходного набора будем использовать набор, который уже приводился ранее (для большей наглядности над каждым элементом записан его индекс):

```
i: 0 1 2 3 4 5 6 7 8 9
a: 0 1 1 0 0 0 1 0 0 0
```

Трассировочная таблица для варианта 1 при обработке данного набора приведена в таблице 1.6.

Таблица 1.6. *Трассировочная таблица для задачи Minmax27 (вариант 1)*

i	a[i – 1] = a[i]	k > kmax	kmax	pmax	k	p
			1	0	1	0
1	0 = 1 – False	1 > 1 – False			1	1
2	1 = 1 – True				2	
3	1 = 0 – False	2 > 1 – True	2	1	1	3
4	0 = 0 – True				2	
5	0 = 0 – True				3	
6	0 = 1 – False	3 > 2 – True	3	3	1	6
7	1 = 0 – False	1 > 3 – False			1	7
8	0 = 0 – True				2	
9	0 = 0 – True				3	
		3 > 3 – False				

Обратите внимание на то, что проверка k > kmax выполняется только в случае, когда условие a[i – 1] = a[i] является ложным, а изменение переменных kmax и pmax происходит только в случае, когда условие k > kmax оказывается истинным. Кроме того, проверка k > kmax выполняется еще один раз после завершения цикла.

После завершения алгоритма будут напечатаны числа 4 (= pmax + 1) и 3 (= kmax).

При составлении трассировочной таблицы для варианта 1а удобнее изменить порядок следования столбцов (см. таблицу 1.7).

В варианте 1а проверка k > kmax выполняется в случае, когда условие a[i – 1] = a[i] является истинным; изменение переменных kmax и pmax, как и в варианте 1, происходит только в случае, когда условие k > kmax оказывается истинным. В данном варианте дополнительная проверка k > kmax после завершения цикла не требуется.

Таблица 1.7. *Трассировочная таблица для задачи Minmax27 (вариант 1а)*

i	a[i – 1] = a[i]	k	p	k > kmax	kmax	pmax
		1	0		1	0
1	0 = 1 – False	1	1			
2	1 = 1 – True	2		2 > 1 – True	2	1
3	1 = 0 – False	1	3			
4	0 = 0 – True	2		2 > 2 – False		
5	0 = 0 – True	3		3 > 2 – True	3	3
6	0 = 1 – False	1	6			
7	1 = 0 – False	1	7			
8	0 = 0 – True	2		2 > 3 – False		
9	0 = 0 – True	3		3 > 3 – False		

При обработке указанного набора данных количество проверок k > kmax для каждого алгоритма оказалось одинаковым. Это случайное совпадение. При большом количестве коротких серий число проверок k > kmax будет больше в алгоритме 1, а в случае, если набор содержит небольшое количество длинных серий число проверок будет больше в алгоритме 1а.

Завершив обсуждение вариантов решения, основанных на переборе элементов исходного массива в цикле, зададимся вопросом: можно ли для этой задачи реализовать решение, основанное на применении запросов? Основываясь на уже проделанном анализе задачи, естественно ожидать, что если такое решение и существует, оно будет гораздо сложнее, чем подобные решения для ранее рассмотренных задач.

Тем не менее, приведем это решение, поскольку оно может рассматриваться как хорошая иллюстрация технологии программирования, основанной на запросах.

Прежде чем приводить текст программы с решением, опишем те преобразования исходного набора, которые позволят получить ответ. В качестве образца будем по-прежнему использовать следующий модельный набор:

0 1 1 0 0 0 1 0 0 0

При работе с последовательностями мы лишены возможность выполнять специальные действия для начального или конечного элемента. В то же время, в исходном наборе как первая, так и последняя серии одинаковых элементов являются особыми: перед первой и после последней серии нет элементов. Чтобы превратить эти особые серии в «обычные», проще

всего добавить в начало и конец нашего набора по «фиктивному» элементу, подобрав их значения так, чтобы они не испортили исходные серии. В нашем случае достаточно взять значения, отличные от 0 и 1, например 2 (в других ситуациях может оказаться удобным использовать варианты integer.MaxValue или integer.MinValue). Для добавления элемента 2 в начало и конец массива a можно использовать операцию + для массивов [1, п. 5.1]: Arr(2) + a + Arr(2).

Итак, первое преобразование состоит во введении фиктивных элементов, превращающих все исходные серии во «внутренние»:

2 0 1 1 0 0 0 1 0 0 0 2

Так как очевидно, что для решения задачи нам потребуется хранить информацию об индексах элементов, на следующем этапе добавим перед каждым элементом его индекс:

(0,2) (1,0) (2,1) (3,1) (4,0) (5,0) (6,0) (7,1) (8,0) (9,0) (10,0) (11,2)

Подобное преобразование мы уже использовали при решении задачи Minmax13, для него достаточно применить к исходному набору запрос Numerate(0). Заметим, что с исходными элементами набора мы фактически связали их *порядковые номера* (начинающиеся от 1), что нас вполне устраивает. Если бы для алгоритма было удобнее связать с исходными элементами их *индексы*, то достаточно было бы указать другой параметр в запросе Numerate: Numerate(–1).

Следующее преобразование является ключевым: мы должны распознать, где начинаются и где заканчиваются серии одинаковых элементов. В алгоритмах, использующих цикл, мы проверяли для этого значения *соседних* элементов массива. Для того чтобы реализовать аналогичную проверку при обработке последовательности, имеется универсальный прием, основанный на применении *последовательности пар* исходных элементов. Напомним, что для генерации последовательности пар предусмотрен специальный запрос Pairwise [1, п. 4.5], в котором можно явно указать способ формирования каждой пары.

В нашем случае нет необходимости объединять оба соседних элемента *вместе с их индексами* (понятно, что их индексы будут отличаться на 1), поэтому не будем включать в результат индекс (т. е. первое поле) *левого* элемента пары. Это означает, что в запросе Pairwise надо указать следующее лямбда-выражение: (e1, e2) -> (e1[1], e2[0], e2[1]) (значение e1[0] в результирующем кортеже не используется).

После выполнения данного запроса мы получим последовательность *троек*, которая будет иметь следующий вид:

(2,1,0) **(0,2,1)** (1,3,1) **(1,4,0)** (0,5,0) (0,6,0) **(0,7,1)** **(1,8,0)** (0,9,0) (0,10,0) **(0,11,2)**

Напомним смысл полей полученных троек e: e[0] — значение левого из двух соседних элементов последовательности, e[1] — индекс правого элемента, e[2] — значение правого элемента.

Для решения задачи нас интересуют только те тройки, у которых *значения* соседних элементов (т. е. значения первого и третьего поля) *различны* (в предыдущей последовательности эти тройки чисел выделены полужирным шрифтом). Для отбора таких троек можно применить запрос Where с лямбда-выражением e -> e[0] <> e[2]. В результате получаем следующую отфильтрованную последовательность:

(2,**1**,0) (0,**2**,1) (1,**4**,0) (0,**7**,1) (1,**8**,0) (0,**11**,2)

Легко заметить, что второе поле каждой тройки (оно выделено полужирным шрифтом) позволяет определить *начальную позицию* каждой серии (за исключением первой «фиктивной» серии, состоящей из единственного числа 2, которая нас не интересует). Кроме того, по третьему полю можно определить *значение* элементов, входящих в серию (хотя в нашей задаче это не требуется).

К сожалению, по отдельной тройке мы не можем определить *длину* серии. Однако длина легко определяется, если использовать информацию из следующей тройки: *для этого достаточно из второго поля, входящего в следующую тройку (т. е. начальной позиции следующей серии), вычесть второе поле, входящее в текущую тройку*. Например, вычисляя разность 2 – 1, мы получаем длину первой серии, разность 4 – 2 — длину второй серии, 7 – 4 — длину третьей серии и т. д.

Для доступа к соседним тройкам нам надо объединить их. Таким образом, нам опять потребуется прибегнуть к запросу Pairwise, теперь уже применив его к имеющимся тройкам чисел. В любой паре соседних троек вида ((e1[0], e1[1], e1[2]), (e2[0], e2[1], e2[2])) нас интересуют лишь значения e1[1] (это *начальная позиция* серии одинаковых элементов) и e2[1] – e1[1] (это *длина* данной серии). Поэтому в запросе Pairwise надо использовать лямбда-выражение вида (e1, e2) -> (e1[1], e2[1] – e1[1]). В результате выполнения второго запроса Pairwise мы получим следующую последовательность пар:

(**1**,1) (**2**,2) (**4**,3) (**7**,1) (**8**,3)

В этой последовательности сохранено полужирное выделение поля, определяющего начальную позицию каждой серии. Вторым полем является длина этой серии. Обратите внимание на то, что в результате последнего преобразования в последовательности пропала информация о последней «фиктивной» серии, состоящей из единственного элемента 2.

Для получения ответа нам достаточно найти первую пару полученной последовательности с *максимальным вторым полем*. В нашем случае это пара (4, 3). Напомним, что для нахождения элемента последовательности,

имеющего максимальное второе поле, следует использовать запрос MaxBy (ранее уже применявшийся при решении задачи Minmax13) с параметром — лямбда-выражением вида e -> e[1].

Итак, порядковый номер первой серии наибольшей длины равен 4, а длина этой серии равна 3. Задача решена. Нам осталось записать в виде цепочки все использованные ранее запросы:

```
Task('Minmax27'); // Вариант 3
var a := ReadArrInteger;
var b := (Arr(2) + a + Arr(2)).Numerate(0)
  .Pairwise((e1, e2) -> (e1[1], e2[0], e2[1]))
  .Where(e -> e[0] <> e[2])
  .Pairwise((e1, e2) -> (e1[1], e2[1] - e1[1]))
  .MaxBy(e -> e[1]);
Write(b[0], b[1]);
```

Использование сложных запросов, а также необходимость дополнять массив фиктивными элементами (что требует *двойного* копирования исходных элементов в памяти) приводит к тому, что данный вариант для больших массивов работает существенно медленнее, чем варианты, основанные на переборе элементов в цикле. Для проверки быстродействия алгоритмов использовались массивы размера size, которые генерировались с помощью функции ArrRandomInteger(size, 0, 1). Результаты численного эксперимента приведены в таблице 1.8.

Таблица 1.8. *Быстродействие вариантов решения задачи Minmax27*

Размер size	Вариант 1	Вариант 1а	Вариант 2
4000000	23	22	500
8000000	44	45	970
16000000	89	88	1984
32000000	178	175	3821

Глава 2. Списки

Стандартная библиотека платформы .NET включает ряд обобщенных классов, которые в ряде случаев оказываются более удобным средством обработки данных, чем массивы и последовательности. Все эти классы можно использовать в PascalABC.NET без каких-либо специальных действий по подключению дополнительных библиотек. В главах 2–5 дается обзор возможностей основных обобщенных классов для хранения наборов данных.

2.1. Список на базе массива List<T> и его свойства

Обобщенный класс List<T> обладает всеми возможностями массива (в частности, доступом к элементам по их индексам) и дополнительно предоставляет средства для вставки и удаления элементов в любой позиции. Подобные средства характерны для структур, называемых *списками*, поэтому данный класс также получил имя List («список»). В реализации класса List<T> данные хранятся во внутреннем массиве, что обеспечивает быстрый доступ к элементам по их индексам, но приводит к тому, что операции вставки и удаления производятся сравнительно долго (время линейно зависит от размера хранящегося набора данных). В библиотеке .NET предусмотрен другой вариант реализации списка — на базе цепочки связанных узлов, с быстрыми операциями вставки и удаления; соответствующий класс называется LinkedList<T> и будет рассмотрен далее в этой главе (см. п. 2.4–2.6).

Несмотря на неэффективность реализации операций вставки и удаления по сравнению с LinkedList<T>, класс List<T> используется чаще, прежде всего, благодаря быстрому доступу к элементам по индексу. Кроме того, в нем быстро выполняется *добавление* новых элементов (в конец набора), а также предусмотрен богатый набор методов, связанных с поиском и преобразованием данных.

В дальнейшем класс List<T> (список на базе массива) будем для краткости называть просто *списком*, а класс LinkedList<T> (список на базе цепочки связанных узлов) будем называть *двусвязным списком*.

Список имеет три свойства: Count, Capacity и Item. Свойство Count имеет тип integer, доступно только для чтения и хранит *размер* списка (таким об-

разом, оно аналогично свойству Length для массива). Свойство Item является *свойством-индексатором* и позволяет обращаться по индексу к элементам списка. Данное свойство практически не используется, так как вместо выражения a.Item[i] для доступа к элементу с индексом i из списка a можно использовать выражение a[i] — такое же, как и для доступа к элементам массива.

Важной характеристикой списка, отсутствующей у динамического массива, является *емкость* (capacity), для доступа к которой предусмотрено одноименное свойство. Емкость определяет текущий размер того внутреннего массива, в котором хранятся элементы списка. Это означает, что емкость не может быть меньше размера списка, однако часто она превышает этот размер. Если емкость превышает размер, то добавление новых элементов в конец списка выполняется быстро, так как для них не нужно специально выделять место (достаточно дописать новые элементы в конец внутреннего массива и откорректировать свойство Count). Свойство Capacity, определяющее емкость списка, имеет тип integer и, в отличие от свойства Count, является изменяемым. Следует, однако, учитывать, что изменение емкости является длительной операцией (подобно операции изменения размера динамического массива), поскольку требует выделения памяти под новый внутренний массив (имеющий размер, равный новому значению емкости) и копирования всех остающихся данных из старого массива в новый.

Емкость может изменяться и автоматически — в том случае, если при очередном добавлении или вставке элементов оказывается, что текущей емкости недостаточно. В этой ситуации прежняя емкость *удваивается* (это позволяет уменьшить число операций по увеличению емкости, если в список добавляется много новых элементов).

2.2. Создание и преобразование списка List<T>

Перед началом работы со списком (как и с динамическим массивом) его надо *инициализировать*. Инициализация списка обычно выполняется с помощью конструктора вида new List<T>. Имеются три варианта конструктора:

- без параметров,
- с параметром capacity типа integer,
- с параметром collection типа sequence of T.

При вызове конструктора без параметров для созданного списка устанавливается емкость, равная 0, которая при добавлении первого элемента изменяется на 4 (далее при ее нехватке она удваивается — см. п. 2.1). Если указать в конструкторе параметр capacity, то для списка сразу будет установлена указанная емкость. При использовании конструктора без парамет-

ров или с параметром capacity *размер* списка полагается равным 0 (т. е. созданный таким образом список не содержит элементов).

Если вызван конструктор с параметром collection, то в список копируются все элементы из указанной *коллекции* (это может быть не только последовательность, но и любая структура, которая допускает неявное преобразование в последовательность, например динамический массив или другой список). Емкость полученного списка зависит от типа коллекции. При указании параметра collection типа массив или List емкость будет равна *размеру* исходной коллекции; если же параметром collection является последовательность, то емкость будет равна *наименьшей из степеней числа 2, которая больше или равна размеру последовательности*.

Замечание. Из этого правила имеется одно исключение: если последовательность получена путем *неявного преобразования массива или списка*, то при ее передаче в качестве параметра collection она будет вести себя подобно массиву или списку, поэтому емкость созданного списка будет в точности равна размеру исходной последовательности. В частности, так будет вести себя последовательность, созданная с помощью короткой функции Seq [1, п. 3.4] (это объясняется тем, что список параметров функции Seq интерпретируется как *массив*, на базе которого и создается последовательность).

Последовательность можно преобразовать в список и другим способом: с помощью запроса ToList или короткой функции Lst [1, п. 4.6]. Таким образом, для создания списка на основе, например, целочисленной последовательности можно использовать три конструкции:

```
var a := Seq(2, 3, 4);
var b1 := new List<integer>(a);
var b2 := a.ToList;
var b3 := Lst(a);
Print(b1, b2, b3); // [2,3,4] [2,3,4] [2,3,4]
```

Две последние конструкции удобны еще и тем, что не требуют указания типа элементов списка (который выводится из типа элементов последовательности). В короткой функции Lst можно вместо исходной последовательности указать *список элементов* через запятую; запрос ToList можно применять к *любым коллекциям*. Относительно емкости полученного списка сохраняется правило, описанное ранее: если a является «обычной» последовательностью, то a.ToList и Lst(a) возвратят список, емкость которого будет равна наименьшей степени числа 2, большей или равной размеру последовательности a; если же a является последовательностью, полученной из массива или списка (см. замечание), то a.ToList и Lst(a) вернут список, емкость которого будет равна его размеру.

Специальных генераторов или функций ввода для списков не предусмотрено, поскольку для создания списка можно использовать любой генератор или функцию ввода для последовательности [1, п. 3.4, 3.5, 3.7], после чего преобразовать полученную последовательность в список запросом ToList или короткой функцией Lst.

Для вывода списков можно использовать те же средства, что и для вывода массивов и последовательностей (см. [1, п. 3.2, 3.5], а также приведенный выше пример).

Список имеет три группы методов преобразования, отсутствующих у массива: это *добавление*, *вставка* и *удаление* элементов. Именно из-за этих методов в ряде случаев применение списка оказывается более удобным, чем применение массива.

При последующем описании методов предполагается, что элементы списка имеют тип T. Методы, отсутствующие в стандартной библиотеке платформы .NET, но имеющиеся в стандартной библиотеке системы PascalABC.NET, помечаются звездочкой *.

Добавление и вставка Методы List<T>

Add(item: T)
AddRange(collection: sequence of T)
Insert(index: integer; item: T)
InsertRange(index: integer; collection: sequence of T)

Метод Add добавляет в конец списка один элемент item, а метод AddRange — все элементы из коллекции collection. Операция добавления выполняется быстро, если текущей емкости списка достаточно для хранения добавленных элементов; в противном случае емкость автоматически увеличивается (удваивается), что требует дополнительного времени и памяти. Вместо вызова a.Add(e) можно использовать выражение a += e.

Метод Insert вставляет элемент item в позицию списка с индексом index; при этом индексы всех последующих элементов списка увеличиваются на 1. Метод InsertRange вставляет, начиная с позиции index, все элементы коллекции collection, сдвигая на нужное расстояние все последующие элементы списка. Поскольку операция вставки приводит к сдвигу оставшейся части списка, время ее выполнения зависит от размеров этой части. Кроме того, при вставке, как и при добавлении элементов, может потребоваться увеличить емкость списка.

В качестве параметра index можно указать текущий размер списка (свойство Count); в этом случае вставка равносильна операции добавления. Если index меньше 0 или больше Count, то возбуждается исключение.

Успешное выполнение любой из описанных операций приводит к увеличению размера списка. Все эти операции являются процедурами, т. е. не возвращают значений.

Удаление Методы List<T>

Remove(item: T): boolean
RemoveAll(pred: T -> boolean): integer
RemoveAt(index: integer)
RemoveRange(index, count: integer)
Clear
* **RemoveLast**

Метод Remove удаляет из списка первое вхождение элемента со значением item и возвращает True, если элемент был удален, и False, если элемент с требуемым значением не был найден.

Метод RemoveAll удаляет из списка все элементы, удовлетворяющие предикату pred, и возвращает количество удаленных элементов.

Метод RemoveAt удаляет из списка элемент с индексом index, а метод RemoveRange — диапазон из count элементов, начиная с элемента с индексом index. Если указанный индекс или диапазон не входит в список, то возбуждается исключение.

Метод Clear удаляет из списка все элементы; при этом свойство Count списка становится равным 0.

Метод RemoveLast удаляет последний элемент непустого списка (если список пуст, то возбуждается исключение).

Ни один из методов удаления не изменяет емкость списка, поскольку изменение емкости сопряжено с дополнительными расходами времени и памяти. Однако предусмотрен метод TrimExcess без параметров, позволяющий уменьшить емкость, приведя ее в соответствие с текущим размером списка. Метод TrimExcess имеет смысл вызывать в ситуации, когда уже известно, что новые элементы добавляться к списку не будут. Следует иметь в виду, что если текущая емкость *несущественно* превышает текущий размер, то метод TrimExcess не выполняет никаких действий, поскольку выигрыш от уменьшения емкости будет несопоставим со связанными с этим уменьшением расходами памяти и времени.

2.3. Дополнительные возможности списка List<T>

Для списков предусмотрены методы преобразования, аналогичные тем, которые имеются у массивов (см. их описание в [1, п. 5.4]):

Различные преобразования Методы List<T>

ConvertAll(conv: T -> T1): List<T1>
* **Shuffle**: List<T1>
Sort

Для сортировки списка a, как и для сортировки массива, можно также использовать *процедуру* вида Sort(a).

Для списков имеется *метод* Reverse, обеспечивающий инвертирование списка или его части (для массивов указанное действие можно выполнить с помощью *процедуры* Reverse [1, п. 5.4]):

Инвертирование Метод List<T>

Reverse(**[**start, count: integer**]**)

Напомним, что в полужирных квадратных скобках указываются необязательные параметры подпрограмм. Если параметры start и count не указаны, то инвертируется весь список, а если указаны — то лишь count его элементов, начиная с элемента с индексом start.

В п. 2.2 уже отмечалось, что для преобразования массива в список достаточно указать массив в качестве параметра конструктора или применить к нему запрос ToList. Обратное преобразование (списка в массив) можно выполнить с помощью двух методов:

Преобразование или копирование в массив Методы List<T>

ToArray: array of T

CopyTo(arr: array of T**[**; arrStart: integer**]**)

Метод ToArray создает и возвращает массив, содержащий те же элементы, что и вызвавший его список. Метод CopyTo является процедурой и позволяет скопировать элементы списка в имеющийся массив arr; при этом можно указать необязательный параметр arrStart, определяющий индекс в массиве, начиная с которого будут записываться элементы списка. Напомним, что метод CopyTo имеется и у массива [1, п. 5.1].

Для списков реализованы методы, позволяющие получить *срезы*:

Получение срезов Методы List<T>

* **Slice**(from, step: integer**[**; count: integer**]**): List<T>

GetRange(from, count: integer): List<T>

Метод Slice обсуждался в [1, п. 3.7]; напомним, что его можно применять также к массивам и последовательностям (для последовательностей имеется ограничение: параметр step должен быть положительным). Метод GetRange реализован только для списков; он возвращает в виде нового списка диапазон из count элементов, начинающийся с индекса from (т. е. этот метод реализует частный случай среза Slice — когда step равен 1).

Наконец, необходимо отметить, что в списке реализовано много методов *поиска*. Большинство из них выполняется в точности так же, как и одноименные методы поиска для массивов, описанные в [1, п. 5.3]: BinarySearch, Find, FindLast, FindAll (для списков этот метод возвращает все найденные элементы в виде списка), FindIndex, FindLastIndex, IndexOf, LastIndexOf. Имеются также два метода-*квантификатора*, отсутствующие у массивов (хотя их аналоги есть среди запросов для последовательностей — см. [1, п. 4.1]):

Квантификаторы Методы List<T>

Exists(pred: T -> boolean): boolean

TrueForAll(pred: T -> boolean): boolean

Метод Exists возвращает True, если в списке найдется *хотя бы один* элемент, удовлетворяющий предикату pred, а метод TrueForAll возвращает True, если предикату pred удовлетворяют *все* элементы списка.

В [1, п. 5.3] отмечалось, что для поиска значения x в массиве a можно использовать *операцию* in, которая записывается в виде x in a. Для списков операция in не определена, однако вместо нее можно использовать метод a.Contains(x), возвращающий True, если список a содержит элемент со значением x.

В заключение надо подчеркнуть, что список List<T> является коллекцией, т. е. может быть неявно преобразован в последовательность sequence of T, и поэтому для него доступны все запросы последовательностей, описанные в [1, гл. 4].

2.4. Двусвязный список LinkedList<T> и его свойства. Тип LinkedListNode<T>

Обобщенный класс LinkedList<T> (*двусвязный список*) хранит данные в виде цепочки связанных *узлов* и имеет средства, позволяющие проходить эту цепочку как в прямом, так и в обратном направлении. Благодаря подобной организации своих данных двусвязный список может очень быстро выполнять операции вставки и удаления узлов. Однако у него отсутствуют средства прямого доступа к узлам (по индексу), что серьезно ограничивает его возможности по сравнению со списком на базе массива List<T>, рассмотренным ранее в этой главе. Тем не менее, в некоторых ситуациях (связанных с многократным выполнением вставок и удалений и не требующих прямого доступа к элементам) использование двусвязного списка позволяет получить наиболее эффективные алгоритмы.

Двусвязный список LinkedList<T> имеет три свойства: Count, First и Last. Все эти свойства доступны только для чтения. Свойство Count типа integer позволяет определить текущий размер двусвязного списка, свойства First и Last типа LinkedListNode<T> позволяют обратиться к первому и последнему узлу списка. Если список пуст, то свойства First и Last имеют значение nil (т. е. равны «пустой» ссылке).

Тип LinkedListNode<T>, в свою очередь, имеет четыре свойства: List, Next, Previous и Value, из которых первые три доступны только для чтения и являются ссылками (т. е. хранят адреса других объектов). Свойство List хранит адрес списка, содержащего данный узел, свойство Next содержит адрес узла, следующего в списке за данным узлом (если узел является последним в списке, то его свойство Next равно nil), свойство Previous содер-

жит адрес предыдущего узла в списке (если узел является первым в списке, то его свойство Previous равно nil).

Свойство Value узла содержит собственно элемент данных типа T, для хранения которых и используется список. Это свойство можно изменять.

Для создания узла можно использовать конструктор класса LinkedListNode<T> с одним параметром — значением свойства Value. Все прочие свойства для созданного узла будут равны nil; их значение изменяется автоматически при добавлении этого узла в какой-либо список (см. п. 2.5).

Зная первый (First) и последний (Last) узлы списка и пользуясь свойствами Next и Previous узлов, можно организовывать обход всей цепочки узлов как в прямом (от начала к концу), так и в обратном (от конца к началу) направлении. Возможности прямого доступа к узлу двусвязного списка отсутствуют; в частности, к узлу нельзя обратиться по его индексу.

2.5. Создание и преобразование двусвязного списка

Для инициализации двусвязного списка типа LinkedList<T> предусмотрены два варианта конструктора:

- без параметров,
- с параметром collection типа sequence of T.

При вызове конструктора без параметров создается пустой список (у которого свойство Count равно 0, а свойства First и Last равны nil). Если вызван конструктор с параметром collection, то в двусвязный список копируются все элементы из указанной коллекции (последовательности или структуры данных, которая может быть неявно преобразована в последовательность).

Последовательность можно преобразовать в двусвязный список с помощью запроса ToLinkedList или короткой функции LLst [1, п. 4.6]:

```
var a := Seq(2,3,4);
var b1 := new LinkedList<integer>(a);
var b2 := a.ToLinkedList;
var b3 := LLst(a);
Print(b1, b2, b3); // [2,3,4] [2,3,4] [2,3,4]
```

Для двусвязных списков справедливы все замечания относительно генерации, ввода и вывода, сделанные по поводу списков List<T> в п. 2.2.

Методы, предназначенные для вставки и удаления узлов в двусвязный список, существенно отличаются от аналогичных методов для списка List<T>. Это связано, во-первых, с тем, что в двусвязный список можно добавлять новые узлы с обеих сторон (перед первым или после последнего узла) и, во-вторых, с отсутствием возможности обращаться к узлам двусвязного списка по индексу.

При последующем описании методов предполагается, что элементы двусвязного списка имеют тип T.

Добавление и вставка Методы LinkedList<T>

AddFirst(value: T): LinkedListNode<T>
AddFirst(newNode: LinkedListNode<T>)

AddLast(value: T): LinkedListNode<T>
AddLast(newNode: LinkedListNode<T>)

AddAfter(node: LinkedListNode<T>; value: T): LinkedListNode<T>
AddAfter(node, newNode: LinkedListNode<T>)

AddBefore(node: LinkedListNode<T>; value: T): LinkedListNode<T>
AddBefore(node, newNode: LinkedListNode<T>)

Метод AddFirst вставляет новый узел в начало списка, метод AddLast — в его конец, методы AddAfter и AddBefore вставляют новый узел после или, соответственно, перед узлом node (узел node должен принадлежать тому списку, для которого вызван метод, иначе будет возбуждено исключение).

Каждый из этих методов реализован в двух вариантах. В варианте, для которого параметром является значение value типа T, автоматически создается новый узел с этим значением, который и добавляется в нужную позицию списка (кроме того, созданный узел является возвращаемым значением метода). Вариант, который принимает в качестве параметра уже имеющийся узел newNode, является *процедурой*, т. е. ничего не возвращает. Добавляемый узел newNode не должен принадлежать какому-либо списку (т. е. его свойство List должно быть равно nil), в противном случае возбуждается исключение.

Удаление Методы LinkedList<T>

Remove(value: T): boolean
Remove(node: LinkedListNode<T>)
RemoveFirst
RemoveLast
Clear

Метод Remove с параметром value типа T удаляет из списка *первый* узел со значением value. Метод возвращает True в случае удаления узла и False, если в списке отсутствует элемент со значением value.

Метод Remove с параметром node типа LinkedListNode<T> удаляет из списка указанный узел node. Если узел node не принадлежит списку, то возбуждается исключение.

Методы RemoveFirst, RemoveLast и Clear удаляют соответственно первый, последний и все элементы списка. При попытке вызвать методы RemoveFirst и RemoveLast для пустого списка возбуждается исключение.

2.6. Дополнительные возможности двусвязного списка

Для двусвязных списков LinkedList<T> имеется существенно меньше дополнительных методов, чем для списков List<T> на основе массива.

Имеется метод для копирования всех элементов двусвязного списка в существующий массив arr, начиная с индекса arrStart:

Копирование в массив	Метод LinkedList<T>

CopyTo(arr: array of T; arrStart: integer)

Обратите внимание на то, что в реализации метода CopyTo для двусвязного списка (как и для массива) параметр arrStart является *обязательным*.

Имеются три метода поиска:

Поиск	Методы LinkedList<T>

Contains(value: T): boolean
Find(value: T): LinkedListNode<T>
FindLast(value: T): LinkedListNode<T>

Метод Contains, подобно одноименному методу для списка List<T>, возвращает True, если в двусвязном списке имеется узел со значением value. Методы Find и FindLast возвращают первый или, соответственно, последний узел списка, имеющий значение value. Если в списке отсутствуют узлы с требуемым значением, то методы Find и FindLast возвращают нулевую ссылку nil.

Двусвязный список LinkedList<T>, как и список на базе массива List<T>, может быть неявно преобразован в последовательность sequence of T, и поэтому для него доступны все запросы последовательностей, описанные в [1, гл. 4].

2.7. Использование списков при решении задач

Списки особенно полезны в ситуациях, когда требуется изменить размер исходного набора данных, удалив из него некоторые элементы или вставив новые. Однако списки можно успешно применять и в алгоритмах, не требующих изменения размера. Примерами таких алгоритмов являются алгоритмы, реализующие *сдвиги* элементов. Сдвиги могут быть левыми или правыми, простыми или циклическими, на одну или на несколько позиций; все эти случаи представлены в группе Array набором задач Array79–Array86.

Мы рассмотрим задачу **Array85**, в которой требуется выполнить правый циклический сдвиг элементов исходного набора вещественных чисел a на k позиций. Если через a[i] обозначить i-й элемент исходного набора, считая, что нумерация начинается от 0, и через n обозначить размер набора a,

то при описанном выше сдвиге a[0] перейдет в a[k], a[1] — в a[k + 1], …, a[n – 1] — в a[k – 1].

В задаче предполагается, что k меньше n и может изменяться лишь в пределах от 1 до 4. Второе условие (1 ≤ k ≤ 4) может оказаться полезным при реализации алгоритма с применением вспомогательного *статического* массива, однако мы не будем использовать эту устаревшую структуру, поэтому в алгоритме нам не потребуется учитывать указанное ограничение. Реализуем алгоритм правого циклического сдвига на k позиций, предполагая, что число k может изменяться в пределах от 1 до n – 1.

Замечание. Описанные далее варианты решения легко модифицировать для случая k ≥ n, поскольку в этом случае достаточно выполнить сдвиг на k mod n позиций в том же направлении (в частности, если k делится на n, то сдвиг выполнять не требуется).

Начнем с вариантов решений, использующих динамические массивы.

```
Task('Array85'); // Вариант 1
var a := ReadArrReal;
var k := ReadInteger;
for var j := 1 to k do
begin
  var x := a[a.Length - 1];
  for var i := a.Length - 2 downto 0 do
    a[i + 1] := a[i];
  a[0] := x;
end;
a.Write;
```

Это самый экономный вариант с точки зрения использования памяти, поскольку, кроме двух переменных цикла, он использует единственную переменную x (типа real). В то же время, это *очень долгий* алгоритм, так как в нем правый циклический сдвиг выполняется k раз (каждый раз производится сдвиг на 1 позицию). Обратите внимание на то, что правый сдвиг необходимо выполнять *с конца* массива, чтобы не «испортить» те элементы, которые еще не перемещены на новую позицию. Переменная x используется для сохранения последнего элемента, который (при сдвиге на 1 позицию) надо переместить в начальный элемент массива.

За счет сравнительно небольшого дополнительного расхода памяти решение можно существенно ускорить:

```
Task('Array85'); // Вариант 2
var a := ReadArrReal;
var k := ReadInteger;
var x := a[a.Length-k:];
for var i := a.Length - k - 1 downto 0 do
```

```
  a[i + k] := a[i];
x.CopyTo(a, 0);
a.Write;
```

Здесь мы используем вспомогательный *массив* x для хранения последних k элементов исходного массива (т. е. тех элементов, которые при циклическом сдвиге надо переместить в начало). Для формирования массива x удобно использовать синтаксис *среза*, описанный в [1, п. 3.7]. В цикле теперь сразу выполняется сдвиг каждого элемента на k позиций вправо (по-прежнему требуется перебирать элементы с конца массива, причем перебираются только элементы, не попавшие в x). После завершения цикла останется скопировать элементы из массива x в начало исходного массива a. Наиболее эффективно это можно выполнить с помощью стандартного метода CopyTo [1, п. 5.1].

Теперь обратимся к вариантам решения, использующим класс List<T>. Используя предусмотренные в этом классе методы InsertRange, RemoveRange, GetRange для работы с *диапазонами* элементов (см. п. 2.2–2.3), мы можем получить решение, состоящее всего из двух операторов (не считая операторов, обеспечивающих ввод и вывод):

```
Task('Array85'); // Вариант 3
var a := ReadSeqReal.ToList;
var k := ReadInteger;
a.InsertRange(0, a.GetRange(a.Count - k, k));
a.RemoveRange(a.Count - k, k);
a.Write;
```

Для ввода исходного набора мы используем функцию ReadSeqReal, возвращающую этот набор в виде последовательности, к которой применяем запрос ToList. Для вывода по-прежнему используем метод Write, реализованный в задачнике Programming Taskbook, поскольку он может применяться к любым коллекциям (т. е. наборам данных, которые могут быть неявно преобразованы в последовательности).

Для определения размера списка мы используем его свойство Count (аналог свойства Length массивов).

Приведенное решение на самом деле тоже выполняет вставку элементов в массив (а именно, во внутренний массив списка List), однако можно ожидать, что метод InsertRange реализует эту вставку более эффективно, чем «обычный» алгоритм в виде цикла for, использованный нами в вариантах 1 и 2.

Кроме того, может показаться, что этот алгоритм максимально эффективно использует и память, поскольку в нем не указываются никакие дополнительные переменные. Однако это не так. Во-первых, метод GetRange возвращает *новый* список (размера k). И, во-вторых, следует помнить, что

при добавлении в список новых данных может потребоваться увеличить его емкость, что приведет к *удвоению* предыдущей емкости и выделении соответствующего количества дополнительной памяти для внутреннего массива (см. п. 2.1). Если предположить, что созданный список имеет емкость, совпадающую с его размером, то добавление даже одного нового элемента приведет к удвоению емкости.

Избежать подобного «лишнего» удвоения можно, если *вначале* удалить из списка требуемые элементы, а лишь потом добавить новые. Мы получаем новый вариант алгоритма для списка List, который на один оператор длиннее, но зато эффективнее использует память:

```
Task('Array85'); // Вариант 4
var a := ReadSeqReal.ToList;
var k := ReadInteger;
var x := a.GetRange(a.Count - k, k);
a.RemoveRange(a.Count - k, k);
a.InsertRange(0, x);
a.Write;
```

Здесь мы связываем диапазон из k последних элементов исходного набора с переменной x, которую затем используем в методе вставки InsertRange. Поскольку на каждом этапе алгоритма размер исходного списка не превышает исходный, увеличения емкости не потребуется.

Обратите внимание также на то, что присваивание переменной x диапазона GetRange не приведет к тому, что все элементы этого диапазона будут скопированы в новый участок памяти. Метод GetRange возвращает объект типа List, а структура данных List (как и все другие структуры, рассматриваемые в настоящем пособии) является, подобно динамическому массиву, *ссылочным* типом данных, и при ее присваивании копируется лишь ссылка, т. е. адрес, по которому этот объект размещается в памяти (по поводу особенностей присваивания ссылочных типов данных см. [1, п. 5.1]). Введение переменной x увеличит размер нашей программы лишь на несколько бит, требуемых для хранения адреса, и в этой переменной будет храниться адрес той же структуры, которая была создана методом GetRange.

Таким образом, благодаря применению списков List<T> нам удалось получить короткие и наглядные варианты решения задачи, скорость работы которых, во всяком случае, не должна быть ниже, чем скорость варианта 2 для массивов (и будет намного выше, чем скорость варианта 1).

Однако во всех рассмотренных вариантах при выполнении сдвига все же происходит копирование на новую позицию *всех* элементов исходного набора данных, поскольку внутреннее хранение данных в списке List<T> организовано в виде массива. Чтобы вставка или удаление элемента «не

затронула» остальные элементы набора, его надо оформить в виде двусвязного списка LinkedList<T>. Можно ожидать, что при использовании двусвязного списка мы получим более эффективный вариант решения:

```
Task('Array85'); // Вариант 5
var a := ReadSeqReal.ToLinkedList;
var k := ReadInteger;
for var j := 1 to k do
begin
  a.AddFirst(a.Last.Value);
  a.RemoveLast;
end;
a.Write;
```

В данном варианте приходится использовать цикл, так как методы, связанные с удалением и вставкой для двусвязного списка, не могут работать с *диапазонами* элементов. Заметим, что в этом варианте не используется дополнительная память (кроме переменной цикла j). Однако утверждать, что данный вариант самый эффективный по использованию памяти, все же нельзя, так как для хранения исходного набора данных в виде двусвязного списка требуется больше памяти, чем для его хранения в виде массива или списка List. Напомним, что двусвязный список хранится в виде связанной цепочки узлов, а каждый узел является объектом, в котором, помимо самого элемента данных (свойство Value), содержатся ссылки на следующий (Next) и предыдущий (Previous) элемент списка, а также на список в целом (свойство List). Суммарный размер памяти, необходимый для хранения этих служебных данных, может *превосходить* размер, необходимый для хранения самого элемента данных (например, если для хранения ссылки используется 4 байта, то свойства Next, Previous и List дадут в сумме 12 байт, тогда как тип real имеет размер 8 байт). Значит, размер структуры LinkedList, содержащей исходный набор данных, будет *более чем в два раза* превосходить размер массива или списка List, содержащих такой же набор.

Мы уже привыкли к тому, что многие задачи, связанные с обработкой наборов данных, могут быть решены с применением запросов для последовательностей, описанных в [1, гл. 4]. При этом получаются короткие и наглядные варианты решений, которые, к тому же, в ряде случаев являются очень эффективными. Для задачи, связанной со сдвигом, тоже существует короткий вариант решения, использующий запросы:

```
Task('Array85'); // Вариант 6
var a := ReadArrReal;
var k := ReadInteger;
a := a.TakeLast(k).Concat(a.Take(a.Length - k)).ToArray;
a.Write;
```

Этот вариант решения состоит из *единственного* оператора (не считая операторов ввода и вывода). В нем использовано следующее наблюдение: для выполнения правого циклического сдвига на k позиций достаточно *поменять местами* две части массива: последнюю часть из k элементов и первую часть, содержащую остальные элементы. Для получения этих частей мы используем запросы TakeLast и Take, а для их объединения в новом порядке — запрос Concat (все эти запросы описаны в [1, п. 4.2]).

Если мы вспомним, что для получения частей массива можно использовать срезы (мы уже использовали их в варианте 2), а для объединения массивов — операцию +, то мы сможем переписать предыдущий вариант, заменив запросы на операции получения среза и объединения массивов:

```
Task('Array85'); // Вариант 7
var a := ReadArrReal;
var k := ReadInteger;
a := a[a.Length - k:] + a[:a.Length - k];
a.Write;
```

Это самое короткое по размеру кода решение. Впрочем, что касается размеров используемой памяти, это решение является *наименее* эффективным из всех рассмотренных. Вспомним, что любая операция среза создает в памяти новый массив (содержащий указанный срез) и, кроме того, новый массив создается и в результате выполнения операции + для массивов. Таким образом, в данном алгоритме будет использована вспомогательная память, в *два раза* превосходящая память, необходимую для хранения исходного массива (если считать, что в исходном массиве содержится N элементов, то N элементов *в сумме* будут храниться в двух созданных срезах и, кроме того, N элементов будут храниться в массиве, созданном при выполнении операции +).

Вариант 6 использует память более экономно, поскольку запросы TakeLast, Take и Concat *сами по себе* не приводят к выделению памяти. Однако при вызове метода ToArray созданная последовательность преобразуется в массив, и на этом этапе выделяется память для хранения элементов созданного массива. Таким образом, этот вариант тоже требует выделения большого количества дополнительной памяти (размер которой *совпадает* с размером исходного набора данных). Из всех предыдущих вариантов такое количество дополнительной памяти использовали только вариант 3 (в котором происходило удвоение емкости исходного списка) и вариант 4 (из-за дополнительных расходов на хранение связанных узлов).

В заключение рассмотрим еще один вариант решения, основанный на идее *инвертирования* исходного массива:

```
Task('Array85'); // Вариант 8
var a := ReadArrReal;
```

```
var k := ReadInteger;
Reverse(a, a.Length - k, k);
Reverse(a, 0, a.Length - k);
Reverse(a);
a.Write;
```

В этом интересном варианте решения вначале инвертируется последняя часть массива, содержащая k элементов, затем инвертируется остальная часть массива, а в конце еще раз инвертируется весь массив. Нетрудно убедиться в том, что в результате мы действительно получим массив, в котором выполнен правый циклический сдвиг на k позиций. Для инвертирования массива (или его части) мы использовали стандартную процедуру Reverse [1, п. 3.7]. Помимо краткости и наглядности полученного решения, следует отметить его максимальную эффективность в отношении использования памяти, поскольку в алгоритме инвертирования применяется единственная вспомогательная переменная, тип которой совпадает с типом элементов массива. Подобная эффективность по памяти достигалась только в варианте 1. При этом в отношении быстродействия вариант 8 должен существенным образом превосходить вариант 1 и лишь незначительно уступать наиболее быстрым вариантам решения. Еще одной особенностью варианта 8, а также вариантов 6 и 7, является то, что их быстродействие не зависит от величины сдвига k.

Итак, мы рассмотрели восемь вариантов решения задачи Array85, отличающихся и по размеру кода (и его наглядности), и по количеству используемой памяти, и по быстродействию. Если оценить размер памяти можно, учитывая свойства используемых структур данных, то для оценки быстродействия следует провести численный эксперимент, обработав с помощью описанных алгоритмов очень большие массивы. Содержимое обрабатываемого массива роли не играет, так как оно никак не влияет на работу алгоритма. Для генерации исходных массивов можно использовать функцию ArrRandomReal(size), где size равен размеру массива; для генерации исходных списков — функцию SeqRandomReal(size), к которой дополнительно применяется запрос ToList или ToLinkedList. Чтобы при выполнении численного эксперимента не возникали побочные эффекты, связанные, например, с работой сборщика мусора, желательно при каждом запуске программы испытывать один вариант алгоритма на одном наборе исходных данных.

В приведенной ниже таблице 2.1 объединены результаты тестирования алгоритмов 1–7 для наборов данных размера 1, 4 и 8 миллионов элементов (количество элементов указывается в столбце «Размер»). В качестве k использовались два значения: 1000 и 100000. Время подсчитывалось только для самого алгоритма, т. е. первый вызов функции MillisecondsDelta выполнялся после завершения генерации данных, а второй, результат ко-

торого выводился на печать, — после завершения алгоритма. Все программы запускались в режиме релиза.

В ячейках таблицы указывается по два числа, разделенных двоеточием: время работы алгоритма (в миллисекундах) для k = 1000 и k = 100000. Исключение сделано для алгоритма 1, в котором расчет времени для k = 100000 не проводился (понятно, что это время было бы примерно в 100 раз больше).

Таблица 2.1. *Быстродействие вариантов решения задачи Array85*

Размер	Вар. 1	Вар. 2	Вар. 3	Вар. 4	Вар. 5	Вар. 6	Вар. 7	Вар. 8
1000000	2184	4:7	2:8	2:3	2:22	61:63	13:12	4:3
4000000	9012	11:20	9:19	9:19	2:18	206:207	41:41	13:13
8000000	17577	20:36	17:32	17:32	2:21	399:402	81:78	27:27

Мы видим, что самым медленным алгоритмом (причем *очень* медленным), как и следовало ожидать, является алгоритм 1. Самым быстрым алгоритмом для k = 1000 является алгоритм 5, использующий двусвязный список. При k = 100000 время работы этого алгоритма ухудшается (из-за существенного увеличения количества итераций) и приближается к времени работы алгоритмов 3 и 4, использующих список на основе массива. Естественно ожидать, что как при k = 1000, так и при k = 100000 время работы алгоритма 5 для массивов любого размера будет одинаковым, и это практически так (заметим, что только этот вариант обладает отмеченной особенностью). Вариант 2, использующий массив, совсем немного уступает вариантам 3 и 4. Вариант 8 также является одним из наиболее быстродействующих, особенно для случая k = 100000. Как уже было отмечено, для варианта 8, как и вариантов 6 и 7, время работы не зависит от значения k.

Должно удивлять почти полное совпадение времени для алгоритмов 3 и 4, поскольку в алгоритме 3, как мы отмечали выше, может увеличиваться емкость исходного списка, что для списков большого размера приводило бы к большому количеству дополнительных операций копирования. Причина совпадения времени кроется в способе создания исходного списка, для которого мы использовали генератор *последовательности* SeqRandomReal(size), применяя к нему метод ToList. Как было отмечено в п. 2.2, в этом случае созданный список будет иметь емкость, равную наименьшей степени числа 2, большей или равной числу size (в этом можно убедиться, выведя на экран значение свойства Capacity сразу после создания списка). В нашем случае емкость имела значения, равные соответственно 1048576, 4194304 и 8388608. Полученной емкости было достаточно для выполнения алгоритма 3 без ее увеличения во всех случаях, кроме случая, при котором массив имел размер 1000000, а k было равно 100000. И действительно, именно в этом случае эксперимент показал более чем двукратное увеличение времени работы алгоритма 3 по сравнением с алгоритмом 4.

Чтобы промоделировать ситуацию, при которой в алгоритме 3 *всегда* будет требоваться увеличение емкости, достаточно сразу после формирования списка (и перед началом отсчета времени) явным образом задать его емкость равной текущему размеру:

```
a.Capacity := a.Count;
```

Интересно отметить, что вызов метода a.TrimExcess в нашем случае не приведет к тому же результату, так как метод TrimExcess изменяет емкость только если она *существенно* отличается от текущего размера (см. описание метода TrimExcess в п. 2.2).

После добавления указанного оператора алгоритм 3 *во всех ситуациях* будет работать медленнее, чем алгоритм 4 (см. таблицу 2.2).

Таблица 2.2. *Быстродействие вариантов решения задачи Array85 (продолжение)*

Размер	Вариант 3 (если начальная емкость равна размеру)	Вариант 4
1000000	7:8	2:3
4000000	27:38	9:19
8000000	47:71	17:32

Что касается алгоритмов 6 и 7, то использование запросов для последовательностей (алгоритм 6) приводит к самому длительному алгоритму (не считая алгоритма 1 — «рекордсмена» в этом отношении). В то же время алгоритм 7, использующий срезы массива, быстрее алгоритма 6 примерно в 4 раза, хотя и не может сравниться по быстродействию с алгоритмом 2.

Завершая обсуждение задачи Array85, приведем еще одну таблицу, в которой указывается *время формирования* структур, использующихся для хранения исходного набора данных.

Напомним, что для формирования массива использовался генератор ArrRandomReal, а для формирования списков — генератор последовательности SeqRandomReal, к которому применялся соответствующий запрос: ToList или ToLinkedList.

Таблица 2.3. *Формирование наборов данных для задачи Array85*

Размер	ArrRandomReal	SeqRandomReal + ToList	SeqRandomReal + ToLinkedList
1000000	15	45	147
4000000	57	162	688
8000000	116	315	1405

Длительное время формирования двусвязного списка (в 10 раз дольше, чем для массива) объясняется его сложной структурой — цепочкой узлов, каждый из которых связан ссылочными свойствами Next и Previous со сво-

ими соседями. Менее понятно трехкратное замедление при формировании списка List, если учесть, что по внутренней структуре список List аналогичен массиву.

В данном случае замедление объясняется особенностями реализации запроса ToList. При генерации списка на основе последовательности элементы последовательности добавляются в список поочередно, причем размер последовательности заранее не известен (для определения размера последовательности можно было бы использовать запрос Count, но выполнение этого запроса потребовало бы повторного перебора всех элементов, поэтому при реализации запроса ToList такой подход не используется). В подобной ситуации начальная емкость списка устанавливается по умолчанию, а затем, при каждом превышении емкости, выполняется ее удвоение (приводящее, в свою очередь, к выделению дополнительной памяти и копированию уже входящих в список элементов во вновь выделенную память). Именно эти дополнительные действия и приводят к более медленному формированию списка по сравнению с массивом. Этими же причинами объясняется и тот факт, что емкость полученного списка равна степени двойки (а не совпадает с размером исходной последовательности).

Формирование списка можно ускорить, если создавать его не на основе последовательности, а на основе массива. Для этого достаточно для генерации набора данных использовать функцию ArrRandomReal (как и при генерации массива), к которой затем применить запрос ToList или ToLinkedList. Так как метод ToList может определить, что он применяется не к обычной последовательности, а к массиву, он в состоянии сразу узнать размер создаваемого списка, и поэтому сразу установить нужную емкость (в точности равную этому размеру). Поскольку теперь ситуации с превышением емкости никогда не возникает, время формирования списка уменьшается и приближается к времени формирования массива (см. таблицу 2.4). Следует, однако, подчеркнуть, что в данном случае в памяти размещается и массив (формируемый генератором ArrRandomReal), и список (формируемый на основе массива запросом ToList). Таким образом, выигрывая во времени выполнения, мы проигрываем в памяти. На скорости формирования двусвязного списка LinkedList замена SeqRandomReal на ArrRandomReal не сказывается (хотя тоже приводит к увеличению используемой памяти).

Таблица 2.4. *Формирование наборов данных для задачи Array85 (продолжение)*

Размер	ArrRandomReal + ToList	ArrRandomReal + ToLinkedList
1000000	21	149
4000000	79	682
8000000	156	1417

Теперь рассмотрим задачу **Array92**, требующую преобразования исходного набора данных, которое сопровождается уменьшением его размера. В задаче дан целочисленный массив, требуется удалить из него все элементы с нечетными значениями и вывести размер преобразованного массива и его элементы.

Начнем с самого неэффективного, но простого по своей идее решения, использующего возможности динамического массива:

```
Task('Array92'); // Вариант 1
var a := ReadArrInteger;
var k := a.Length;
for var i := a.Length - 1 downto 0 do
  if Odd(a[i]) then
  begin
    for var j := i + 1 to k - 1 do
      a[j - 1] := a[j];
    k -= 1;
  end;
SetLength(a, k);
a.WriteAll;
```

Исходный массив просматривается с конца, и при обнаружении нечетного числа выполняется левый сдвиг всех последующих элементов (в результате которого нечетное число из массива удаляется). Обратите внимание на то, что при выполнении левого сдвига (в отличие от правого) индексы сдвигаемых элементов должны перебираться в порядке их *возрастания*.

В переменной k хранится текущая *заполненность* массива (т. е. количество оставшихся в нем элементов); эта заполненность уменьшается на 1 при удалении каждого нечетного числа. Заполненность используется при выполнении левого сдвига, а также в конце алгоритма для того, чтобы откорректировать размер полученного массива путем вызова процедуры SetLength [1, п. 5.2]. Для вывода размера массива и его элементов используется метод WriteAll, реализованный в задачнике Programming Taskbook.

Второй вариант решения основан на той же идее, но использует дополнительные возможности класса List<T>:

```
Task('Array92'); // Вариант 2
var a := ReadSeqInteger.ToList;
for var i := a.Count - 1 downto 0 do
  if Odd(a[i]) then
    a.RemoveAt(i);
a.WriteAll;
```

Решение получилось гораздо более наглядным, однако в нем по-прежнему происходит многократный сдвиг элементов списка (просто это происходит «за кулисами», в методе RemoveAt).

Чтобы удаление какого-либо элемента не влияло на размещение других элементов набора, следует использовать двусвязный список. Правда, код алгоритма с двусвязным списком оказывается более сложным:

```
Task('Array92'); // Вариант 3
var a := ReadSeqInteger.ToLinkedList;
var cur := a.First;
while cur <> nil do
begin
  var x := cur;
  cur := cur.Next;
  if Odd(x.Value) then
    a.Remove(x);
end;
a.WriteAll;
```

Здесь допустимо организовать перебор элементов от начала к концу списка. Ссылка на текущий узел хранится в переменной cur; цикл выполняется, пока эта ссылка не равна значению nil. На каждой итерации ссылка cur перемещается к следующему элементу списка, но предварительно ее значение сохраняется во вспомогательной переменной x. Если x.Value оказывается нечетным числом, то узел x удаляется из списка.

Можно ожидать, что данный алгоритм будет выполняться гораздо быстрее, чем предыдущие, особенно для наборов данных большого размера. Однако не следует забывать о том, что для хранения двусвязного списка требуется больше памяти (а также что время его формирования существенно больше времени формирования массивов и списков типа List).

Задача допускает короткое решение, использующее запросы. Действительно, в нашем случае достаточно выполнить фильтрацию исходного массива запросом Where, оставив только четные числа:

```
Task('Array92'); // Вариант 4
var a := ReadArrInteger;
a := a.Where(e -> e mod 2 = 0).ToArray;
a.WriteAll;
```

Это самое короткое из решений, причем можно ожидать, что и скорость алгоритма будет одной из самых высоких. Однако надо учитывать, что в этом алгоритме используется дополнительная память, в которой размещается массив, полученный путем применения запроса ToArray к отфильтрованной последовательности. Впрочем, из-за применения процеду-

ры SetLength дополнительная память (причем того же размера) выделяется и в алгоритме 1.

Интересно отметить, что данная задача допускает очень эффективное решение, в котором используются лишь простейшие средства массивов (как в варианте 1). Идея этого решения состоит в том, чтобы не выполнять многократный левый сдвиг правой части массива при удалении каждого элемента, а просматривать массив с его начала, ведя подсчет нечетных чисел (в переменной k) и используя эту информацию для перемещения элементов с четными значениями *сразу* на их окончательную позицию:

```
Task('Array92'); // Вариант 5
var a := ReadArrInteger;
var k := 0;
for var i := 0 to a.Length - 1 do
  if Odd(a[i]) then
    k += 1
  else
    a[i - k] := a[i];
SetLength(a, a.Length - k);
a.WriteAll;
```

Количество нечетных чисел k используется также для того, чтобы в конце алгоритма откорректировать размер полученного массива.

Для проверки быстродействия алгоритмов их следует применить к наборам данных большого размера. При этом желательно, чтобы для разных алгоритмов обрабатывались одни и те же наборы данных (что обеспечит одинаковое количество удаляемых элементов). Можно, например, использовать наборы всех целых чисел от 1 до size, где size — размер набора. Если size является четным числом, то в таких наборах потребуется удалить *ровно половину элементов*. Для генерации исходных массивов, списков List и двусвязных списков LinkedList указанной структуры можно использовать генератор последовательности Range(1, size) [1, п. 3.5], применяя к нему подходящий экспортирующий запрос: ToArray, ToList или ToLinkedList. Как обычно, будем определять только время, требуемое для обработки уже сформированного набора данных. Результаты численного эксперимента для наборов данных трех размеров (от ста тысяч до 10 миллионов) приведены в таблице 2.5.

Таблица 2.5. *Быстродействие вариантов решения задачи Array92*

Размер	Вар. 1	Вар. 2	Вар. 3	Вар. 4	Вар. 5
100000	1827	348	3	4	1
1000000	185160	35621	25	14	4
10000000			180	121	38

Для медленных алгоритмов 1 и 2 приведены результаты только для двух начальных размеров. Алгоритмы 2 и тем более 1 существенно уступают трем последним. Более высокая скорость работы алгоритма 2 по сравнению с алгоритмом 1 объясняется более эффективной реализацией метода удаления элемента из списка (по сравнению с реализацией в виде обычного цикла, использованного в алгоритме 1).

Три последних алгоритма имеют очень высокую скорость выполнения, причем алгоритм, использующий запрос Where, выполняется быстрее алгоритма, основанного на применении двусвязного списка, а последний алгоритм оказывается самым быстрым.

Глава 3. Множества

Множеством называется структура данных, оптимизированная для выполнения таких операций, как добавление или удаление элемента, проверка принадлежности элемента множеству, проверка вложения множеств, а также стандартных теоретико-множественных операций (пересечение, объединение, разность, симметричная разность). Язык PascalABC.NET поддерживает как традиционный синтаксис Паскаля для работы с множествами, так и стандартные классы платформы .NET HashSet<T> и SortedSet<T>, предназначенные для хранения и преобразования множеств.

3.1. Описание и инициализация множеств

В традиционном Паскале тип «множество элементов типа T» описывается как set of T. При этом в качестве типа T можно использовать лишь некоторые из стандартных типов; кроме того, налагаются дополнительные ограничения на максимальный размер множества. В языке PascalABC.NET сохранен традиционный синтаксис Паскаля для работы с множествами, однако сняты ограничения на размер и тип элементов.

Множество, описанное как set of T (для некоторого типа T), не требует дополнительной инициализации и является пустым. Вместо описателя типа set of T множество можно определить с помощью инициализирующего выражения (также пришедшего из традиционного синтаксиса Паскаля), в котором элементы множества перечисляются через запятую, а их список заключается в квадратные скобки:

```
var a := [1, 2, 3, 4];
```

Тип полученного множества выводится из типа указанных элементов.

В PascalABC.NET можно также использовать обобщенные классы HashSet<T> и SortedSet<T> из стандартной библиотеки платформы .NET, предназначенные для работы с множествами. У этих классов почти одинаковый набор свойств и методов; отличаются они способом хранения элементов. Класс HashSet использует для хранения *хеш-таблицу*, что позволяет осуществлять быстрый поиск элемента в множестве. Класс SortedSet для этих же целей хранит свои элементы в отсортированном виде (что позволяет для нахождения элементов использовать быстрый алгоритм бинарного поиска).

Для инициализации множеств типа HashSet<T> или SortedSet<T> можно использовать несколько вариантов конструктора:

- без параметров,
- с параметром collection типа sequence of T,
- с параметром comparer типа IEqualityComparer<T>,
- с двумя параметрами (collection, comparer) указанных выше типов.

Параметр comparer позволяет определить способ сравнения элементов множества на равенство, если этот способ должен отличаться от стандартного (мы не будем обсуждать эту возможность, поскольку она требует привлечения понятий, выходящих за рамки настоящего пособия, — см., например, [3, п. 8.6, 9.7]). Параметр collection позволяет сразу заполнить множество элементами из указанной коллекции (т. е. последовательности или любой структуры, которая может быть неявно преобразована в последовательность). Если параметр collection не указан, то созданное множество является пустым.

Для создания множества на основе существующей последовательности a можно использовать запросы a.ToHashSet и a.ToSortedSet или короткие функции HSet(a) и SSet(a) [1, п. 4.7]. В коротких функциях можно также перечислить элементы создаваемого множества через запятую.

Для вывода множеств можно использовать те же средства, что и для вывода массивов и последовательностей [1, п. 3.2, 3.5]. При выводе множеств с помощью *процедур* Write/WriteIn и Print/PrintIn элементы множеств перечисляются через запятую, причем список элементов обрамляется *фигурными* скобками.

Важной особенностью языка PascalABC.NET является то, что традиционные множества Паскаля set of T и множества типа HashSet<T> *совместимы по присваиванию*. Множества типа SortedSet<T> такой особенностью по отношению к другим видам множеств не обладают.

3.2. Свойства множеств. Операции над множествами

Традиционное множество Паскаля set of T не имеет свойств, и для него не предусмотрены стандартные методы. Все действия над множествами типа set of T выполняются с применением операций и процедур.

Множество типа HashSet<T> имеет два свойства, доступных только для чтения: Comparer (типа IEqualityComparer<T>), содержащее объект-*компаратор*, в котором определяется способ сравнения элементов множества на равенство, и Count (типа integer), содержащее количество элементов в множестве.

Эти же свойства имеет и множество типа SortedSet<T>. Дополнительно к ним оно имеет свойства Min и Max типа T, доступные только для чтения, в которых хранятся значения минимального и максимального элемента

множества (для пустого множества эти свойства возвращают нулевые значения типа T).

Для всех видов множеств в PascalABC.NET определен одинаковый стандартный набор *операций*. При описании операций предполагается, что a и b являются множествами, совместимыми по присваиванию и имеющими элементы типа T, а выражение e имеет тип T.

e in a — возвращает True, если значение e принадлежит множеству a;

a = b — возвращает True, если множества a и b совпадают;

a <> b — возвращает True, если множества a и b не совпадают;

a < b — возвращает True, если множество a строго вложено во множество b;

a <= b — возвращает True, если множество a вложено во множество b или совпадает с ним;

a > b — возвращает True, если множество b строго вложено во множество a;

a >= b — возвращает True, если множество b вложено во множество a или совпадает с ним;

a + b — объединение множеств a и b;

a – b — разность множеств a и b;

a * b — пересечение множеств a и b;

a += b — нахождение объединения a + b и присваивание этого объединения множеству a (данная операция определена только для множеств a и b одного и того же типа);

a –= b — нахождение разности a – b и присваивание этой разности множеству a (данная операция определена только для множеств a и b одного и того же типа).

С помощью некоторых из перечисленных операций можно выполнять *добавление* и *удаление* элементов. Например, добавить элемент e типа T к множеству a типа set of T, можно следующими способами: a := a + [e] или a += [e]. Первый из приведенных вариантов можно применять и для множества a типа HashSet<T> (поскольку типы set of T и HashSet<T> совместимы по присваиванию); второй вариант применять нельзя, так как в данном случае типы множеств a и [e] являются различными. Однако для множества a типа HashSet<T> и элемента e типа T можно использовать вариант a += HSet(e). Аналогичные варианты можно использовать для множества типа SortedSet<T>. Для удаления элементов достаточно в приведенных выше примерах заменить символ «+» на символ «–».

Для множества set of T добавление и удаление элементов можно также выполнить с помощью стандартных *процедур*:

```
procedure Include(a: set of T; e: T)
procedure Exclude(a: set of T; e: T)
```

Процедура Include добавляет элемент e во множество a, процедура Exclude удаляет элемент из множества.

При работе с множествами надо учитывать, что все элементы множества являются *различными* (в частности, при добавлении к множеству элемента, который уже в нем присутствует, множество не изменяется).

3.3. Методы классов HashSet<T> и SortedSet<T>

В дополнение к операциям, рассмотренным в предыдущем пункте, классы HashSet<T> и SortedSet<T> позволяют использовать набор методов, причем почти все методы у этих классов совпадают. Вначале опишем общие методы этих классов, разбив их на несколько групп.

Добавление и удаление элементов Методы HashSet<T> и SortedSet<T>

Add(item: T): boolean
Remove(item: T): boolean
RemoveWhere(pred: T -> boolean): integer
Clear

Метод Add добавляет элемент со значением item к множеству; этот метод возвращает True, если в результате его выполнения множество изменилось; если же значение item уже содержалось в данном множестве, то метод не изменяет множество и возвращает False (напомним, что множество не может содержать элементы с одинаковыми значениями).

Метод Remove удаляет элемент со значением item из множества; смысл возвращаемого значения — такой же, как и для метода Add: возвращается True, если в результате выполнения метода множество изменилось; если же значение item отсутствовало в данном множестве, то метод не изменяет множество и возвращает False.

Метод RemoveWhere удаляет из множества все элементы, удовлетворяющие предикату pred, и возвращает количество удаленных элементов.

Метод Clear удаляет из множества все элементы; в результате свойство Count множества становится равным 0.

Логические операции Методы HashSet<T> и SortedSet<T>

Contains(item: T): boolean
SetEquals(other: sequence of T): boolean
IsProperSubsetOf(other: sequence of T): boolean
IsSubsetOf(other: sequence of T): boolean
IsProperSupersetOf(other: sequence of T): boolean
IsSupersetOf(other: sequence of T): boolean
Overlaps(other: sequence of T): boolean

Метод Contains является аналогом операции in; он возвращает True, если элемент item содержится в множестве.

Прочие методы сравнивают между собой наборы данных, причем, в отличие от аналогичных операций, второй операнд (который указывается в виде параметра other) может быть не только множеством, но и любой коллекцией. Допускается, чтобы параметр other содержал одинаковые элементы; это никак не повлияет на результат выполнения метода (можно считать, что метод выполняется для копии коллекции other, в которой отсутствуют повторяющиеся элементы). На результат выполнения метода также не влияет и порядок расположения элементов в коллекции other.

Почти все методы, выполняющие сравнение множеств, являются полными аналогами соответствующих операций. Перечислим еще раз названия этих методов, указав в скобках после каждого названия аналогичную операцию: SetEquals (=), IsProperSubsetOf (<), IsSubsetOf (<=), IsProperSupersetOf (>), IsSupersetOf (>=).

Метод Overlaps не имеет аналога среди операций. Он возвращает True, если исходное множество и параметр other имеют общие элементы (т. е. «перекрываются» — англ. overlap).

Теоретико-множественные операции Методы HashSet<T> и SortedSet<T>

UnionWith(other: sequence of T)
ExceptWith(other: sequence of T)
IntersectWith(other: sequence of T)
SymmetricExceptionWith(other: sequence of T)

Методы данной группы являются процедурами: они не возвращают результат выполнения операции, а соответствующим образом изменяют множество, для которого был вызван метод. Это множество считается первым операндом теоретико-множественной операции; второй операнд указывается в качестве параметра other; как и для методов предыдущей группы, этот параметр может быть любой коллекцией. Таким образом, имеются два отличия этих методов от аналогичных операций: во-первых, методы *изменяют* первый операнд и, во-вторых, второй операнд может быть произвольной коллекцией.

Метод UnionWith находит *объединение* множеств (аналог операции +), ExceptWith — *разность* (–), IntersectWith — *пересечение* (*). Метод SymmetricExceptionWith не имеет аналога среди операций; он находит *симметричную разность* множеств. Симметричная разность множеств a и b может быть записана с использованием операций следующим образом: (a – b) + (b – a), т. е. это объединение разностей a – b и b – a.

Как и любые ранее рассмотренные структуры данных, множества HashSet<T> и SortedSet<T> можно скопировать в массив arr типа array of T с помощью процедуры CopyTo:

Копирование в массив Метод HashSet<T> и SortedSet<T>

CopyTo(arr: array of T**[**; arrStart: integer**]**)

Необязательный параметр arrStart определяет, как обычно, позицию массива, начиная с которой в массив будут записаны элементы множества (по умолчанию arrStart равен 0). Заметим, что при копировании в массив множества типа SortedSet<T> элементы множества будут записаны в возрастающем порядке; в случае типа HashSet<T> порядок следования элементов может быть произвольным.

Нам осталось описать немногочисленные методы, которые реализованы для одного из классов HashSet<T> или SortedSet<T> и отсутствуют у другого.

В классе HashSet<T> таким методом является процедура без параметров TrimExcess, позволяющая уменьшить размер внутренней хеш-таблицы до фактического размера множества. Это действие оптимизирует память, используемую множеством, в предположении, что к нему не будут добавляться новые элементы (ср. с одноименным методом для списка List<T>, описанным в п. 2.2).

В классе SortedSet<T> таких методов два:

Методы SortedSet<T>

```
GetViewBetween(minValue, maxValue: T): SortedSet<T>
Reverse(): sequence of T
```

Метод GetViewBetween возвращает *поддиапазон* элементов исходного множества со значениями, расположенными между minValue и maxValue (включая эти значения). Параметры должны удовлетворять неравенству minValue ≤ maxValue; при его нарушении возбуждается исключение.

Метод Reverse представляет собой оптимизированный вариант одноименного запроса для последовательностей [1, п. 3.7, 4.2]; он возвращает последовательность, содержащую элементы исходного множества в *убывающем* порядке.

3.4. Использование множеств при решении задач

Множества обычно используются в программе, если требуется сформировать набор *различных* элементов и впоследствии *быстро* проверять, входит ли в данный набор некоторое значение. Кроме того, множества предоставляют средства для выполнения теоретико-множественных операций над наборами данных (один из которых даже не обязан быть множеством).

То обстоятельство, что множества всегда содержат различные элементы, может натолкнуть нас на мысль применить эти структуры данных для решения уже обсуждавшейся в первом выпуске задачи **Array47** [1, п. 5.3], в которой требовалось найти количество различных элементов в исходном массиве целых чисел. В [1, п. 5.3] мы реализовали четыре алгоритма: три использовали средства, связанные с массивами, а четвертый — запрос Dis-

tinct. Именно четвертый алгоритм оказался самым быстродействующим, оставив далеко позади все прочие варианты решения.

Возникает вопрос: смогут ли конкурировать с вариантом решения, основанным на запросе Distinct, варианты, использующие множества? Поскольку любую коллекцию можно преобразовать как во множество типа HashSet, так и во множество типа SortedSet, мы можем получить еще два варианта решения, текст которых является таким же коротким, как и текст варианта 4. В варианте 5 используется запрос ToHashSet, преобразующий исходный массив во множество типа HashSet:

```
Task('Array47'); // Вариант 5
var a := ReadArrInteger;
var k := a.ToHashSet.Count;
Write(k);
```

В варианте 6 используется запрос ToSortedSet, преобразующий исходный массив во множество типа SortedSet:

```
Task('Array47'); // Вариант 6
var a := ReadArrInteger;
var k := a.ToSortedSet.Count;
Write(k);
```

Нас интересует только размер полученных множеств, поэтому мы не связываем множества с какой-либо переменной, а сразу обращаемся к их свойству Count, сохраняя его в переменной k.

Для проверки быстродействия надо применить эти алгоритмы к большим массивам. Поскольку естественно ожидать, что алгоритмы 4–6 являются очень эффективными, для их проверки можно использовать массивы большего размера size, чем в [1, п. 5.3] при проверке алгоритмов 1–4. Как и в [1, п. 5.3], исходный массив размера size генерировался с помощью функции ArrRandom(size, 1, size div 2), а программа запускалась в режиме релиза.

Таблица 3.1. *Быстродействие вариантов решения задачи Array47*

Размер size	Вариант 4 (Distinct)	Вариант 5 (ToHashSet)	Вариант 6 (ToSortedSet)	Результат k
80000	6	3	525	34647
160000	10	7	2070	69057
320000	22	15	8204	138390

Результаты численного эксперимента приведены в таблице 3.1 (в последнем столбце выводится полученный результат — число k). Эти результаты показывают, что вариант решения, использующий множество HashSet, имеет такое же (очень высокое) быстродействие, как и вариант решения с запросом Distinct. Быстродействие варианта, основанного на множестве SortedSet, существенно ниже, и разрыв увеличивается с увеличением раз-

мера обрабатываемых данных. Впрочем, алгоритм 6 все равно выполняется в несколько раз быстрее, чем алгоритмы 1–3 для массивов, рассмотренные в [1, п. 5.3].

Запросы a.ToHashSet и a.ToSortedSet фактически приводят к вызову конструкторов соответствующих классов с параметром-коллекцией: new HashSet<integer>(a) и new SortedSet<integer>(a). Вместо запросов (или эквивалентных им конструкторов с параметром) можно создать пустое множество и заполнить его в цикле, используя метод Add (см. п. 3.3). Например, для множества SortedSet подобный вариант решения будет выглядеть следующим образом:

```
Task('Array47'); // Вариант 6а
var a := ReadArrInteger;
var s := new SortedSet;
foreach var e in a do
  s.Add(e);
var k := s.Count;
Write(k);
```

Изменится ли быстродействие наших алгоритмов при изменении способа формирования множества? Численный эксперимент (с массивами того же размера) дает интересный результат, приведенный в таблице 3.2.

Таблица 3.2. Быстродействие вариантов решения задачи Array47 (продолжение)

Размер size	Вариант 5а (HashSet, цикл foreach)	Вариант 6а (SortedSet, цикл foreach)	Результат k
80000	3	26	34684
160000	7	51	69074
320000	17	143	138390

Быстродействие нового варианта для множества HashSet практически не изменяется, тогда как быстродействие варианта 6а для множества SortedSet *многократно увеличивается* (хотя по-прежнему уступает варианту для HashSet). Это означает, что вариант конструктора класса SortedSet с параметром-коллекцией имеет в библиотеке .NET неэффективную реализацию.

Анализируя полученные результаты, можно прийти к следующим выводам: если в алгоритме не требуется хранить элементы множества в отсортированном виде (и получать различные диапазоны элементов отсортированного множества), то для хранения множества следует использовать класс HashSet, имеющий более эффективную реализацию по сравнению с классом SortedSet. Если все же желательно использовать класс SortedSet, то при его создании следует избегать неэффективного конструктора с параметром-коллекцией, применяя вместо него цикл с методом Add.

Глава 4. Словари (ассоциативные массивы)

Обобщенный класс Dictionary<TKey, TVal> (*словарь*) хранит данные в виде набора пар «ключ–значение» (key–value), причем для доступа к *значению* value пары (key, value) из словаря d можно использовать операцию *индексирования по ключу*: d[key]. Таким образом, словарь может рассматриваться как обобщение массива, для которого в качестве индексов можно использовать данные *любого типа* (требуется лишь, чтобы для этого типа была определена операция сравнения на равенство). По этой причине класс Dictionary<TKey, TVal> еще называют *ассоциативным массивом*.

4.1. Свойства словаря и особенности его операции индексирования. Тип KeyValuePair<TKey, TVal>

Словарь Dictionary<TKey, TVal> имеет пять свойств: Count, Keys, Values, Comparer, Item. Все свойства, кроме Item, доступны только для чтения.

Свойство Count типа integer позволяет определить текущий размер словаря, свойства Keys и Values возвращают коллекции, содержащие только ключи или только значения словаря. У коллекций Keys и Values также имеется свойство Count, значение которого совпадает со значением свойства Count словаря.

Свойство Comparer представляет собой объект-*компаратор* типа IEqualityComparer<TKey>, определяющий способ сравнения ключей на равенство (обычно в качестве такого объекта используется стандартный компаратор для типа TKey).

Свойство Item является свойством-*индексатором* и позволяет получать значения элементов словаря по их ключам. Данное свойство (как и одноименное свойство списка List<T>) практически не используется, так как вместо выражения d.Item[key] для доступа к значению элемента с ключом key из словаря d можно использовать выражение d[key] — такое же по форме, как и для индексного доступа к элементам массива или списка List<T>.

Операция индексирования d[key] для словаря d имеет две важные особенности. Во-первых, при доступе к выражению d[key] *на чтение* необходимо, чтобы в словаре существовала пара с ключом key (если такая пара отсутствует, то будет возбуждено исключение). Во-вторых, при доступе к

выражению d[key] *на запись* (при указании этого выражения в левой части оператора присваивания: d[key] := newValue) выполняемые действия зависят от того, существует ли уже в словаре пара с ключом key. Если такая пара существует, то в ней изменяется второй элемент (т. е. пара принимает вид (key, newValue)). Если пары с ключом key в словаре еще нет, то она *автоматически создается* в виде (key, newValue).

В словаре не может быть пар с одинаковыми ключами, однако допускается, чтобы некоторые пары имели одинаковые значения. Это, в частности, означает, что в коллекции Keys все элементы (*ключи*) различны, а в коллекции Values некоторые элементы (*значения*) могут совпадать.

Пары в словаре представляются в виде специального обобщенного типа KeyValuePair<TKey, TVal> со свойствами Key и Value, доступными только для чтения. В методах словаря этот тип не используется, однако он возникает в трех конструкциях, связанных со словарем (все эти конструкции обсуждаются в следующем пункте):

- при *инициализации* словаря функцией Dict со списком пар;
- при *переборе* пар словаря в цикле foreach;
- при *проверке* (с помощью операции in), содержится ли требуемая пара в словаре.

Для создания новой пары (key, value) можно использовать конструктор типа KeyValuePair<TKey, TVal> с параметрами (key, value), например:

```
var p := new KeyValuePair<string, integer>('abc', 0);
```

В этом примере созданная пара имеет строковый ключ и целочисленное значение.

Однако в PascalABC.NET предусмотрен более удобный вариант создания пары: с помощью *короткой функции* KV с параметрами (key, value), например:

```
var p := KV('abc', 0);
```

Важным преимуществом короткой функции KV является то, что типы для ключа и значения *выводятся* из ее параметров (в то время как при вызове конструктора типы надо обязательно указывать).

4.2. Создание словаря и перебор его элементов. Операция in

Для инициализации словаря типа Dictionary<TKey, TVal> предусмотрены шесть вариантов конструктора:

- без параметров,
- с параметром capacity типа integer,
- с параметром comparer типа IEqualityComparer<TKey>,
- с параметрами capacity и comparer,

- с параметром dict типа Dictionary<TKey, TVal>,
- с параметрами dict и comparer.

Параметр capacity позволяет задать начальную *емкость* словаря, т. е. размер внутренних структур, используемых словарем для хранения данных. Его можно использовать, если заранее примерно известен будущий размер словаря. Если этот параметр не указан, то используется емкость по умолчанию. Заметим, что, в отличие от списка List<T>, словарь не имеет средств для явного управления своей емкостью (или даже для получения ее текущего значения).

Параметр comparer позволяет определить способ сравнения ключей словаря на равенство, если этот способ должен отличаться от стандартного (как и для множеств, мы не будем обсуждать эту возможность).

Параметр dict позволяет сразу заполнить словарь элементами из другого словаря. Если параметр dict не указан, то созданный словарь является пустым. В отличие от аналогичных конструкторов ранее рассмотренных структур данных (списков и множеств), произвольную *последовательность* в качестве параметра dict указывать нельзя (даже если эта последовательность состоит из элементов типа KeyValuePair).

Для создания словаря на базе последовательности (или любой коллекции) следует использовать запрос ToDictionary [1, п. 4.6]. Напомним, что в этом запросе требуется указать *селектор ключа* — лямбда-выражение, позволяющее по элементу исходной последовательности определить его ключ. Для всех элементов исходной последовательности должны возвращаться *различные* ключи, в противном случае запрос ToDictionary возбудит исключение. В запросе можно также указать необязательный *селектор значения* — лямбда-выражение, определяющее по элементу исходной последовательности то значение, которое будет связано с ключом (если селектор значения не указан, то с ключом связывается сам элемент исходной последовательности).

Для быстрого создания словаря, содержащего конкретные пары «ключ–значение», удобно использовать *короткую функцию* Dict с параметрами — конкретными парами (для представления этих пар, в свою очередь, удобно использовать короткую функцию KV — см. п. 4.1). В следующем примере один и тот же вариант словаря создается двумя разными способами: с применением запроса ToDictionary и короткой функции Dict:

```
var d1 := Seq(43, 23, 52, 61, 32).ToDictionary(e -> e div 10,
  e -> e mod 10);
var d2 := Dict(KV(4, 3), KV(2, 3), KV(5, 2), KV(6, 1), KV(3, 2));
Println(d1); // {(4,3),(2,3),(5,2),(6,1),(3,2)}
Println(d2); // {(4,3),(2,3),(5,2),(6,1),(3,2)}
```

В данном примере по каждому элементу исходной последовательности (двузначному числу) строится пара цифр, первый элемент которой (ключ) равен левой цифре числа, т. е. разряду десятков, а второй (значение) — правой цифре, т. е. разряду единиц.

Обратите внимание на то, что при выводе словаря в процедурах Write/WriteIn и Print/PrintIn пары «ключ–значение» заключаются в круглые скобки (как при выводе кортежей), а список пар заключается в фигурные скобки (как при выводе множеств).

В полученных словарях d1 и d2 имеются совпадающие значения, но все ключи различны. Если бы в исходной последовательности или в списке параметров функции Dict были указаны числа с одинаковыми значениями разряда десятков (например, 81 и 82), то произошло бы аварийное завершение программы с сообщением «*Элемент с тем же ключом уже был добавлен*».

Для перебора элементов словаря можно использовать несколько вариантов цикла foreach. Самый быстрый вариант основан на переборе пар «ключ–значение», т. е. данных типа KeyValuePair. Например, напечатать элементы исходной последовательности, из которых были получены элементы словаря d1, можно следующим образом:

```
foreach var e in d1 do
  Print(e.Key * 10 + e.Value); // 43 23 52 61 32
```

Можно также организовать перебор всех ключей словаря, используя свойство-коллекцию Keys:

```
foreach var e in d1.Keys do
  Print(e * 10 + d1[e]); // 43 23 52 61 32
```

Однако данный вариант менее эффективен по сравнению с предыдущим, так как в цикле на каждой итерации при обработке выражения d1[e] приходится осуществлять дополнительный поиск значения, соответствующего ключу e. Следует заметить, что, хотя хранение данных в словаре оптимизировано для быстрого поиска по ключу, все же индексный доступ в словаре требует больше времени, чем аналогичный доступ в массиве или списке List<T> (в которых по индексу может быть сразу вычислен адрес требуемого элемента).

Если требуется перебрать только вторые элементы пар, входящих в словарь, то удобно использовать цикл foreach по свойству-коллекции Values:

```
foreach var e in d1.Values do
  Print(e); // 3 3 2 1 2
```

Для того чтобы проверить, входит ли пара p (типа KeyValuePair) в словарь d, предусмотрена операция p in d, возвращающая логическое значение.

При использовании данной операции может оказаться полезной запись пары в виде KV(key, value), например:

```
Writeln(KV(4, 3) in d1); // True
Writeln(KV(3, 4) in d1); // False
```

Операцию in можно применять и к свойствам-коллекциям словаря Keys и Values; кроме того, для поиска *ключа* можно использовать вариант операции in для самого словаря, например:

```
Writeln(4 in d1.Keys); // True
Writeln(7 in d1); // False
```

Словарь Dictionary<TKey, TVal> и его свойства-коллекции Keys и Values могут быть неявно преобразованы в последовательности соответствующих типов (sequence of KeyValuePair<TKey, TVal>, sequence of TKey, sequence of TVal), поэтому для них доступны все запросы последовательностей, описанные в [1, гл. 4].

4.3. Методы класса Dictionary<TKey, TVal>

Класс Dictionary<TKey, TVal> имеет небольшое количество методов, которые можно разбить на три группы, связанные с добавлением и удалением, поиском и доступом к данным по ключу.

Добавление и удаление Методы Dictionary<TKey, TVal>

Add(key: TKey; value: TValue)
Remove(key: TKey): boolean
Clear

Метод Add добавляет в словарь пару (key, value). Если словарь уже содержит пару с ключом key, то метод возбуждает исключение. Напомним, что для добавления указанной пары в словарь d можно использовать более безопасный вариант: d[key] := value. Если при выполнении этого варианта окажется, что словарь уже содержит ключ key, то просто произойдет замена значения, связанного с этим ключом.

Метод Remove удаляет из словаря пару с ключом key. Этот метод возвращает True, если в результате его выполнения словарь изменился; если же в словаре отсутствует пара с ключом key, то метод не изменяет словарь и возвращает False.

Метод Clear удаляет из словаря все его элементы; при этом свойство Count словаря становится равным 0.

Поиск Методы Dictionary<TKey, TVal>

ContainsKey(key: TKey): boolean
ContainsValue(value: TVal): boolean

Методы ContainsKey и ContainsValue ищут в словаре указанный ключ key или, соответственно, значение value и при его нахождении возвращают

True. Напомним, что для поиска в словаре d *пары* (key, value) можно использовать операцию in, например: KV(key, value) in d.

Доступ к элементам по ключу Методы Dictionary<TKey, TVal>

* **Get**(key: TKey): TValue
TryGetValue(key: TKey; var value: TValue): boolean

Данные методы позволяют получить значение value для пары (key, value). Функция Get возвращает требуемое значение, а метод TryGetValue записывает его в выходной параметр value. Напомним, что получить значение value для ключа key, содержащегося в словаре d, можно также с помощью операции индексирования d[key]. Однако операция индексирования возбуждает исключение при отсутствии в словаре ключа key, тогда как методы Get и TryGetValue в этом отношении являются «безопасными»: если ключ key отсутствует, то метод Get возвращает нулевое значение типа TValue, а метод TryGetValue возвращает False и записывает нулевое значение в параметр value.

Завершая обзор методов класса Dictionary, отметим, что у этого класса отсутствует метод CopyTo, позволяющий записать его данные в массив. Однако метод CopyTo имеется у свойств-коллекций Keys и Values:

Копирование в массив Метод коллекций Keys и Values

CopyTo(arr: array of T; arrStart: integer)

Содержимое указанной коллекции будет записано в массив arr (типа array of TKey, если CopyTo вызывается для коллекции Keys словаря Dictionary<TKey, TVal>, и типа array of TVal, если CopyTo вызывается для коллекции Values этого же словаря). Обратите внимание на то, что в реализации метода CopyTo для коллекций Keys и Values (как и в рассмотренных ранее реализациях для массива и двусвязного списка — см. [1, п. 5.1] и п. 2.6) параметр arrStart является *обязательным*.

4.4. Использование словарей при решении задач

Некоторые варианты задач, связанных с обработкой одинаковых элементов исходного набора, допускают наглядные и эффективные решения, основанные на применении словарей. Примером такой задачи является задача **Array48**, в которой требуется найти максимальное количество одинаковых элементов в исходном целочисленном массиве.

В данном случае нам не удастся использовать множество, так как при формировании множества по исходному массиву теряется информация о том, сколько раз каждый из элементов множества входил в массив. Однако мы можем сформировать словарь типа Dictionary<integer, integer>, ключами которого будут значения элементов исходного массива, а значениями — количество их вхождений в массив:

```
Task('Array48'); // Вариант 1
var a := ReadArrInteger;
var d := new Dictionary<integer, integer>;
foreach var e in a do
  if d.ContainsKey(e) then
    d[e] += 1
  else
    d[e] := 1;
var k := d.Values.Max;
Write(k);
```

Если на очередной итерации словарь d уже содержит пару с ключом e, то значение (т. е. вторая компонента этой пары) увеличивается на 1; если число e появилось в первый раз, то выполняется оператор присваивания d[e] := 1, обеспечивающий добавление в словарь пары с ключом e и значением 1.

После выхода из цикла достаточно получить коллекцию всех значений словаря (d.Values) и найти в этой коллекции максимальный элемент.

Если воспользоваться методом d.Get(e), который возвращает d[e] или 0, если словарь d не содержит ключ e, то в теле цикла foreach можно обойтись без условного оператора:

```
Task('Array48'); // Вариант 1а
var a := ReadArrInteger;
var d := new Dictionary<integer, integer>;
foreach var e in a do
  d[e] := d.Get(e) + 1;
var k := d.Values.Max;
Write(k);
```

Впрочем, при решении этой задачи можно вообще обойтись без словаря, если использовать *группирующий запрос* GroupBy [1, п. 4.5], позволяющий разбить исходную коллекцию на группы с одинаковым ключом.

В нашем случае ключом (как и в варианте 1) будет само значение элемента:

```
Task('Array48'); // Вариант 2
var a := ReadArrInteger;
var k := a.GroupBy(e -> e).Max(e -> e.Count);
Write(k);
```

Напомним, что вариант запроса GroupBy с одним параметром — лямбда-выражением, определяющим ключ, возвращает последовательность элементов типа IGrouping, представляющих собой группы элементов исходной коллекции с одинаковым ключом. Элемент типа IGrouping сам является последовательностью, поэтому к нему можно применить запрос Count для

определения его размера. Максимальный из размеров полученных групп и будет требуемым результатом.

Можно избавиться от параметра в запросе Max, если использовать вариант запроса GroupBy с двумя параметрами:

```
Task('Array48'); // Вариант 2a
var a := ReadArrInteger;
var k := a.GroupBy(e -> e, (k, ee) -> ee.Count).Max;
Write(k);
```

Второй параметр запроса GroupBy, также являющийся лямбда-выражением, позволяет явно определить элементы возвращаемой последовательности по двум своим параметрам: ключу k и связанной с ним группе элементов (последовательности) ee. Поскольку нам достаточно знать размеры всех групп, именно их мы и возвращаем в качестве результата группировки, после чего применяем к ним запрос Max (уже без параметров).

Наконец, в качестве еще одного варианта можно рассмотреть комбинированное решение, в котором используются и группирующий запрос, и словарь:

```
Task('Array48'); // Вариант 3
var a := ReadArrInteger;
var d := a.GroupBy(e -> e).ToDictionary(e -> e.Key, e -> e.Count);
var k := d.Values.Max;
Write(k);
```

В данном варианте сгруппированная последовательность превращается в словарь с помощью запроса ToDictionary (см. [1, п. 4.6] и п. 4.1), после чего, как и в варианте 1, находится максимум среди всех *значений* полученного словаря.

Для анализа быстродействия следует применить полученные алгоритмы к массивам большого размера. При этом интересно проанализировать две ситуации: когда количество различных чисел в массиве велико (и поэтому полученный словарь или сгруппированная последовательность будут содержать большое число элементов) и когда массив большого размера содержит небольшое количество различных чисел.

Начнем с ситуации, когда массив содержит большое количество различных чисел. Для этого будем генерировать массив с помощью функции ArrRandom(size, 1, size div 2) для больших значений size. Результаты тестирования алгоритмов приведены в таблице 4.1; последний столбец содержит ответ — число k, полученное алгоритмом.

Таблица 4.1. *Быстродействие вариантов решения задачи Array48 (случай с большим количеством различных чисел в исходном наборе)*

Размер size	Вар. 1	Вар. 1а	Вар. 2	Вар. 2а	Вар. 3	Результат k
800000	128	129	453	894	989	13

1600000	302	305	1189	1876	1809	11
3200000	625	623	3149	4470	4371	12

Мы видим, что более эффективными в отношении быстродействия являются алгоритмы 1 и 1а, не применяющие группирующие запросы, а среди тех алгоритмов, которые эти запросы применяют, быстрее работает вариант 2, не преобразующий полученные группы элементов и не формирующий на их основе словарь.

Теперь обратимся к ситуации, когда в большом массиве имеется мало различных чисел; это означает, что теперь (в отличие от предыдущей ситуации) у нас будет мало групп, но группы будут иметь большой размер. Для генерации подобных массивов достаточно изменить последний параметр функции ArrRandom, например: ArrRandom(size, 1, 100). В такой ситуации алгоритмы будут выполняться быстрее, поэтому увеличим размер тестируемых массивов в 10 раз.

Таблица 4.2. *Быстродействие вариантов решения задачи Array48 (случай с малым количеством различных чисел в исходном наборе)*

Размер size	Вар. 1	Вар. 1а	Вар. 2	Вар. 2а	Вар. 3	Результат k
8000000	327	255	337	372	326	80691
16000000	654	509	675	736	720	161083
32000000	1335	1017	1328	1506	1552	321273

В данном случае более быстрым оказывается алгоритм 1а, однако разница в быстродействии между всеми вариантами невелика.

Таким образом, можно сделать вывод, что если заранее неизвестен размер создаваемого словаря, то (для больших наборов данных) предпочтительнее создавать словарь, используя цикл. С другой стороны, группирующие запросы также выполняются достаточно быстро и, кроме того, приводят к весьма наглядному коду.

В заключение рассмотрим задачу **Array100**, при решении которой можно использовать большинство из ранее изученных структур. В ней требуется удалить из исходного целочисленного массива элементы, встречающиеся ровно два раза, после чего вывести размер полученного массива и все его элементы.

Для удаления элементов можно использовать любой из подходов, ранее применявшихся при решении задачи Array92 (см. п. 2.7). Однако предварительно необходимо сформировать вспомогательную структуру, позволяющую для каждого элемента исходного набора быстро определить, следует ли его удалять. В качестве такой структуры естественно использовать словарь.

В первом варианте решения будем применять список на базе массива:

```
Task('Array100'); // Вариант 1
var a := ReadSeqInteger.ToList;
```

```
var d := new Dictionary<integer, integer>;
foreach var e in a do
  d[e] := d.Get(e) + 1;
for var i := a.Count - 1 downto 0 do
  if d[a[i]] = 2 then
    a.RemoveAt(i);
a.WriteAll;
```

Во втором варианте будем совместно использовать словарь и двусвязный список:

```
Task('Array100'); // Вариант 2
var a := ReadSeqInteger.ToLinkedList;
var d := new Dictionary<integer, integer>;
foreach var e in a do
  d[e] := d.Get(e) + 1;
var cur := a.First;
while cur <> nil do
begin
  var x := cur;
  cur := cur.Next;
  if d[x.Value] = 2 then
    a.Remove(x);
end;
a.WriteAll;
```

Начальные фрагменты вариантов 1 и 2, связанные с формированием словаря d, полностью совпадают. Обратите внимание на то, что цикл foreach можно использовать для перебора и списков List на базе массива, и двусвязных списков LinkedList.

В третьем варианте решения будем активно применять запросы: как для формирования словаря, так и для отбора требуемых элементов из исходного массива. Как обычно, этот вариант окажется самым коротким:

```
Task('Array100'); // Вариант 3
var a := ReadArrInteger;
var d := a.GroupBy(e -> e).ToDictionary(e -> e.Key, e -> e.Count);
a := a.Where(e -> d[e] <> 2).ToArray;
a.WriteAll;
```

Основываясь на результатах численных экспериментов, проведенных для решений задач Array92 и Array48, можно ожидать, что дольше всего будет выполняться вариант 1 (из-за многократных сдвигов оставшейся части списка при удалении элементов), тогда как варианты 2 и 3 будут не слишком сильно отличаться по быстродействию. Эти предположения оказываются справедливыми (см. таблицу 4.3). Массивы для тестирования

формировались с помощью функции ArrRandom(size, 1, size div 2); в последнем столбце выводится размер преобразованных массивов.

Таблица 4.3. *Быстродействие вариантов решения задачи Array100*

Размер size	Вариант 1	Вариант 2	Вариант 3	Новый размер
80000	188	17	45	58372
160000	738	37	97	116788
320000	2993	92	229	233806

Глава 5. Стеки и очереди

Помимо списков, множеств и словарей в стандартной библиотеке платформы .NET имеются еще два обобщенных класса, предназначенных для хранения данных: Stack<T> (*стек*) и Queue<T> (*очередь*) . Эти классы являются более простыми по сравнению с ранее рассмотренными классами и не так широко используются, однако в некоторых ситуациях их применение приводит к наиболее эффективным алгоритмам. Данные классы имеют много общего, поэтому их удобно рассмотреть совместно.

5.1. Особенности стеков и очередей, их инициализация и свойство Count

Стеки и очереди можно рассматривать как цепочки элементов, для которых добавление, удаление и чтение возможно только на их концах, причем у стека для всех операций доступен только один конец (*вершина*, top), а у очереди добавление возможно в один конец цепочки (называемый *хвостом* очереди, tail), а чтение и удаление — с другого конца (*головы*, head).

Это означает, что элемент, добавленный в стек последним, будет удален из нее первым (данное правило сокращенно обозначается LIFO — «Last In — First Out», т. е. «последним пришел — первым ушел»). В случае очереди элемент, первым добавленный в нее, первым и будет извлечен (правило FIFO — «First In — First Out», т. е. «первым пришел — первым ушел») .

Для инициализации стека и очереди необходимо использовать конструкторы new Stack<T> и new Queue<T>, реализованные в трех вариантах (как и для списка List<T>):

- без параметров,
- с параметром capacity типа integer,
- с параметром collection типа sequence of T.

Параметр capacity позволяет задать начальную *емкость* стека или очереди; если этот параметр не указан, то используется емкость по умолчанию. Подобно классу Dictionary (и в отличие от класса List), стек и очередь не имеют средств для явного управления своей емкостью или определения ее текущего значения, однако у них имеется метод TrimExcess без параметров, позволяющий уменьшить емкость, приведя ее в соответствие с теку-

щим размером стека или очереди (напомним, что такой же метод имеется в классах List и HashSet).

С помощью конструктора с параметром collection стек и очередь можно сразу заполнить данными из любой коллекции. При этом данные пересылаются в эти структуры по одному элементу, поэтому вершиной созданного стека будет *последний* элемент исходной коллекции, а головой созданной очереди — *первый* элемент коллекции.

У стека и очереди имеется единственное свойство Count типа integer, доступное только для чтения и позволяющее узнать текущий размер этих структур.

5.2. Добавление, извлечение и чтение элементов. Операция in и дополнительные методы

И стек, и очередь имеют по одному методу для добавления, извлечения и чтения элемента, которые удобно описать совместно.

Добавление в стек Метод Stack<T>

Push(item: T)

Добавление в очередь Метод Queue<T>

Enqueue(item: T)

Метод Push добавляет новый элемент со значением item в вершину стека, а метод Enqueue — в хвост очереди. Методы являются процедурами.

Извлечение из стека Метод Stack<T>

Pop: T

Извлечение из очереди Метод Queue<T>

Dequeue: T

Функция Pop удаляет элемент из вершины стека и возвращает значение удаленного элемента. Функция Dequeue удаляет элемент из головы очереди и возвращает его значение. При попытке вызова этих функций для пустого стека или очереди возбуждается исключение.

Чтение из стека и очереди Метод Stack<T> и Queue<T>

Peek: T

Метод для чтения имеет у стека и очереди одинаковое имя. Он возвращает значение верхнего элемента стека или головного элемента очереди, не удаляя их из данной структуры.

Хотя доступ к элементам в середине очереди или стека невозможен, предусмотрена логическая операция in, позволяющая проверить, содержится ли в стеке или очереди элемент с указанным значением. Вместо этой операции можно использовать метод Contains(item: T), также возвращающий логическое значение.

Для очистки стека и очереди предусмотрен метод Clear — процедура без параметров:

Удаление всех элементов Метод Stack<T> и Queue<T>

Clear

В результате выполнения этого метода свойство Count становится равным 0.

Наконец, у стека и очереди имеется метод, позволяющий записать их содержимое в массив arr, начиная с позиции массива arrStart (параметр arrStart является обязательным):

Копирование в массив Метод Stack<T> и Queue<T>

CopyTo(arr: array of T; arrStart: integer)

При этом первым в массив записывается элемент из вершины стека или из головы очереди.

Подобно всем структурам, рассмотренным ранее, стек и очередь могут быть неявно преобразованы в последовательности, и поэтому для них доступны все запросы последовательностей, описанные в [1, гл. 4].

5.3. Использование очередей при решении задач

В качестве примера использования класса Queue<T> рассмотрим решение задачи **Array63**, посвященной алгоритму *слияния упорядоченных наборов данных*. В этой задаче даны два вещественных массива a и b размера 5, элементы которых упорядочены по возрастанию. Требуется сформировать новый массив c (размера 10), который содержал бы все элементы исходных массивов и при этом также был упорядочен по возрастанию.

Как обычно, при решении необязательно использовать массивы; можно выбрать другую структуру данных, для которой полученное решение будет иметь какие-либо преимущества (например, краткость и наглядность кода, эффективное использование памяти или высокое быстродействие).

Начнем с традиционного решения, использующего массивы. Основная идея алгоритма состоит в том, чтобы в каждый момент времени обрабатывать пару *текущих элементов* массивов a и b: тот из текущих элементов, который имеет меньшее значение, записывается в массив c, и при этом текущим делается элемент, следующий за только что записанным. Необходимо также предусмотреть особую обработку ситуации, когда алгоритм дойдет до конца одного из массивов. Понятно, что в этом случае будет достаточно добавлять в массив c все оставшиеся элементы другого исходного массива.

Для хранения информации об *индексе* текущего элемента массивов a и b будем использовать вспомогательные переменные ka и kb, а для хранения *значения* текущего элемента — переменные va и vb. В начале алгоритма переменные ka и kb равны 0, а va и vb равны соответственно a[0] и b[0].

После увеличения переменной ka необходимо проверять, существует ли элемент массива a с индексом ka; если существует, то он записывается в переменную va, если такого элемента нет (в нашем случае это произойдет, если ka получит значение 5), то в va достаточно записать *очень большое значение*. Это гарантирует, что на последующих итерациях в массив c будут записаны оставшиеся элементы из массива b. Обработка переменной kb выполняется аналогичным образом. Получаем следующий вариант решения:

```
Task('Array63'); // Вариант 1
var a := ReadArrReal(5);
var b := ReadArrReal(5);
var c := new real[10];
var ka := 0;
var kb := 0;
var va := a[0];
var vb := b[0];
for var i := 0 to 9 do
  if va < vb then
  begin
    c[i] := va;
    ka += 1;
    if ka < 5 then
      va := a[ka]
    else
      va := real.MaxValue;
  end
  else
  begin
    c[i] := vb;
    kb += 1;
    if kb < 5 then
      vb := b[kb]
    else
      vb := real.MaxValue;
  end;
c.Write;
```

Для ввода исходных массивов мы использовали функцию ReadArrReal с параметром 5, так как в этой задаче (в отличие от большинства других задач группы Array) размер исходных массивов фиксирован (равен 5), и поэтому для *размера* не требуется предусматривать операцию ввода (напомним, что функция ReadArrReal без параметров вначале вводит размер массива, а затем — его элементы). Для задания большого вещественного зна-

чения мы использовали свойство MaxValue типа real, возвращающее наибольшее значение этого типа.

Мы получили эффективный алгоритм, решающий задачу за один просмотр исходных массивов. Правда, код алгоритма является достаточно большим из-за необходимости корректировки переменных ka и kb и отслеживания особых ситуаций, связанных с этими переменными.

Реализуем этот же алгоритм, используя вместо массивов очереди:

```
Task('Array63'); // Вариант 2
var a := new Queue<real>(ReadSeqReal(5));
var b := new Queue<real>(ReadSeqReal(5));
var c := new Queue<real>(10);
a.Enqueue(real.MaxValue);
b.Enqueue(real.MaxValue);
for var i := 0 to 9 do
  if a.Peek < b.Peek then
    c.Enqueue(a.Dequeue)
  else
    c.Enqueue(b.Dequeue);
c.Write;
```

Для заполнения исходных очередей мы использовали конструктор класса Queue<real> с параметром — последовательностью, возвращаемой функцией ReadSeqReal(5). При создании очереди c для хранения результирующего набора, мы явно указали ее емкость 10, поскольку нам известен итоговый размер полученной очереди.

Затем мы добавили в конец исходных очередей большие значения. Это позволило избежать в последующем цикле дополнительных проверок, связанных с возможностью выхода за границы одной из очередей.

В цикле нам теперь не надо отслеживать текущие элементы: на каждой итерации анализируются *головные элементы* очередей a и b. Элемент, добавляемый в очередь c, извлекается из очереди, в которой он содержался, и, таким образом, головным элементом этой очереди становится следующий за ним. Для доступа к головным элементам на чтение мы используем метод Peek, а для их извлечения и добавления в очередь c — методы Dequeue и Enqueue, причем извлечение и добавление реализуются в виде одного оператора. Вывод полученной очереди выполняется с помощью метода Write, который можно применять к любой коллекции.

Идею с добавлением значения real.MaxValue в конец исходных наборов данных можно реализовать и для массивов, однако короткая запись для такого добавления требует дополнительных расходов памяти:

```
Task('Array63'); // Вариант 1a
var a := ReadArrReal(5) + Arr(real.MaxValue);
```

```
var b := ReadArrReal(5) + Arr(real.MaxValue);
var c := new real[10];
var ka := 0;
var kb := 0;
for var i := 0 to 9 do
  if a[ka] < b[kb] then
  begin
    c[i] := a[ka];
    ka += 1;
  end
  else
  begin
    c[i] := b[kb];
    kb += 1;
  end;
c.Write;
```

Здесь кроме массивов, создаваемых функциями ReadArrReal(5), создаются новые массивы, имеющие на один элемент больше, и именно эти массивы связываются с переменными a и b. Теперь отпадает необходимость в переменных va и vb и в дополнительных проверках значений ka и kb в цикле.

Можно избежать дополнительных расходов памяти, если реализовать ввод данных в массивы a и b «вручную»:

```
Task('Array63'); // Вариант 1b
var a := new real[6];
for var i := 0 to 4 do
  a[i] := ReadReal;
a[5] := real.MaxValue;
var b := new real[6];
for var i := 0 to 4 do
  b[i] := ReadReal;
b[5] := real.MaxValue;
var c := new real[10];
// Завершающая часть алгоритма совпадает с вариантом 1a
```

Однако во всех модификациях варианта 1 по-прежнему надо использовать переменные ka и kb, тогда как в варианте 2 в них нет необходимости.

В заключение приведем еще один вариант решения, менее эффективный, но чрезвычайно короткий:

```
Task('Array63'); // Вариант 3
var a := ReadArrReal(5);
var b := ReadArrReal(5);
```

```
var c := a + b;
Sort(c);
c.Write;
```

В этом варианте никак не учитывается упорядоченность исходных массивов; они просто объединяются и сортируются. Понятно, что сортировка не может быть выполнена за один просмотр элементов, поэтому данный вариант будет работать медленнее, чем предыдущие. Однако он является очень кратким и, кроме того, для небольших наборов данных увеличение времени работы будет крайне незначительным. Даже для больших наборов данных это увеличение не будет слишком большим благодаря эффективной реализации метода сортировки.

Логическим завершением «идеи» варианта 3 будет решение, не использующее ни одной переменной и состоящее из единственного оператора, в котором вместо процедуры Sort для массивов используется запрос Sorted для последовательностей:

```
Task('Array63'); // Вариант 3а
(ReadArrReal(5) + ReadArrReal(5)).Sorted.Write;
```

Однако быстродействие при этом может только уменьшиться (см. анализ аналогичных вариантов для задачи Array88 в [1, п. 5.4]).

Замечание. В языке PascalABC.NET не определен порядок вычисления операндов в бинарных операциях (за исключением операций and и or), поэтому нельзя гарантировать, что первое слагаемое будет соответствовать исходному массиву a, а второе — массиву b. Однако в нашем алгоритме порядок слагаемых роли не играет.

При анализе быстродействия вариантов 1–3 были получены результаты, приведенные в таблице 5.1. В этих вариантах размеры исходных массивов равны size, поэтому в алгоритмах значения 5 были изменены на size, а значение 9 — на 2 * size – 1. Исходные массивы формировались с помощью выражений Range(0.0, 1.0, size – 1).ToArray и Range(0.0, 1.5, size – 1).ToArray. Измерялось время, потребовавшееся на инициализацию результирующего массива (или очереди) c и его заполнение элементами из уже созданных массивов (или очередей) a и b.

Таблица 5.1. *Быстродействие вариантов решения задачи Array63*

Размер size	Вариант 1	Вариант 2	Вариант 3
4000000	44	195	507
8000000	90	317	1036
16000000	180	680	2155
32000000	365	1247	4457

Глава 6. Многомерные массивы

В главах 6 и 7 рассматриваются средства языка PascalABC.NET, позволяющие обрабатывать *многомерные структуры*, т. е. структуры, для доступа к элементам которых требуется использовать два или более индексов. Традиционным вариантом для работы с подобными структурами является *многомерный массив*, имеющийся во всех реализациях языка Паскаль. Многомерным массивам и их дополнительным возможностям в PascalABC.NET посвящена глава 6. Альтернативой многомерным массивам являются *многомерные иерархические структуры* (в частности, *невыровненные массивы*, или *массивы массивов*) которые в ряде случаев оказываются более удобным средством. Вариантам реализации многомерных иерархических структур и их особенностям посвящена глава 7. В конце каждой главы, как обычно, приводятся решения задач из задачника Programming Taskbook, иллюстрирующие изученные структуры. В данных главах используются задачи из группы Matrix, посвященные обработке *матриц* — двумерных таблиц чисел.

6.1. Многомерные массивы: описание и инициализация

При знакомстве с массивами в [1, гл. 3] мы приводили пример статического массива с двумя индексами. В описании такого массива требуется указать несколько диапазонов индексов, разделяя их запятыми:

```
var a: array[1..10, 1..20] of real;
```

Полученный в результате массив называется *многомерным*. Из многомерных массивов чаще всего используются *двумерные* массивы, элементы которых имеют два индекса (как в описанном выше массиве a). При обращении к элементам многомерных массивов необходимо указывать в *общих* квадратных скобках значения всех индексов, разделяя их запятыми, например: a[1, 1], a[10, 20], a[i, j], где i и j — переменные целого типа.

Двумерные массивы удобно интерпретировать как *прямоугольные таблицы* элементов, состоящие из нескольких *строк* и *столбцов*; при этом первый индекс определяет, в какой строке находится элемент, а второй индекс — в каком столбце. Подобные таблицы в математике называются *матрицами*, поэтому так можно называть и двумерные массивы. *Размер*

двумерного массива-матрицы определяется двумя числами, между которыми указывается символ «×», например 10 × 20; при этом первое число (в нашем примере 10) задает количество строк матрицы, а второе (20) — количество столбцов. Если матрица имеет одинаковое количество строк и столбцов, то она называется *квадратной*. Ясно, что для задания размера квадратной матрицы достаточно указать количество ее строк. Эта величина называется *порядком* квадратной матрицы. Понятия квадратной матрицы и ее порядка также переносятся на соответствующие двумерные массивы.

Некоторые (или даже все) индексы в многомерном статическом массиве могут иметь тип, отличный от integer, например:

```
var b: array[1..10, 'A'..'Z'] of real;
```

Обратиться к элементу подобного массива можно, например, так: a[1, 'S']. Но эта возможность используется достаточно редко.

При появлении в Delphi Pascal версии 4 *динамических* массивов они были только одномерными. Однако платформа .NET и основанный на этой платформе язык PascalABC.NET допускают использование *многомерных динамических массивов*.

Как и в одномерном случае, при описании многомерных динамических массивов не указывается их размер (т. е. диапазон изменения каждого индекса), однако необходимо указать тип элементов, а также дополнительную информацию: *количество* индексов, т. е. *размерность* массива. Для указания размерности используются квадратные скобки, в которых указывается нужное число запятых, разделяющих «невидимые» индексы, например:

```
var a: array [,] of integer;
var b: array [,,] of real;
```

Здесь описан двумерный целочисленный массив a и трехмерный вещественный массив b.

Как уже отмечалось при знакомстве с одномерными динамическими массивами [1, п. 3.1], динамические массивы необходимо инициализировать, вызвав соответствующий конструктор. Конструктор для многомерных динамических массивов аналогичен конструктору для одномерных с тем дополнением, что в квадратных скобках требуется указать через запятую размер по каждому измерению, например:

```
a := new integer[3, 10];
b := new real[5, 10, 10];
```

Массив a в результате указанной инициализации получает размер 3 × 10 (и содержит 30 элементов), а массив b — размер 5 × 10 × 10 (и содержит 500 элементов). Как и в случае одномерных массивов, в результате

инициализации элементам многомерного массива присваиваются нулевые значения соответствующего типа.

Инициализацию можно совместить с описанием массива; при этом достаточно указать инициализирующее выражение, по которому компилятор PascalABC.NET *выведет* требуемый тип (поскольку вызов конструктора содержит для этого всю необходимую информацию — и тип элементов, и количество индексов):

```
var a := new integer[3, 10];
var b := new real[5, 10, 10];
```

Начальное значение каждого индекса динамического массива равно 0. Таким образом, например, для массива a первый индекс можно изменять в пределах от 0 до 2, а второй — в пределах от 0 до 9.

Если для работы с многомерными статическими массивами (как и с одномерными) PascalABC.NET предлагает только стандартное средство традиционного Паскаля — доступ к элементам по индексу, для многомерных динамических массивов можно использовать ряд свойств и методов.

Свойство Length, имеющееся у всех динамических массивов, возвращает *общее число* их элементов (и поэтому это свойство для многомерных массивов не очень полезно). Имеется свойство Rank (также не слишком полезное), в котором хранится *размерность* массива. Например, для ранее описанных динамических массивов a.Rank равно 2, а b.Rank равно 3. Свойство Rank, как и Length, доступно только для чтения.

Имеется метод GetLength(dim), возвращающий размер массива по измерению dim (измерения нумеруются от 0). Например, a.GetLength(0) = 3, а b.GetLength(2) = 10. Если попытаться указать отсутствующее измерение, то метод GetLength возбудит исключение.

В [1, п. 3.1] отмечалось, что размер динамического массива можно также получить с помощью *функции* Length с одним параметром — именем массива. Это справедливо и для многомерных массивов, причем для них можно дополнительно указать второй параметр dim — измерение, и тогда функция Length вернет размер по этому измерению. Например, Length(a, 1) вернет 10.

6.2. Вывод многомерных массивов

Для многомерных массивов *метод* Print/Println не определен, однако подобные массивы можно указывать в качестве параметров *процедур* Write/Writeln и Print/Println. Правда, вывод будет иметь не слишком наглядный вид. Например, при выполнении оператора Write(a) для ранее описанного массива размера 3×10, инициализированного нулевыми значениями, на экран будет выведен следующий текст (в виде *одной* строки):

```
[[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0]]
```

Если вспомнить аналогию двумерного массива с прямоугольной таблицей чисел (матрицей), то можно заметить, что вывод выполняется по *строкам* матрицы, причем каждая строка заключаются в квадратные скобки и, кроме того, в дополнительные квадратные скобки заключается сам список строк.

Именно в таком порядке (*по строкам*) хранятся элементы двумерного массива в памяти. Точнее, элементы многомерного массива располагаются в памяти таким образом, чтобы при их переборе быстрее изменялся *последний* из возможных индексов. Например, для *трехмерного* массива b, описанного выше, первым элементом является, естественно, элемент b[0,0,0], за которым в памяти будут располагаться элементы b[0,0,1], b[0,0,2] и так далее до b[0,0,9], после которого (так как достигнуто максимальное значение последнего индекса) будет изменен *предпоследний* индекс (а последний при этом будет сброшен в 0): b[0,1,0], b[0,1,1], …, b[0,1,9], b[0,2,0], …, b[4,9,9]. Для двумерных массивов такой способ перебора индексов как раз и дает перебор по строкам, например: a[0,0], a[0,1], …, a[0,9], a[1,0], a[1,1], …, a[1,9], …, a[2,9]. Вначале идут элементы первой строки (с первым индексом 0 и изменяющимся вторым индексом), затем элементы второй строки (с первым индексом 1) и так далее.

Для более наглядного вывода элементов матрицы, при котором элементы одного столбца располагаются на одной вертикали, требуется использовать двойной цикл следующего вида:

```
for var i := 0 to a.GetLength(0) - 1 do
begin
  for var j := 0 to a.GetLength(1) - 1 do
    Write(a[i, j]:6);
  Writeln;
end;
```

В результате матрица, инициализированная нулями, будет выведена в следующем виде:

```
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0
```

Обратите внимание на то, что в процедуре Write мы использовали атрибут форматирования 6, отделив его от элемента массива двоеточием. Это означает, что элемент должен выводиться в 6 экранных позициях и дополняться, при необходимости, слева пробелами [1, п. 1.2]. Без подобного атрибута нам не удалось бы наглядно вывести матрицу, элементы которой имеют разное количество десятичных цифр. Предположим, например, что все элементы первой строки по-прежнему равны 0, все элементы второй

равны 10, а все элементы третьей равны 100. Тогда матрица будет выведена в следующем виде:

```
 0   0   0   0   0   0   0   0   0   0
10  10  10  10  10  10  10  10  10  10
100 100 100 100 100 100 100 100 100 100
```

Если бы в предыдущем фрагменте программы в процедуре Write не указывался атрибут форматирования :6, то вывод был бы таким:

```
0000000000
10101010101010101010
10010010010010010010010010010010
```

Если бы использовалась процедура Print(a[i, j]), то вывод был бы не намного лучше, поскольку по-прежнему нарушалось бы выравнивание столбцов по вертикали:

```
0 0 0 0 0 0 0 0 0 0
10 10 10 10 10 10 10 10 10 10
100 100 100 100 100 100 100 100 100 100
```

Заметим, что в процедурах Print и Println *нельзя* указывать атрибуты форматирования.

Итак, вывод двумерных массивов в стандартных процедурах вывода допустим, но не обеспечивает нужной наглядности, и поэтому обычно приходится прибегать к использованию вложенных циклов.

Для вывода двумерных массивов в программах, использующих задачник Programming Taskbook, *нельзя* использовать процедуры Write/Writeln и Print/Println с параметрами-массивами, однако можно организовывать поэлементный вывод с использованием вложенных циклов. При этом программа не должна заботиться о форматировании выводимых данных, так как сам задачник обеспечивает вывод полученной матрицы в наглядном виде. Кроме того, в модуле PT4 предусмотрена специальная *процедура* WriteMatr**,** упрощающая вывод двумерных массивов (вместо имени WriteMatr можно использовать его синоним PrintMatr). Подчеркнем, что в то время как для быстрого вывода одномерных массивов при решении задач из задачника Programming Taskbook удобно использовать *методы* вида a.Write или a.Print, для вывода двумерных массивов предназначены *процедуры* вида WriteMatr(a) или PrintMatr(a).

6.3. Соглашения об именах вспомогательных переменных при работе с двумерными массивами

Еще раз обратимся к последнему примеру, в котором элементы двумерного массива (матрицы) перебираются в двойном цикле. При организации таких циклов в качестве переменной для перебора строк (т. е. значений

первого индекса) традиционно используется идентификатор i, а в качестве переменной для перебора столбцов (т. е. значений второго индекса) — идентификатор j. В результате элемент, обрабатываемый в цикле, имеет вид a[i, j]. Если внешний цикл является циклом по i, то это означает, что элементы перебираются *по строкам* (как в приведенном выше примере), если внешний цикл является циклом по j, то элементы перебираются *по столбцам*.

Чтобы избежать многократных вызовов метода GetLength (в котором легко сделать ошибку, указав неверный параметр), размеры двумерных массивов по каждому измерению часто сохраняют во вспомогательных переменных. Для хранения размера по первому индексу (т. е. для хранения числа *строк*) обычно используется переменная m, а для хранения размера по второму индексу (т. е. числа *столбцов*) — переменная n. Поэтому при описании матрицы произвольного размера говорят, что она имеет размер $m \times n$.

Если учитывать эти соглашения, то получим, что в цикле по i (т. е. по строкам) для перебора всех строк матрицы параметр i должен изменяться в пределах от 0 до $m - 1$, а в цикле по j для перебора всех столбцов параметр должен изменяться от 0 до $n - 1$. Для того чтобы легче запомнить связь между именем параметра цикла и диапазоном его изменения, следует обратить внимание на тот факт, что в латинском алфавите буквы i и m, используемые нами для перебора строк (т. е. *первых* индексов), *предшествуют* буквам j и n, используемым для перебора столбцов (*вторых* индексов).

При работе с двумерными массивами настоятельно рекомендуется следовать перечисленным соглашениям об именах, что позволит существенно уменьшить число ошибок, связанных с неверной индексацией, а также упростит анализ текстов программ.

6.4. Генерация и ввод двумерных массивов

В [1, п. 3.3, 3.7] были подробно описаны многочисленные средства PascalABC.NET, предназначенные для генерации и ввода *одномерных* динамических массивов. Для двумерных массивов эти средства применять нельзя, и их аналоги также не предусмотрены, за исключением аналогов функций генерации случайных массивов.

Для генерации целочисленных и вещественных матриц, заполненных случайными значениями, предусмотрены функции MatrixRandom и MatrixRandomReal со следующими заголовками:

```
function MatrixRandom(m: integer := 5; n: integer := 5;
  a: integer := 0; b: integer := 100): array [,] of integer
function MatrixRandomReal(m: integer := 5; n: integer := 5;
  a: real := 0; b: real := 10): array [,] of real
```

В каждой из этих функций можно указывать от 0 до 4 необязательных параметров. Первые два параметра m и n задают число строк и столбцов создаваемой матрицы (по умолчанию каждый из них равен 5), последние два параметра a и b задают диапазон, из которого выбираются случайным образом значения элементов. Как и для одномерных массивов, в случае целых чисел в диапазон входят все целые числа от a до b включительно (по умолчанию используется диапазон от 0 до 100), а в случае вещественных чисел диапазон представляет собой полуинтервал [a, b) (по умолчанию используется полуинтервал [0; 10)). В случае вещественных чисел при $a \neq b$ можно считать, что случайные числа выбираются из *интервала* (a, b), так как вероятность случайного выбора числа, равного a, практически равна нулю.

Параметр a может быть больше параметра b; вытекающие отсюда возможности подробно обсуждались в [1, п. 3.3].

Хотя в стандартной библиотеке PascalABC.NET отсутствуют средства, упрощающие ввод двумерных массивов, подобные средства можно использовать при выполнении заданий с применением задачника Programming Taskbook. Прежде всего, это *функции*, аналогичные функциям для ввода одномерных массивов с элементами трех базовых типов (integer, real и string):

```
function ReadMatrInteger([m, n: integer]): array [,] of integer
function ReadMatrReal([m, n: integer]): array [,] of real
function ReadMatrString([m, n: integer]): array [,] of string
```

Каждая из этих функций может вызываться либо без параметров, либо с двумя параметрами m и n. При вызове *без параметров* функция вначале считывает размеры матрицы (два числа — количество строк и столбцов), после чего создает матрицу указанного типа и размера, считывает значения ее элементов (по строкам) и возвращает сформированную матрицу. Если параметры m и n указаны, то функция выполняет все описанные выше действия, кроме считывания размеров, поскольку в качестве размеров берет значения параметров m и n.

Например, для описания и ввода исходной целочисленной матрицы достаточно указать следующий единственный оператор:

```
var a := ReadMatrInteger;
```

При этом тип переменной a (array [,] of integer) будет определен по типу возвращаемого значения функции.

Кроме того, для ввода матриц (в отличие от ввода одномерных массивов) в задачнике предусмотрена универсальная *процедура* ReadMatr, которая в ряде случаев может оказаться более удобной, чем описанные выше функции. Процедура ReadMatr может вызываться с двумя или тремя параметрами: ReadMatr(m, a) или ReadMatr(m, n, a), причем все параметры явля-

ются выходными. Первый вариант надо использовать, если в задании дается *квадратная* матрица порядка m, и поэтому для определения ее размера надо прочесть только одно число m (определяющее и количество строк, и количество столбцов). Второй вариант используется при вводе обычной прямоугольной матрицы, для которой указывается и число строк m, и число столбцов n.

Для использования процедуры ReadMatr необходимо предварительно описать ее параметры, например:

```
var m, n: integer;
var a: array [,] of integer;
ReadMatr(m, n, a);
```

Казалось бы, подобный способ ввода менее удобен, поскольку требует большего числа операторов. Однако он позволяет сразу использовать в программе переменные m и n в качестве размеров введенной матрицы, тогда как в случае использования функции ввода определять m и n пришлось бы явным (и более сложным) образом:

```
var a := ReadMatrInteger;
var m := a.GetLength(0);
var n := a.GetLength(1);
```

Кроме того, применение процедуры ReadMatr оказывается более удобным при вводе квадратных матриц, а также при использовании многомерных иерархических структур, которые будут обсуждаться в следующей главе (см. п. 7.4).

6.5. Дополнительные средства обработки двумерных массивов

По сравнению с одномерными массивами для многомерных динамических массивов предусмотрено значительно меньше дополнительных средств, связанных с их обработкой.

Прежде всего, многомерные массивы *не являются коллекциями*; это означает, что они не могут быть преобразованы в последовательности, и следовательно, к ним нельзя применять запросы последовательностей, описанные в [1, гл. 4].

Что же касается средств, аналогичных тем средствам одномерных массивов, которые были рассмотрены в [1, гл. 5], то для многомерных массивов можно применять только два из них: функцию Copy(a), позволяющую получить копию массива a (в том числе многомерного), и процедуру SetLength, позволяющую изменять размер массива.

Функция Copy(a) возвращает новый массив, содержимое которое совпадает с содержимым массива a. При этом размерность созданного массива и размеры по всем измерениям совпадают с аналогичными характери-

стиками массива a. Таким образом, если a — существующий многомерный массив, то для получения его копии достаточно использовать оператор

```
var b := Copy(a);
```

Подчеркнем, что оператор вида

```
var b := a;
```

имеет совсем другой смысл. Этот оператор лишь создает еще одну *ссылку* на массив a; в результате мы сможем обращаться с помощью двух переменных a и b *к одному и тому же массиву*. Таким же образом ведут себя одномерные динамические массивы [1, п. 5.1] и вообще все *ссылочные типы данных*. Напомним, что к ссылочным типам относятся все структуры, рассмотренные в главах 2–5.

Процедура SetLength, вариант которой для одномерных массивов был описан в [1, п. 5.2], может использоваться и для многомерных динамических массивов, позволяя изменять их размеры *по каждому измерению*. Параметрами этой процедуры является изменяемый массив и новые значения всех его измерений (таким образом, для изменения двумерных массивов процедура SetLength должна вызываться с тремя параметрами, для изменения трехмерных массивов — с четырьмя параметрами, и т. д.). Оставшиеся в новом массиве элементы сохраняют свои прежние значения, добавленные элементы имеют нулевые значения соответствующего типа.

Например, для того чтобы увеличить существующую матрицу a размера 3×4 до размера 6×5, достаточно выполнить следующий оператор:

```
SetLength(a, 6, 5);
```

При этом к каждой из имеющихся 3 строк исходной матрицы будет добавлен один нулевой элемент и, кроме того, к матрице будут добавлены еще 3 строки из 5 нулевых элементов каждая. Если считать, что элементы исходной матрицы были равны 1, то новая матрица будет иметь следующий вид (полужирным шрифтом выделены элементы исходной матрицы):

1 1 1 1 0
1 1 1 1 0
1 1 1 1 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

Процедуру SetLength удобно использовать для добавления в конец матрицы новых строк или столбцов. Следует, однако, учитывать, что в результате ее выполнения в памяти будет создана *копия* исходной матрицы нового размера, в которую будут записаны значения всех сохранившихся элементов. Таким образом (как уже было отмечено в [1, п. 5.3]), изменение размера массива является *длительной* и *ресурсоемкой* операцией.

6.6. Использование двумерных массивов при решении задач

Начнем с задачи **Matrix7**, в которой дана вещественная матрица размера m × n и число k — порядковый номер некоторой строки данной матрицы. Требуется вывести все элементы строки с номером k. При решении задач из группы Matrix необходимо учитывать, что в их формулировках предполагается, что *номера* строк и столбцов начинаются с 1 (тогда как *индексы* двумерных динамических массивов начинаются с 0).

В первом варианте решения воспользуемся функцией ReadMatrReal, выполняющей ввод как размера, так и элементов исходной матрицы:

```
Task('Matrix7'); // Вариант 1
var a := ReadMatrReal;
var k := ReadInteger - 1;
for var j := 0 to a.GetLength(1) - 1 do
  Write(a[k, j]);
```

Для вывода мы использовали обычный цикл, в котором количество столбцов определили с помощью метода GetLength. Обратите внимание на то, что в переменную k мы записали прочитанное значение *минус* 1; тем самым мы перешли от данного в задаче номера строки к ее индексу.

Во втором варианте решения будем использовать процедуру ReadMatr, также вводящую все данные, связанные с матрицей, но дополнительно записывающую ее размеры m и n в отдельные переменные:

```
Task('Matrix7'); // Вариант 2
var m, n: integer;
var a: array [,] of real;
ReadMatr(m, n, a);
var k := ReadInteger - 1;
for var j := 0 to n - 1 do
  Write(a[k, j]);
```

Благодаря наличию переменной n заголовок цикла стал более коротким. Впрочем, в данной задаче использование процедуры ReadMatr не дает особых преимуществ; в частности, в нашей программе даже не потребовалось обратиться к переменной m, содержащей число строк матрицы.

В каждом из приведенных решений при вводе исходной матрицы мы обошлись без цикла благодаря использованию специальных функций и процедур, предусмотренных в задачнике Programming Taskbook. Для вывода нам пришлось использовать цикл, поскольку средства вывода для коллекций (методы Write или Print), также имеющиеся в задачнике, нельзя применять к *частям* двумерных массивов. Однако имеется универсальный способ, позволяющий применять методы обработки коллекций к двумер-

ным массивам: это использование *генераторов последовательностей*. Например, следующий вариант решения вообще не содержит циклов:

```
Task('Matrix7'); // Вариант 3
var m, n: integer;
var a: array [,] of real;
ReadMatr(m, n, a);
var k := ReadInteger - 1;
SeqGen(n, j -> a[k, j]).Write;
```

В этом варианте мы генерируем последовательность, содержащую k-ю строку исходной матрицы, после чего просто применяем к ней метод Write для вывода всех ее элементов. Разумеется в данном случае такой вариант не дает особых преимуществ, однако с помощью такого же подхода мы можем применять к нужным частям двумерного массива весь арсенал *запросов* для последовательностей, что может оказаться очень удобным. Заметим также, что применение в программе вспомогательной последовательности не приводит к дополнительному существенному расходу памяти: мы уже неоднократно отмечали, что элементы последовательностей не хранятся в памяти одновременно, а генерируются непосредственно при их переборе (в цикле foreach или при выполнении некоторых запросов, например Write).

Обратите внимание на то, что в лямбда-выражении, использованном в функции SeqGen мы выбрали в качестве имени параметра букву j (хотя обычно в таких ситуациях указывается буква i — стандартное имя переменной для перебора индексов). Причина состоит в том, что во всех вариантах решений мы следовали соглашениям, описанным в п. 6.3, согласно которым для перебора значений *второго* индекса (отвечающего за номер или индекс столбца) желательно использовать параметр j. В частности, эти соглашения «страхуют» нас от возможных описок, связанных с неверным указанием диапазона изменения индекса: достаточно помнить, что с индексом i (номером строки) связывается число m (количество строк), а с индексом j (номером столбца) связывается число n (количество столбцов); при этом обе буквы в первой паре (i, m) *предшествуют* в алфавите буквам из второй пары (j, n).

От задачи, связанной с *выводом* части матрицы, перейдем к задаче **Matrix24** из подгруппы, посвященной *анализу элементов* матриц. В этой задаче дана вещественная матрица размера $m \times n$ и требуется найти максимальный элемент в каждом столбце этой матрицы.

Первый вариант решения является традиционным вариантом, использующим вложенные циклы:

```
Task('Matrix24'); // Вариант 1
var m, n: integer;
```

```
var a: array [,] of real;
ReadMatr(m, n, a);
for var j := 0 to n - 1 do
begin
  var max := a[0, j];
  for var i := 1 to m - 1 do
    if a[i, j] > max then
      max := a[i, j];
  Write(max);
end;
```

Здесь применение процедуры ReadMatr более оправдано (по сравнением с задачей Matrix7), так как в алгоритме используется и число строк m, и число столбцов n. Поскольку требуется обработать *столбцы* матрицы, внешний цикл обеспечивает перебор столбцов (и поэтому имеет параметр j). В этом цикле явным образом реализован алгоритм нахождения максимального элемента (см. п. 1.1) среди всех элементов столбца j, т. е. элементов вида a[i, j], где i меняется от 0 до m – 1. Сразу после нахождения максимума он выводится, поэтому нам достаточно иметь единственную переменную max для хранения того максимума, который вычисляется в данный момент.

Вместо явной реализации алгоритма нахождения максимума можно применить запрос Max (см. [1, п. 4.1] и п. 1.1), что гораздо короче, но требует формирования той последовательности, к которой будет применен запрос. Как это сделать, мы уже обсуждали при рассмотрении последнего варианта решения задачи Matrix7:

```
Task('Matrix24'); // Вариант 2
var m, n: integer;
var a: array [,] of real;
ReadMatr(m, n, a);
for var j := 0 to n - 1 do
  Write(SeqGen(m, i -> a[i, j]).Max);
```

В данном случае мы использовали в лямбда-выражении параметр i, так как для формирования последовательности нам надо перебирать различные *строки* матрицы.

Во втором варианте решения мы избавились от внутреннего цикла по i (являющегося частью алгоритма нахождения максимального элемента), но сохранили внешний цикл по j, в котором перебираются столбцы и для каждого столбца выводится результат. Однако можно обойтись и без этого цикла, сформировав *последовательность* результатов и выведя ее запросом Write (ср. со способом вывода данных, использованным в варианте 3 решения задачи Matrix7):

```
Task('Matrix24'); // Вариант 3
var m, n: integer;
var a: array [,] of real;
ReadMatr(m, n, a);
SeqGen(n, j -> SeqGen(m, i -> a[i, j]).Max).Write;
```

Мы получили программу, в которой алгоритм решения (и вывода) состоит из единственного оператора. Правда, в нем используются вложенные вызовы функций-генераторов. В подобной ситуации особенно важно применять подходящие имена для параметров лямбда-выражений. В частности, в нашем случае мы можем, глядя на эти параметры, сразу определить, что задача связана с обработкой *столбцов*, для каждого из которых выводится некоторая его характеристика.

Завершая этот пункт, рассмотрим задачу **Matrix59**, которая относится к подгруппе задач на *преобразование матриц*. В ней также дается вещественная матрица размера m × n и требуется зеркально отразить ее строки относительно горизонтальной оси симметрии. Иными словами, требуется расположить строки матрицы в обратном порядке (т. е. *инвертировать* строки матрицы).

Во всех задачах этой подгруппы надо вывести преобразованную матрицу; это легко сделать с помощью стандартной процедуры WriteMatr, входящей в задачник:

```
Task('Matrix59');
var m, n: integer;
var a: array [,] of real;
ReadMatr(m, n, a);
for var i := 0 to m div 2 - 1 do
  for var j := 0 to n - 1 do
    Swap(a[i,j], a[m-i-1,j]);
WriteMatr(a);
```

В данном случае проще всего реализовать алгоритм инвертирования строк явно, с применением вложенных циклов. Во внешнем цикле перебираются пары строк, которые надо поменять местами; это строки с индексами 0 и m – 1, 1 и m – 2, …, i и m – i – 1 и т. д. (всего имеется m div 2 таких пар). Внутренний цикл (по j) требуется для того, чтобы поменять местами *все элементы* этих строк. Для обмена значений двух переменных используется обобщенная процедура Swap (входящая в стандартную библиотеку PascalABC.NET), которую можно применять к *любым* переменным, имеющим *одинаковый* тип.

Особенностью приведенного алгоритма является то, что порядок следования циклов можно изменить: внешним циклом сделать цикл по j, а внутренним — цикл по i. При описании такого варианта можно сказать,

что во внешнем цикле перебираются столбцы матрицы, а внутренний цикл обеспечивает инвертирование каждого столбца.

Следует заметить, что в приведенном варианте решения производится обмен значений почти для всех элементов матрицы (неизменными остаются только элементы «средней» строки, причем только в случае, если матрица имеет нечетное число строк); таким образом, число обменов для матриц большого размера тоже будет большим. Можно разработать алгоритм, требующий существенно меньшего количества операций обмена, однако для этого исходную матрицу потребуется сохранить в многомерной структуре, отличной от двумерного массива. Подобным *иерархическим* многомерным структурам посвящена следующая глава.

Глава 7. Многомерные иерархические структуры

7.1. Массивы массивов (невыровненные массивы): описание и инициализация

Многомерные структуры данных можно реализовать не только в виде многомерных массивов. В некоторых языках программирования (например, в C/C++) многомерные массивы отсутствуют, что не препятствует создавать и использовать в программах структуры для хранения, например, матриц. Для этого применяются «обычные» одномерные индексируемые структуры, элементами которых также являются индексируемые структуры. В результате возникают *структуры структур* (иерархические структуры), для доступа к элементам которых, как и для доступа к элементам двумерных массивов, можно использовать *два индекса*. Правда, в данном случае каждый из индексов относится к своей собственной структуре и поэтому должен указываться в отдельных скобках, например a[i][j].

Самым распространенным примером подобных иерархических структур является *массив массивов*, называемый также (по причинам, которые будут объяснены далее) *невыровненным массивом* (jagged array). Статические невыровненные массивы практически не используются, поэтому мы ограничимся рассмотрением динамических невыровненных массивов, описание которых имеет следующий вид:

```
var a: array of array of integer;
```

В данном случае описан невыровненный массив для хранения целых чисел. Подобное описание означает, что a является массивом, элементами которых являются массивы целых чисел. Чтобы это стало более понятным, можно *мысленно* добавить в описание скобки: array of (array of integer) (заметим, что в программе скобки в подобных описаниях использовать запрещено). Таким образом, запись a[i] означает обращение к некоторому массиву целых чисел. Но первый индекс в двумерном массиве-матрице обозначает, по нашему соглашению, индекс *строки*, поэтому удобно считать, что массив a[i] представляет собой *строку* элементов нашего невыровненного массива. Тогда запись a[i][j] обозначает элемент i-й строки с индексом j, что соответствует смыслу второго индекса для двумерного массива (поскольку элемент матрицы со вторым индексом j находится в j-м столбце). Итак, вы-

ражение a[i][j] позволяет обратиться к элементу, находящемуся в i-й строке и j-м столбце, как и аналогичное выражение a[i, j] для двумерного массива.

Для *инициализации* невыровненного массива потребуется выполнить более сложные действия по сравнению с инициализацией многомерного массива. В качестве примера рассмотрим действия по инициализации описанного выше невыровненного массива a для хранения матрицы размера 3×10.

Вначале необходимо выделить память для самого массива a (т. е. массива, содержащего строки нашей матрицы). *Элементы* массива a имеют тип array of integer, который *нельзя* указывать в конструкторе массива new, поскольку имя этого типа состоит *более чем из одного слова* (попытка использовать выражение new (array of integer)[3] или new array of integer[3] приведет к ошибке компиляции). Для решения возникшей проблемы можно ввести вспомогательный тип — *синоним* типа array of integer, состоящий из одного слова, например:

```
type arrInt = array of integer;
```

Это описание необходимо поместить в раздел описаний программы (перед словом begin, открывающим раздел операторов). Теперь для вызова конструктора для массива a можно использовать следующий оператор:

```
a := new arrInt[3];
```

Заметим, что нам не потребовалось изменять описание массива a, хотя теперь его можно было бы представить в следующем виде:

```
var a: array of arrInt;
```

Дело в том, что для динамических массивов в PascalABC.NET реализована *структурная эквивалентность* типов (ранее о структурной эквивалентности говорилось в [1, п. 5.1]), благодаря которой любые типы, введенные для динамических массивов, содержащих элементы одного и того же типа (в нашем случае типа integer) считаются эквивалентными.

Итак, проблему с вызовом конструктора для невыровненного массива можно решить путем введения имени-синонима для типа элементов этого массива.

Однако имеется более простое решение, основанное на применении процедуры SetLength. Напомним, что данная процедура может использоваться для инициализации массивов [1, п. 5.2], поэтому в нашем случае инициализацию массива a можно выполнить следующим образом:

```
SetLength(a, 3);
```

При этом никакой вспомогательный тип описывать и использовать не потребуется.

На данном этапе с переменной a связан массив, состоящий из трех элементов типа array of integer (т. е. из трех строк создаваемой нами матри-

цы). Однако все эти элементы имеют *нулевые значения* типа array of integer. Поскольку тип array of integer является ссылочным, его нулевым значением является нулевая ссылка nil. Итак, ссылки на каждую из созданных строк (a[0], a[1] и a[2]) пока не указывают на реальные строки в памяти. Фактически их можно рассматривать как имена описанных, но не инициализированных целочисленных массивов.

Поэтому на втором этапе инициализации невыровненного массива a необходимо инициализировать его строки. Это действие обычно выполняется в цикле, в котором перебираются все строки массива и выполняется их инициализация. Поскольку типом элементов строки является integer, в данном случае удобно использовать конструктор new:

```
for var i := 0 to a.Length - 1 do
  a[i] := new integer[10];
```

Можно также использовать вариант инициализации с помощью процедуры SetLength:

```
for var i := 0 to a.Length - 1 do
  SetLength(a[i], 10);
```

Разумеется, вместо выражения a.Length - 1 можно просто указать число 2, однако использованный нами вариант является более гибким, поскольку не требует корректировки при изменении количества строк в создаваемой матрице.

В результате этого этапа инициализации все строки нашей матрицы будут содержать по 10 нулевых целочисленных элементов, в чем можно убедиться, выведя массив a процедурой Write(a):

```
[[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0]]
```

Приведем полный текст кода, позволяющего описать и инициализировать невыровненный целочисленный массив размера 3×10:

```
var a: array of array of integer;
SetLength(a, 3);
for var i := 0 to a.Length - 1 do
  a[i] := new integer[10];
```

Теперь для доступа к элементам созданного массива мы можем использовать операцию индексирования, указывая каждый индекс в своих квадратных скобках, например a[2][6].

Замечание. Используя генераторы массивов [1, п. 3.3], можно реализовать инициализацию невыровненного массива в виде *единственного оператора* (соответствующие примеры будут приведены в п. 7.2, а также в п. 7.5 при решении задачи Matrix24). Однако вариант в виде цикла является более универсальным, так как может использоваться при инициализа-

ции и других видов многомерных структур, не требуя при этом вызовов дополнительных запросов (см. п. 7.3).

7.2. Использование невыровненных массивов для хранения матриц специального вида. Преимущества невыровненных массивов

Мы видим, что невыровненные массивы требуют более сложных действий при своем создании по сравнению с созданием многомерных массивов. Однако подобные массивы имеют ряд преимуществ при последующем использовании.

Первое преимущество состоит в том, что в невыровненном массиве можно хранить *строки разного размера*, что бывает удобно при работе с матрицами специального вида. Именно с этим обстоятельством связано появление термина «невыровненный массив»: его строки не обязаны «выравниваться» по одной длине.

Предположим, что в нашей программе требуется сохранить матрицу размера 5×5 (т. е. квадратную матрицу порядка 5), для которой заранее известно, что все ее элементы, расположенные *выше* главной диагонали, равны 0 (*главной диагональю* называется диагональ квадратной матрицы, содержащая элементы с одинаковыми индексами: a[0, 0], a[1, 1], …). Приведем пример подобной матрицы, выделив полужирным шрифтом элементы ее главной диагонали:

1 0 0 0 0
2 **2** 0 0 0
3 3 **3** 0 0
4 4 4 **4** 0
5 5 5 5 **5**

Такие матрицы называются *треугольными*, или *нижнетреугольными* (последнее название подчеркивает, что интерес представляет только содержимое той части матрицы, которая образует нижний левый треугольник).

Если хранить нижнетреугольную матрицу в виде обычного двумерного массива, то *примерно половина* выделенной памяти будут использоваться для хранения ненужных нулевых значений. Разумеется, для матрицы порядка 5 этот перерасход памяти будет ничтожным, однако для матриц большого размера он может стать весьма значительным. Нетрудно заметить, что количество элементов в «незанятой» части треугольной матрицы порядка n равно $1 + 2 + \ldots + (n - 1)$ (в первой строке имеется $(n - 1)$ таких элементов, во второй — $(n - 2)$ и т. д. до 1 нулевого элемента в предпоследней строке). Используя формулу суммы членов арифметической прогрессии, нетрудно подсчитать, что это количество равно $n \cdot (n - 1)/2$. Таким

образом, для нижнетреугольной матрицы, имеющей размер 1000×1000, «лишних» элементов будет $1000 \cdot 999/2 = 499500$, т. е. почти полмиллиона!

Для хранения треугольных матриц хорошо подходят невыровненные массивы, позволяющие выделять память только для ненулевой части каждой строки. Приведем фрагмент кода, в котором создается указанная выше треугольная матрица, после чего ее содержимое выводится на экран:

```
var a: array of array of integer;
SetLength(a, 5);
for var i := 0 to a.Length - 1 do
  a[i] := ArrFill(i + 1, i + 1);
Writeln(a);
```

В результате выполнения этого фрагмента на экран будет выведен следующий текст:

```
[[1],[2,2],[3,3,3],[4,4,4,4],[5,5,5,5,5]]
```

В этом фрагменте цикл используется не только для инициализации строк, но и для генерации их элементов. Для подобной генерации в нашем случае наиболее удобно использовать функцию ArrFill(n, x) (см. [1, п. 3.3]), создающую массив размера n, все элементы которого получают значение x.

Приведенный фрагмент демонстрирует **второе** важное преимущество невыровненных массивов (и других иерархических многомерных структур) перед многомерными массивами: возможность применения к ним всего арсенала средств, имеющихся в PascalABC.NET для обработки *одномерных массивов и последовательностей* (а также любых структур, описанных в главах 2–5).

Поскольку не только строки матрицы a являются одномерными массивами (имеющими элементы типа integer), но и сама матрица a реализована в виде одномерного массива (с элементами типа array of integer), мы можем пойти еще дальше и использовать подходящую функцию-генератор для создания невыровненного массива a. В результате все действия по созданию нашей треугольной матрицы можно будет представить в виде единственного оператора, содержащего *вложенные функции-генераторы*:

```
var a := ArrGen(5, i -> ArrFill(i + 1, i + 1));
```

При обработке невыровненных массивов (и других иерархических многомерных структур) многие операции не только оформляются проще и нагляднее (за счет использования, например, запросов), но и являются более быстродействующими по сравнению с аналогичными операциями для многомерных массивов. В это состоит **третье** их преимущество.

В качестве примера рассмотрим задачу *инвертирования* строк исходной треугольной матрицы a, т. е. изменения их порядка на противоположный. Инвертированная матрица a должна иметь следующий вид:

5 5 5 5 **5**
4 4 4 **4** 0
3 3 **3** 0 0
2 **2** 0 0 0
1 0 0 0 0

Теперь выделенные элементы (ранее располагавшиеся на главной диагонали) находятся на *побочной диагонали* матрицы. Заметим, что для элементов a[i, j], расположенных на побочной диагонали матрицы a, выполняется следующее соотношение: i + j = n – 1, где n — порядок матрицы.

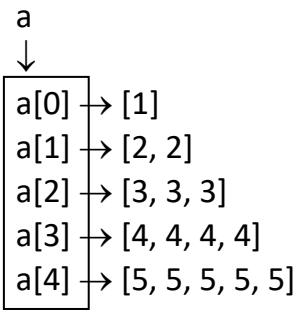
Для решения поставленной задачи достаточно воспользоваться процедурой инвертирования массива Reverse [1, п. 3.7]:

```
Reverse(a);
```

Если после этого еще раз вызвать процедуру Writeln(a), то на экран будет выведено содержимое инвертированной матрицы a:

[[5,5,5,5,5],[4,4,4,4],[3,3,3],[2,2],[1]]

Помимо краткости кода приведенное решение оказывается *очень быстрым*, так как процедура Reverse меняет порядок лишь *ссылок*, содержащихся в массиве a. Сами строки, на которые указывают эти ссылки, не изменяются. Чтобы более наглядно продемонстрировать этот факт, удобно представить данные, связанные с *исходной* матрицей a в виде следующей схемы:



Здесь стрелка, указывающая вниз, означает, что в переменной a содержится ссылка на массив [a[0], a[1], a[2], a[3], a[4]], а стрелки, указывающие вправо, означают, что каждый из элементов a[i], в свою очередь, содержит ссылку на массив целых чисел — соответствующую строку матрицы. Процедура Reverse изменяет только содержимое элементов a[0], …, a[4]. В результате ее выполнения элемент a[0] получает ссылку на массив [5, 5, 5, 5, 5] (которая ранее хранилась в a[4]), элемент a[1] — ссылку на массив [4, 4, 4, 4] (которая ранее хранилась в a[3]), и т. д. Расположение в памяти самих массивов [5, 5, 5, 5, 5], [4, 4, 4, 4], …, [1] не изменяется. Таким образом, для решения задачи нам потребовалось поменять местами содержимое *двух пар* ячеек памяти с адресами (пары a[0], a[4] и пары a[1], a[3]). Если бы эта задача решалась для матрицы размера $n \times n$, то потребовалось бы n div 2 опе-

раций обмена (например, для $n = 1000$ число операций обмена было бы равно 500).

Если бы мы инвертировали строки в обычном двумерном массиве, то нам бы потребовалось явным образом менять *содержимое самих строк* (см. решение задачи Matrix59 в п. 6.6), поэтому для матрицы размера $n \times n$ потребовалось бы выполнить $n \cdot (n \text{ div } 2)$ операций обмена целых чисел — элементов матрицы. Для $n = 1000$ это составляет *полмиллиона* операций обмена. Правда, в нашем случае это количество можно было бы уменьшить, меняя местами только *ненулевые части* строк треугольной матрицы, но эта модификация, во-первых, усложнила бы алгоритм и, во-вторых, все равно потребовала бы очень большого числа обменов (например, для $n = 1000$ количество обменов было бы равно 375250).

7.3. Другие варианты многомерных иерархических структур

Хотя массивы массивов являются наиболее естественной альтернативой двумерным массивам, иногда более предпочтительным является конструирование многомерных структур на основе других коллекций (рассмотренных нами в главах 2–5). Поскольку обработка многомерных структур обычно требует частого обращения к их элементам по индексу, для конструирования таких структур наилучшим образом подходят те коллекции, для которых реализована операция индексирования, прежде всего списки на базе массивов List<T> (см. п. 2.1).

Для создания двумерных иерархических структур массивы и списки можно комбинировать четырьмя способами, получая, кроме уже рассмотренных массивов массивов, также массивы списков, списки массивов и списки списков.

Приведем фрагмент, позволяющий описать и инициализировать список списков для хранения вещественной матрицы размера 3×8 со следующими значениями:

```
 1  2  3  4  5  6  7  8
 2  4  6  8 10 12 14 16
 3  6  9 12 15 18 21 24
```

Вариант, использующий конструкторы списка, выглядит следующим образом:

```
// Вариант 1
var a := new List<List<integer>>(3);
for var i := 0 to a.Capacity - 1 do
begin
  a.Add(new List<integer>(8));
  for var j := 0 to a[i].Capacity - 1 do
```

```
    a[i].Add((i + 1) * (j + 1));
end;
```

В данном случае при инициализации списка a вполне допустимо использовать конструктор класса List<T>, так как в угловых скобках можно указывать любые типы данных, в том числе и состоящие из более чем одного слова. Для добавления к спискам новых элементов мы использовали метод Add.

В конструкторах списков мы явно указали емкость, что позволило, во-первых, сразу выделить память требуемого размера для хранения элементов списка (в виде внутреннего массива) и, во-вторых, использовать свойство Capacity в заголовках последующих циклов.

Если к данному фрагменту добавить оператор вывода Write(a), то при его выполнении на экран будут выведены следующие данные:

```
[[1,2,3,4,5,6,7,8],[2,4,6,8,10,12,14,16],[3,6,9,12,15,18,21,24]]
```

Напомним, что для более наглядного вывода по строкам, при котором сохраняется вертикальное выравнивание столбцов, можно использовать двойной цикл с явно указанными атрибутами форматирования:

```
for var i := 0 to a.Count - 1 do
begin
  for var j := 0 to a[i].Count - 1 do
    Write(a[i][j]:4);
  Writeln;
end;
```

При выполнении данного фрагмента вид матрицы будет в точности соответствовать виду, приведенному в начале данного пункта. Аналогичного результата можно было бы добиться, используя для вывода вложенные циклы foreach:

```
foreach var row in a do
begin
  foreach var e in row do
    Write(e:4);
  Writeln;
end;
```

В качестве параметра внешнего цикла foreach мы использовали имя row (переводится как «ряд, строка»), чтобы подчеркнуть, что в нем перебираются строки исходной матрицы.

Теперь приведем второй вариант создания списка списков с тем же содержимым, основанный на использовании *генераторов последовательностей*. Этот вариант будет состоять из единственного, хотя и достаточно длинного оператора:

```
// Вариант 2
var a := SeqGen(3,
  i -> SeqGen(8, j -> (i + 1) * (j + 1)).ToList).ToList;
```

Внешняя функция-генератор создает 3 строки, для заполнения которых вызывается внутренняя функция-генератор (с первым параметром 8). Каждая из полученных последовательностей преобразуется в список запросом ToList.

Анализ этих алгоритмов для генерации больших матриц размера $n \times n$ показывает, что алгоритм 1 (с использованием циклов) обладает лучшим быстродействием. В таблице 7.1 приводится длительность выполнения алгоритмов 1 и 2 для матриц порядка n при n = 5000, 10000, 20000.

Таблица 7.1. *Быстродействие вариантов инициализации больших матриц*

Порядок матрицы n	Вариант 1	Вариант 2
5000	248	716
10000	985	2753
20000	3842	9451

Более высокое быстродействие варианта 1 объясняется тем обстоятельством, что при преобразовании последовательности в список запросом ToList нужная емкость не устанавливается сразу, а растет постепенно, за счет многократного удвоения (эта особенность запроса ToList ранее обсуждалась в п. 2.7).

7.4. Дополнительные средства задачника Programming Taskbook для ввода и вывода многомерных иерархических структур

Так как при выполнении многих заданий на обработку матриц структуры типа «массив массивов» и «список списков» могут оказаться наиболее подходящими, в задачнике Programming Taskbook предусмотрены средства, упрощающие вывод, инициализацию и ввод подобных структур. Эти средства реализованы в виде дополнительных вариантов процедур WriteMatr/PrintMatr и ReadMatr (варианты данных процедур для работы с двумерными массивами были описаны в п. 6.2 и 6.4).

Кроме вывода двумерных массивов с целочисленными, вещественными и строковыми элементами процедуры WriteMatr/PrintMatr могут использоваться для вывода массивов типа array of array of T и списков типа List<List<T>>, где в качестве T могут быть указаны типы integer, real и string. Структуры этих же типов можно использовать в качестве выходного параметра a в вариантах процедуры ReadMatr(m, n, a) и ReadMatr(m, a).

Таким образом, если в некотором задании дана, например, матрица целых чисел размера m × n, причем из условия задачи следует, что наибо-

лее подходящей структурой для хранения данной матрицы является невыровненный массив (массив массивов), то для организации ввода исходных данных достаточно использовать следующий фрагмент:

```
var m, n: integer;
var a: array of array of integer;
ReadMatr(m, n, a);
```

Обратите внимание на то, что этот фрагмент отличается от аналогичного фрагмента для ввода двумерного массива, приведенного в п. 6.4, лишь описателем переменной a (в примере из п. 6.4 он имел вид array [,] of integer, а в данном случае используется тип array of array of integer).

Если бы подобный вариант процедуры ReadMatr не был реализован, то для инициализации и ввода массива a потребовался бы существенно больший фрагмент кода, даже если бы в нем применялись средства задачника для ввода одномерных массивов, например:

```
var m := ReadInteger;
var n := ReadInteger;
var a: array of array of integer;
SetLength(a, m);
for var i := 0 to m - 1 do
  a[i] := ReadArrInteger(n);
```

Напомним про полезную возможность, предоставляемую процедурой ReadMatr — запись в переменные m и n размеров введенной матрицы. Это позволяет в дальнейшем использовать в программе переменные m и n вместо более длинных выражений (в нашем случае — a.Length и a[0].Length соответственно).

7.5. Использование многомерных иерархических структур при решении задач

Познакомившись с возможностями многомерных иерархических структур, вернемся к уже рассмотренным в п. 6.6 задачам и попытаемся реализовать для них более эффективные (в том или ином отношении) решения, использующие данные структуры.

В случае задачи **Matrix7**, в которой требуется вывести элементы k-й строки, применение невыровненного массива приводит к самому короткому варианту решения:

```
Task('Matrix7'); // Вариант 4
var m, n: integer;
var a: array of array of real;
ReadMatr(m, n, a);
var k := ReadInteger - 1;
```

```
a[k].Write;
```

Этот вариант, как и вариант 3, использует *метод* Write, но, в отличие от варианта 3, в котором нам потребовалось явным образом создавать последовательность, содержащую элементы k-й строки, здесь мы просто обращаемся к элементу a[k], который *уже* представляет собой k-ю строку и, будучи одномерным массивом, допускает применение метода Write. Здесь наглядно проявляется одно из преимуществ невыровненных массивов (подробно рассмотренных в п. 7.2), а именно — возможность доступа не только к отдельным элементам матрицы, но и к ее строкам (как одномерным массивам).

К сожалению, иерархические структуры не обеспечивают *одновременного* быстрого доступа к строкам и столбцам матрицы. При обсуждении иерархических структур в предыдущих пунктах мы всегда предполагали, что элементы матрицы в них хранятся *по строкам*, т. е. что элементы верхнего уровня (вида a[i]) представляют собой строки матрицы. Это наиболее естественный подход, поскольку он приводит к записи элементов вида a[i][j], где первым указывается индекс строки, а вторым — индекс столбца. По этой же причине процедура ввода ReadMatr реализована таким образом, чтобы типы array of array of T и List<List<T>> интерпретировались как наборы (массивы или списки) *строк* матрицы, и то же самое предполагается в процедуре вывода WriteMatr.

Однако в ситуациях, когда задача требует обработки *столбцов* матрицы, такой подход к хранению матриц в иерархических структурах оказывается неудобным. Разумеется, в этом случае можно реализовать вариант решения с двумерным массивом и даже использовать в этом варианте запросы, применив их к последовательностям, созданным с помощью функций-генераторов. Подобные варианты мы рассматривали при решении задачи Matrix24 в п. 6.6. Но имеется и другая возможность: организовать в иерархической двумерной структуре хранение элементов матрицы *по столбцам*. Такая структура a (массив массивов или список списков) будет состоять из элементов, которые удобно обозначать как a[j], чтобы подчеркнуть, что каждый такой элемент представляет собой *столбец* матрицы с индексом j. К сожалению, использование структур, хранящих данные по столбцам, приводит к двум проблемам. Во-первых, для них отсутствуют специальные процедуры ввода и вывода, поэтому инициализацию и заполнение, а также вывод таких структур придется выполнять «вручную». Во-вторых, при обращении к элементам таких структур придется использовать непривычный порядок следования индексов: a[j][i] (в данной ситуации особенно важно следовать соглашениям об именованиях индексов, приведенным в п. 6.3). В качестве «компенсации» за отмеченные неудобства мы сможем реализовать короткие, наглядные и эффективные алгоритмы обработки столбцов.

Еще раз обратимся к задаче **Matrix24**, в которой требуется найти максимальные значения в каждом столбце (см. п. 6.6), и приведем вариант решения, в котором для хранения матрицы используется массив столбцов:

```
Task('Matrix24'); // Вариант 4
var m := ReadInteger;
var n := ReadInteger;
var a:= ArrGen(n, j -> new real[m]);
for var i := 0 to m - 1 do
  for var j := 0 to n - 1 do
    a[j][i] := ReadReal;
a.Select(col -> col.Max).Write;
```

Для инициализации массива столбцов a мы воспользовались функцией-генератором ArrGen, что позволило инициализировать и сам массив, и его элементы-столбцы в единственном операторе. Обратите внимание на то, что размер массива a равен n (т. е. числу столбцов), а в конструкторе для j-го элемента этого массива (который вызывается в лямбда-выражении) указывается переменная m, т. е. число строк: new real[m].

Ввод элементов приходится организовывать традиционным способом: с помощью двойного цикла. Поскольку исходные данные для матрицы *всегда* передаются задачником в программу *по строкам*, внешний цикл имеет параметр i, а внутренний — параметр j. Важно указать эти параметры в правильном порядке при обращении к элементам матрицы (в данном случае индекс столбца указывается первым, а индекс строки — вторым).

Наградой за усилия, связанные с формированием структуры, содержащей матрицу по столбцам, является короткое и наглядное решение задачи — в виде единственного оператора. Собственно говоря, идея решения является той же, что и в варианте 3. Однако в новом варианте *не требуется применять вложенные функции-генераторы*: мы просто формируем по *уже имеющемуся* набору столбцов набор их максимальных элементов (используя запрос проецирования Select), после чего выводим полученный набор методом Write. Для большей наглядности мы использовали имя col (от англ. column — «столбец») в качестве параметра лямбда-выражения в запросе Select. Напомним, что для параметра, перебирающего строки, удобно использовать имя row.

Переходя к задачам на *преобразование* матриц, еще раз напомним, что представление матрицы в виде массива или списка строк (или столбцов) позволяет особенно эффективно выполнять преобразования, связанные и перестановками строк (или, соответственно, столбцов). Причина этого подробно обсуждалась в конце п. 7.2; она связана с тем, что для изменения порядка следования, например, двух строк в массиве строк достаточно изменить лишь порядок *ссылок на эти строки*, не затрагивая те участки памяти, в которых хранятся элементы строк.

В качестве иллюстрации можно было бы привести соответствующее решение задачи Matrix59, в которой требуется изменить порядок следования строк на противоположный. Однако такой вариант решения уже обсуждался в п. 7.2 в связи с инвертированием треугольных матриц, поэтому здесь мы рассмотрим другую, более сложную задачу на преобразование строк — **Matrix78**, в которой требуется отсортировать строки исходной вещественной матрицы в порядке убывания их минимальных элементов.

В данном случае требуется обрабатывать строки, а не столбцы матрицы, поэтому мы можем использовать стандартные средства, предусмотренные в задачнике Programming Taskbook для ввода и вывода двумерных структур. Так как нам не требуется вставлять или удалять строки, достаточно использовать структуру array of array of real.

Для сортировки элементов созданного массива (т. е. строк матрицы) удобно использовать запрос OrderByDescending, указав в его параметре (лямбда-выражении) *ключ* сортировки:

```
Task('Matrix78'); // Вариант 1
var m, n: integer;
var a: array of array of real;
ReadMatr(m, n, a);
a := a.OrderByDescending(row -> row.Min).ToArray;
WriteMatr(a);
```

Поскольку результатом сортировки является последовательность, а не массив, мы применяем к ней запрос ToArray, позволяющий записать результат в исходный массив a.

Мы видим, что и в данном случае использование подходящей структуры для хранения исходной матрицы дало возможность реализовать алгоритм решения, состоящий фактически из одной строки.

При анализе полученного решения может возникнуть вопрос о его эффективности. Дело в том, что лямбда-выражение для определения ключа представляет собой алгоритм нахождения минимального элемента, который работает достаточно долго, перебирая все элементы обрабатываемой строки. Если в процессе сортировки ключи для строк *каждый раз будут вычисляться заново* с применением запроса Min, то это серьезным образом скажется на быстродействии алгоритма, ухудшив его. С другой стороны, если программисты, реализовавшие запрос OrderByDescending, использовали *вспомогательную структуру для хранения набора ключей*, предварительно (до сортировки) заполнив ее с помощью указанного лямбда-выражения и затем, в процессе сортировки, используя значения из этой структуры, не перевычисляя их каждый раз заново, то алгоритм будет выполняться эффективно, независимо от сложности вычисления ключей (поскольку каждый ключ будет вычисляться *единственный раз*).

Проверить эффективность алгоритма OrderByDescending можно следующим простым способом: реализовать свой эффективный вариант и сравнить их быстродействие. Попутно мы вспомним полезный прием обработки последовательностей, ранее использовавшийся в п. 1.2 и 1.3.

Итак, реализуем вариант решения задачи Matrix78, в котором по-прежнему используется запрос OrderByDescending, но дополнительно позаботимся о том, чтобы ключи сортировки в лямбда-выражении вычислялись максимально быстро. Для этого надо каким-либо образом сохранить минимальные элементы строк *вместе* с самими строками. Простейшим способом такого сохранения является использование *кортежа* [1, п. 1.4]. Объединим в кортеж два значения: ссылку на строку, хранящуюся в исходном массиве строк, и минимальное значение элементов этой строки. Это не потребует существенных расходов памяти, но позволит сразу получать значение минимального элемента строки без его повторного вычисления. Останется выполнить сортировку полученной последовательности кортежей по их второму полю, а затем еще одним запросом Select создать отсортированную последовательность первых полей — ссылок на строки матрицы. Получаем следующий вариант алгоритма:

```
Task('Matrix78'); // Вариант 2
var m, n: integer;
var a: array of array of real;
ReadMatr(m, n, a);
a := a.Select(row -> (row, row.Min))
  .OrderByDescending(e -> e[1]).Select(e -> e[0]).ToArray;
WriteMatr(a);
```

Разумеется, в данном варианте, по сравнению с первым вариантом, используется больше запросов: один дополнительный запрос для формирования последовательности кортежей, другой — для извлечения из этой последовательности первого поля. Но теперь в запросе сортировки ключи будут вычисляться быстро независимо от внутренней реализации этого запроса.

Для сравнения эффективности полученных вариантов надо обработать с их помощью большие матрицы. Будем использовать квадратные матрицы порядка size, генерируя их с помощью выражения ArrGen(size, i -> ArrRandomReal(size)). Таким образом, каждая строка матрицы будет содержать вещественные числа в диапазоне от 0 до 10 (см. описание метода ArrRandomReal в [1, п. 3.3]). Время работы алгоритмов 1 и 2 в миллисекундах для матриц порядка 5000, 10000 и 20000 приведено в таблице 7.2.

Таблица 7.2. *Быстродействие вариантов решения задачи Matrix78*

Размер size	Вариант 1	Вариант 2
5000	247	239

10000	955	958
20000	3733	3755

Полученные результаты свидетельствуют о том, что в запросах сортировки использована эффективная реализация, при которой ключи сортировки вычисляются один раз. Таким образом, необходимости в применении кортежей *при решении данной задачи* нет. Однако указанный прием добавления «сопутствующих» данных к последовательности может оказаться полезным во многих ситуациях. В частности, иногда бывает удобно добавить к элементам последовательности их *индексы*; для этого даже предусмотрен специальный запрос проецирования Numerate, описанный в [1, п. 4.3] и использованный нами при решении задач из группы Minmax (см. п. 1.2, 1.3).

Если в процессе преобразования матрицы приходится добавлять или удалять ее строки или столбцы, то наиболее подходящей структурой будет список списков, в котором предусмотрены специальные методы для выполнения подобных действий. В качестве примера рассмотрим задачу **Matrix70**, в которой требуется продублировать строку матрицы, содержащую ее максимальный элемент.

Замечание. Может возникнуть вопрос: что делать, если в матрице имеется *несколько* максимальных элементов, расположенных в различных строках? Ответ будет таким: для матриц с вещественными элементами, предлагаемых задачником, такая ситуация невозможна. В главе 1 уже отмечалось, что во всех задачах, в которых предлагаются для обработки наборы *вещественных* чисел, следует предполагать, что эти числа являются *различными*. Все исходные данные задачник генерирует с помощью датчика случайных чисел, а вероятность получить в этой ситуации два одинаковых вещественных числа практически является нулевой.

Для вставки новой строки в список строк предусмотрен метод Insert (см. п. 2.2). Необходимо лишь определить индекс строки с максимальным элементом. Среди запросов агрегирования [1, п. 4.1] отсутствует запрос, позволяющий найти индекс минимума/максимума, однако имеется запрос MaxBy, позволяющий определить элемент последовательности, имеющий максимальный *ключ*. Воспользуемся этим запросом, а также приемом, основанным на применении последовательности *кортежей*:

```
Task('Matrix70'); // Вариант 1
var m, n: integer;
var a: List<List<real>>;
ReadMatr(m, n, a);
var k := a.Select((row, i) -> (row.Max, i)).MaxBy(e -> e[0])[1];
a.Insert(k, a[k]);
WriteMatr(a);
```

Используя запрос Select с лямбда-выражением, имеющим параметры row и i (элемент последовательности и его индекс), мы формируем новую последовательность кортежей, содержащую максимальный элемент каждой строки и ее индекс. Затем мы ищем кортеж e, максимальный по первому полю e[0] (т. е. кортеж с максимальным значением среди максимальных элементов строк), и сохраняем его второе поле (т. е. индекс строки с найденным максимальным значением) в переменной k.

После этого остается выполнить вставку строки a[k] в позицию k. В результате в позициях k и k + 1 будут размещены строки с одинаковыми элементами. Заметим, что при описанной вставке изменятся только *ссылки* на строки (все ссылки с индексами, большими k, будут сдвинуты на одну позицию вправо). Никаких изменений в размещении значений *элементов строк* в памяти не произойдет, т. е. требуемое преобразование матрицы будет выполнено быстро и эффективно. Более того, вставленная ссылка будет указывать *на ту же строку*, что и исходная ссылка a[k]. Это означает, что если мы изменим один из элементов вида a[k][j], то одновременно изменится и соответствующий элемент (k + 1)-й строки: a[k + 1][j]. Для нашей задачи указанная особенность не является существенной. Если же такое поведение полученной матрицы нежелательно, то достаточно изменить второй параметр метода Insert следующим образом:

```
a.Insert(k, a[k].GetRange(0, n));
```

Теперь мы передаем в метод Insert ссылку не на k-ю строку, а на ее *копию*, созданную методом GetRange (см. п. 2.3). При этом в памяти будут размещены *два списка с одинаковыми элементами*, и изменение элементов одного из этих списков не будет влиять на элементы другого.

Завершая обсуждение задачи Matrix70, приведем еще один вариант нахождения индекса строки с максимальным элементом:

```
Task('Matrix70'); // Вариант 2
var m, n: integer;
var a: List<List<real>>;
ReadMatr(m, n, a);
var k := a.Select(row -> row.Max).ToArray.IndexMax;
a.Insert(k, a[k]);
WriteMatr(a);
```

Здесь мы обошлись без применения последовательности кортежей, зато нам пришлось преобразовать последовательность максимальных элементов строк в массив (запросом ToArray), чтобы вызвать для этого массива метод IndexMax, возвращающий индекс максимального элемента [1, п. 5.3]. Данный вариант решения немного короче предыдущего, однако требует создания дополнительной структуры в памяти.

Если проанализировать быстродействие этих вариантов нахождения номера строки с максимальным элементом, то можно убедиться в том, что они выполняются практически с одинаковой скоростью (см. таблицу 7.3, в которой указывается порядок квадратной матрицы size и время, требуемое для нахождения номера строки с максимальным элементом).

Таблица 7.3. *Быстродействие вариантов решения задачи Matrix70*

Размер size	Вариант 1	Вариант 2
5000	303	283
10000	1049	1068
20000	4335	4317

Глава 8. Дополнение. Статические методы класса Array

В стандартной библиотеке платформы .NET для работы с массивами (любой размерности) предусмотрен класс Array. Любые *динамические* массивы, создаваемые в программах PascalABC.NET, имеют тип Array и поэтому могут использовать все свойства и методы данного типа, в частности, свойства Length [1, п. 3.2] и Rank (см. п. 6.1) и методы CopyTo [1, п. 5.1] и GetLength (см. п. 6.1).

По историческим причинам большинство методов класса Array, связанных с анализом и преобразованием массивов, оформлено в виде *статических* методов этого класса. Статический метод должен вызываться не для конкретного объекта (в нашем случае некоторого массива a), а для всего класса Array; при этом обрабатываемый объект (массив a) обычно указывается в виде *первого параметра* статического метода. Например, для поиска в массиве a индекса первого элемента со значением x в классе Array предусмотрен статический метод IndexOf, вызывать который надо следующим образом: Array.IndexOf(a, x). Примером статического метода, преобразующего массив, является метод Clear, вызов которого имеет вид Array.Clear(a, start, count) и обеспечивает обнуление count элементов массива a, начиная с элемента с индексом start.

При вызове статических методов класса Array в программе на языке PascalABC.NET необходимо указывать символ & перед именем Array. Это связано с тем, что слово «array» является зарезервированным словом языка PascalABC.NET и поэтому интерпретируется в программах особым образом (как начало описания массива). Кроме того, в PascalABC.NET не учитывается регистр символов. Символ & можно рассматривать как дополнительное «пояснение» компилятору, которое указывает ему, что в данном случае слово «array» не следует интерпретировать как зарезервированное слово языка. Итак, в программах на PascalABC.NET приведенные выше вызовы статических методов класса Array должны оформляться следующим образом: &Array.IndexOf(a, x) и &Array.Clear(a, start, count).

В стандартной библиотеке PascalABC.NET для многих статических методов класса Array реализованы их нестатические, т. е. «экземплярные» аналоги с теми же именами (методы называются *экземплярными*, если они

применяются к некоторому *объекту*, т. е. экземпляру класса; если говорится о методе некоторого класса, то имеется в виду экземплярный метод). Экземплярными аналогами удобнее пользоваться; к тому же в других классах библиотеки .NET, связанных с хранением данных (и рассмотренных нами в главах 2–5), все методы для их анализа и обработки реализованы именно в виде экземплярных методов. Например, организовать поиск элемента со значением x в массиве a в программе на PascalABC.NET можно с помощью (экземплярного) метода a.IndexOf(x) (описанного в [1, п. 5.3]), и аналогичный метод имеется у класса List<T> (см. п. 2.3).

Однако для некоторых статических методов класса Array их экземплярные аналоги в библиотеке PascalABC.NET не предусмотрены. Примером такого метода является упомянутый выше метод Clear, позволяющий обнулить требуемый диапазон элементов массива. Подобные методы используются достаточно редко, но в некоторых случаях могут оказаться полезными. Кроме того, достоинством статических методов класса Array является то, что они могут применяться в программах на любых языках платформы .NET (например, на языке C# или VB.NET), тогда как их экземплярные реализации для массивов доступны только в PascalABC.NET.

В данном пункте описываются все статические методы класса Array, включая все их перегруженные варианты (за исключением вариантов методов BinarySearch и Sort с параметром типа IComparer, позволяющим использовать при поиске и сортировке нестандартный способ сравнения элементов). Если метод может вызываться только для одномерного массива, то соответствующий параметр описывается как array of T. Если метод корректно обрабатывает не только одномерные, но и многомерные массивы, то при описании метода тип массива описывается с применением следующего условного обозначения: array [...] of T. Таких методов очень немного: это Clear, Copy и ConstrainedCopy.

8.1. Поиск

Вначале приведем связанные с поиском статические методы класса Array, которые имеют экземплярные аналоги (см. их описание в [1, п. 5.3]):

Поиск	Статические методы Array
BinarySearch(a: array of T; **[**start, count: integer;**]** value: T): integer	
Find(a: array of T; pred: T -> boolean): T	
FindLast(a: array of T; pred: T -> boolean): T	
FindAll(a: array of T; pred: T -> boolean): array of T	
FindIndex(a: array of T; **[**start: integer; **[**count: integer;**]]** pred: T -> boolean): integer	

FindLastIndex(a: array of T; **[**start: integer; **[**count: integer;**]]**
pred: T -> boolean): integer

IndexOf(a: array of T; value: T**[**; start: integer**[**;
count: integer**]]**): integer

LastIndexOf(a: array of T; value: T**[**; start: integer**[**;
count: integer**]]**): integer

Следует отметить, что у всех статических методов, связанных с поиском *индекса* элемента (FindIndex, FindLastIndex, IndexOf, LastIndexOf), имеется перегруженный вариант, отсутствующий у экземплярных методов, в котором можно указать не только индекс start, начиная с которого будет выполняться поиск, но и количество count элементов, для которых этот поиск будет проведен.

Дополнительные параметры предусмотрены и у метода BinarySearch, выполняющего двоичный поиск: это start и count, ограничивающие диапазон поиска.

Имеются также методы-*квантификаторы*, отсутствующие среди экземплярных методов массива, но имеющиеся у класса List<T> (см. их описание в п. 2.3):

Квантификаторы Статические методы Array

Exists(a: array of T; pred: T -> boolean): boolean

TrueForAll(a: array of T; pred: T -> boolean): boolean

Заметим, что их аналоги есть также среди запросов для последовательностей — см. [1, п. 4.1].

Все методы поиска, перечисленные в данном пункте, можно вызывать только для *одномерных* массивов.

8.2. Преобразование исходного массива

Изменение размера Статический метод Array

Resize(var a: array of T; newLength: integer)

Этот метод является аналогом процедуры SetLength [1, п. 5.2]; он изменяет размер массива a на newLength; при этом сохраняются значения всех оставшихся элементов, а при увеличении размера новые элементы получают нулевые значения. Это единственный статический метод класса Array, в котором первый параметр является одновременно входным и выходным (обратите внимание на модификатор var).

В отличие от процедуры SetLength метод Resize может использоваться только для *одномерных* массивов.

Обнуление элементов Статический метод Array

Clear(a: array […] of T; start, count: integer)

Метод Clear записывает нулевые значения соответствующего типа в count элементов массива a, начиная с элемента с индексом start. Этот метод может использоваться и для *многомерных* массивов; в этом случае многомерный массив интерпретируется как одномерный, причем перебор его элементов выполняется так, чтобы быстрее изменялись *правые* индексы элементов массива a (в частности, двумерные массивы-матрицы перебираются *по строкам*). Среди экземплярных методов массивов аналог метода Clear отсутствует.

Инвертирование Статический метод Array

Reverse(a: array of T**[**; start, count: integer**]**)

Данный метод *инвертирует* (т. е. меняет порядок элементов на противоположный) весь массив a или, при указании параметров start и count, его часть (count элементов, начиная с элемента с индексом start). Метод Reverse является полным аналогом *процедуры* Reverse [1, п. 3.7, 5.4], которая тоже имеет два перегруженных варианта с такими же параметрами. Как и процедура, метод Reverse может вызываться только для одномерных массивов.

Сортировка Статический метод Array

Sort(a: array of T; comp: (T, T) -> integer)

Sort(a: array of T**[**; start, count: integer**]**)

Sort(keys: array of TKey; a: array of T**[**; start, count: integer**]**)

Статический метод Sort класса Array, выполняющий сортировку массива a, имеет гораздо больше перегруженных вариантов, чем одноименный экземплярный метод (описанный в [1, п. 5.4]). Помимо вариантов с единственным параметром a и двумя параметрами a и comp, имеющих экземплярные аналоги, предусмотрены варианты с параметрами start, count и параметром keys (а также не приведенные здесь варианты с параметром типа IComparer), причем параметры могут комбинироваться любым способом.

При наличии параметров start и count сортируются только count элементов массива, начиная с элемента с индексом start.

При указании параметра-массива keys он выступает в роли *массива ключей* сортировки (этот параметр должен иметь тот же размер, что и массив a). Метод Sort выполняет сортировку массива keys и при этом изменяет порядок элементов массива a в соответствии с порядком элементов в *отсортированном* массиве keys. Следует заметить, что взаимное расположение элементов массива a, соответствующих *одинаковым* ключам из массива keys, в ходе сортировки может измениться (т. е. алгоритм сортировки массива a является в данном случае *неустойчивым*).

Все варианты статического метода Sort, как и аналогичные варианты экземплярного метода, можно использовать только для одномерных массивов.

8.3. Копирование и проецирование

Копирование Статические методы Array

Copy(src: array [...] of T; srcStart: integer; dst: array [...] of T;
dstStart, count: integer)
Copy(src: array [...] of T; dst: array [...] of T; count: integer)
ConstrainedCopy(src: array [...] of T; srcStart: integer;
dst: array [...] of T; dstStart, count: integer)

Метод Copy копирует count элементов из массива src в массив dst. Если указаны параметры srcStart и dstStart, то элементы с массиве src берутся, начиная с индекса srcStart и записываются в массив dst, начиная с индекса dstStart. Необходимо, чтобы в массиве src существовали элементы с указанными индексами, а в массиве dst было достаточно места для записи копируемых элементов; в противном случае возбуждается исключение. Требуется также, чтобы массивы src и dst имели одинаковое число измерений, причем допускаются и многомерные массивы. В случае многомерных массивов перебор производится так же, как и для метода Clear (см. п. 8.2).

Заметим, что экземплярные методы Copy для массива отсутствуют, однако имеется аналогичный метод CopyTo, вызов которого имеет вид src.CopyTo(dst, dstStart) [1, п. 5.1]. Метод CopyTo имеет больше ограничений по сравнению со статическим методом Copy, так как в нем нельзя указать ни количество копируемых элементов, ни начальную позицию в массиве-источнике src. Кроме того, метод CopyTo можно использовать только для одномерных массивов.

Напомним, что в PascalABC.NET имеется обычная функция (не метод) Copy(a), возвращающая копию массива a (см. [1, п. 5.1] и п. 6.5); в качестве параметра этой функции можно указывать массивы любой размерности.

Статический метод ConstrainedCopy выполняет те же действия по копированию, что и метод Copy с таким же набором параметров. Единственная его особенность состоит в том, что если при копировании некоторого элемента будет возбуждено исключение, то метод ConstrainedCopy *восстановит исходное содержимое массива-приемника* dst (разумеется, для того чтобы данный эффект мог проявиться в программе, необходимо, чтобы возбужденное исключение не привело к ее аварийному завершению). Подчеркнем, что речь идет только об исключениях, возникающих при *уже запущенном процессе копирования* (обычно они связаны с несоответствием типов и возникают *крайне редко*). Перед началом процесса копирования и метод Copy, и метод ConstrainedCopy проверяют допустимость параметров srcStart, dstStart, count, и если в массиве-приемнике dst недостаточно места или в массиве-источнике src недостаточно данных для копирования, то исключение возбуждается *немедленно*; таким образом, при подобных видах ошибок *содержимое массива* dst *не изменится и в случае использования*

обычного метода Copy. Это означает, что метод ConstrainedCopy имеет очень незначительные преимущества перед методом Copy, и следовательно, имеет весьма ограниченную область применения.

Проецирование Статический метод Array

ConvertAll(a: array of T; conv: T -> TRes): array of TRes

Метод ConvertAll может считаться методом *проецирования*, поскольку он подвергает преобразованию conv каждый элемент массива a и возвращает полученный массив преобразованных элементов (возможно, имеющих другой тип). Этот метод работает аналогично одноименному экземплярному методу, рассмотренному в [1, п. 5.4]. Он, как и его экземплярный аналог, могут применяться только к *одномерным* массивам.

Действия над элементами Статический метод Array

ForEach(a: array of T; action: T -> ())

Метод ForEach позволяет применить к каждому элементу массива a некоторое *действие*, оформленное в виде лямбда-выражения action, не возвращающего никакого значения. Он является аналогом одноименного запроса [1, п. 4.6], доступного для любых коллекций. Метод ForEach может вызываться только для *одномерных* массивов.

8.4. Использование статических методов класса Array при решении задач

Чтобы проиллюстрировать применение статических методов класса Array, вернемся к ранее рассмотренной задаче **Matrix78** (см. п. 7.5), в которой требуется отсортировать строки исходной матрицы по убыванию их минимальных элементов. Это типичная задача о сортировке по ключу, поэтому ее можно решить, используя вариант статического метода Sort с параметром keys — массивом ключей. Для генерации массива ключей можно использовать другой статический метод — ConvertAll (хотя, разумеется, вместо него можно использовать и одноименный экземплярный метод). Получаем следующий вариант решения:

```
Task('Matrix78'); // Вариант 3
var m, n: integer;
var a: array of array of real;
ReadMatr(m, n, a);
&Array.Sort(&Array.ConvertAll(a, row -> -row.Min), a);
WriteMatr(a);
```

Обратите внимание на символы &, которые необходимо указывать перед именем класса Array, чтобы компилятор не рассматривал это имя как зарезервированное слово языка Паскаль. Поскольку метод Sort всегда выполняет сортировку по возрастанию, а нам требуется выполнить сортиров-

ку по убыванию, мы использовали простой прием: при формировании ключей *изменили их знак на противоположный*.

Полученный вариант сравним по краткости с первым вариантом, использующим запрос OrderByDescending. Более того, он выполняет сортировку непосредственно в массиве a, не создавая новый отсортированный массив. С другой стороны, в варианте 3 нам приходится создавать дополнительный массив ключей. Таким образом, относительно расходов памяти эти варианты примерно равноценны. Они также не отличаются и по быстродействию, что доказывает численный эксперимент, в котором сортировались матрицы большого размера. Ниже приводится дополненный вариант таблицы 7.2 — таблица 8.1, в которой к измерениям времени работы алгоритмов 1 и 2 добавлены результаты для алгоритма 3.

Таблица 8.1. *Быстродействие вариантов решения задачи Matrix78*

Размер size	Вариант 1	Вариант 2	Вариант 3	Вариант 4
5000	247	239	244	7576
10000	955	958	942	31189
20000	3733	3755	3703	138028

В столбце «Вариант 4» этой же таблицы приведены результаты для следующего правильного, но *крайне неэффективного по быстродействию* алгоритма:

```
Task('Matrix78'); // Вариант 4
var m, n: integer;
var a: array of array of real;
ReadMatr(m, n, a);
&Array.Sort(a, (row1, row2) -> Sign(row2.Min - row1.Min));
WriteMatr(a);
```

В этом варианте указывается лямбда-выражение, позволяющее сравнивать две строки матрицы (аналогичный вариант имеется также и у экземплярного метода Sort для массивов — см. [1, п. 5.4]). Функция Sign(x) возвращает –1 для отрицательных чисел x, 0 для нулевых x и 1 для положительных x. Она использована в лямбда-выражении, чтобы преобразовать вещественную разность в целое число, сохранив информацию о знаке разности. Благодаря тому, что из row2.Min вычитается row1.Min, мы обеспечиваем сортировку *по убыванию*.

Причина неэффективности варианта 4 должна быть понятна из обсуждения вариантов 1 и 2 (см. п. 7.5). Первые три варианта решения оказались эффективными, поскольку набор ключей для сортировки вычислялся в них *единственный раз*. Для варианта 4 ситуация иная: при *каждом* сравнении строк матрицы row1 и row2 вызывается лямбда-выражение для их сравнения, в котором *каждый раз заново* находятся минимальные элементы этих строк, что существенно замедляет работу алгоритма.

Литература

1. *Абрамян М. Э.* Структуры данных в PascalABC.NET. Выпуск 1. Массивы и последовательности. Запросы. — Ростов н/Д : Изд-во ЮФУ, 2016. — 119 с.
2. *Абрамян М. Э.* Практикум по программированию на языке Паскаль: Массивы, строки, файлы, рекурсия, линейные динамические структуры, бинарные деревья. — 7-е изд., перераб. и доп. — Ростов н/Д : Изд-во ЮФУ, 2010. — 276 с.
3. *Абрамян М. Э.* Платформа .NET: Основные типы стандартной библиотеки. Работа с массивами, строками, файлами. Объекты, интерфейсы, обобщения. Технология LINQ. — Ростов н/Д : Изд-во ЮФУ, 2014. — 218 с.

Указатель

Чтобы упростить использование указателя, методы и свойства в нем размещены в двух местах: в алфавитном порядке их имен и в группах, связанных с различными структурами данных (Array, Dictionary, HashSet и SortedSet, KeyValuePair, LinkedList, LinkedListNode, List, Queue, set of T, Stack, многомерный динамический массив, невыровненный массив (массив массивов), список списков). Сгруппированы также подпрограммы, описанные в модуле PT4. В группе «Решения задач» перечислены все задачи из задачника Programming Taskbook, рассмотренные в данной книге, с указанием страниц, на которых приведены различные варианты их решения.

Содержание