Нужно быть очень терпеливым,

чтобы научиться терпению.

Е. Лец

Нельзя говорить нельзя.

Д. Араго

###### Введение

Полезность, важность и необходимость рекурсии, как одного из концептуальных методов решения практических задач подчеркивалась многими мэтрами информатики. Сошлемся лишь на двух лауреатов премии Тьюринга: американского специалиста по системному программированию Д.Кнута и английского теоретика информатики Ч.Хоара.

Д.Кнут широко использовал рекурсию при изложении материала в ставшем уже классическим его трехтомном выпуске “Искусство программирования для ЭВМ” [1-3]. Кроме того, он предполагал продолжить издание книг этой серии и в четвертом томе одну из двух глав назвать “Рекурсия”, полностью посвятив её рекурсивным методам решения задач [1, стр.11]. К великому сожалению, тома с 4 по 7 до сих пор не вышли. Однако в настоящее время появилась надежда, что в ближайшие годы (1999г.-2004г.) они будут дописаны и опубликованы [9].

Ч.Хоару принадлежат следующие слова “Следует отдать должное гению разработчиков Алгола-60 за то, что они включили в свой язык рекурсию и дали мне тем самым возможность весьма элегантно описать мое изобретение (речь идет о так называемой быстрой сортировке – Quick Sort). Сделать возможным изящное выражение хороших мыслей – я считал это наивысшей целью проекта языка программирования” [4, стр. 176]. К этому лишь следует добавить, что, на сегодняшний день, практически все действующие языки программирования поддерживают рекурсию.

В данном пособии дается неформальное понятие рекурсии, рассказывается об общей схеме решения задач с помощью рекурсии и приведены рекурсивные алгоритмы решения весьма разнообразных по содержанию и степени сложности задач.

###### 1.Что такое рекурсия?

Понятие рекурсии достаточно просто для понимания и не связано со знанием какого-либо определенного формализма или специальной нотации. В общем случае на рекурсию следует смотреть как на введение в определение объекта ссылку на сам объект или, более определенно, как на прием сведения решения некоторой задачи к решению “более простой” задачи такого же класса. В программировании это выражается в построении программ (процедур и функций), которые при выполнении обращаются сами к себе непосредственно или через цепочку других программ. Кажущаяся при этих самовызовах или последовательных циклических вызовах видимость порочного круга (circulus vitiosus – лат.) не более чем иллюзия. Во многих конкретных случаях простыми рассуждениями путем отслеживания значений одной или нескольких управляющих величин удается провести доказательство завершимости вычислений за конечное число шагов.

Функция называется рекурсивной, если в её определении содержится вызов этой же функции. Различают простую рекурсию, когда текст программы функции F напрямую содержит вызов F, и косвенную рекурсию, когда F обращается к иным функциям, которые содержат вызов F. Поэтому, по тексту программы рекурсивность не всегда явно определима. Знание механизмов реализации рекурсии помогает эффективно её использовать. Что происходит, когда функция F выполняет рекурсивный вызов? Прежде всего, запоминается текущее состояние программы, необходимое для продолжения вычислений, когда управление снова вернется к ней. Затем F с новыми значениями аргументов начинает выполняться заново как бы с новым экземпляром программы. При следующем рекурсивном вызове F всё повторяется и т.д. до тех пор, пока очередной вызов F не приводит к какому-либо тривиальному случаю, разрешаемому без рекурсивных вызовов. Далее, в порядке, обратном тому, в котором запоминалась серия вызовов, производятся возвраты управления. В практических приложениях важно убедиться, что максимальная глубина рекурсивных вызовов не только конечна, но и достаточно мала. В противном случае не избежать переполнения стека – специально организованного участка памяти, где запоминаются отдельные состояния программы-функции.

В таблице 1.1 приведена общая схема решения задач с помощью рекурсии. Эта схема обращается сама к себе и поэтому, является примером рекурсивного объекта. Решение конкретной задачи рекурсивным методом распадается на несколько шагов, основными из которых являются четыре этапа: параметризация, выделение базы и возможных правил её модификации, декомпозиция и проведение отложенных вычислений. Первые три из них называют рекурсивной триадой. В таблице 1.1 триада выделена общей рамкой. Остановимся на указанных этапах подробнее.

**Параметризация задачи** заключается в выявлении совокупности исходных величин, определяющих постановку и решение задачи. Значения этих параметров или некоторых из них влияют на трудоемкость решения задачи. Иногда бывает полезно ввести в рассмотрение дополнительные параметры, напрямую с постановкой задачи не связанные, но помогающие организовать рекурсию.

**Выделение базы** – поиск одной или нескольких подзадач, которые могут быть решены непосредственно без рекурсивного вызова.

**Таблица 1.1.** Рекурсивная схема решения задач с помощью рекурсии

**Решение задач с помощью р е к у р с и и**

**С**

**о**

**з**

**д**

**а**

**н**

**и**

**е**

**д**

**о**

**к**

**у**

**м**

**е**

**н**

**т**

**а**

**ц**

**и**

**и**

Тестирование

программ

**Мотивация**

Программная

реализация

алгоритма

**Параметризация**

**задачи**

**Выделение**

**базы**

# Формирование

**правил пополнения базы**

**Декомпозиция**

Логическое

обоснование идей

этапов триады

**Коррекция**

**этапов**

**триады**

Триада

**Постановка задачи**

Анализ требований и определение

спецификаций

Подготовка

тестов

Подготовка

контрольных

примеров

Отладка

программ

Построение модели

Оценка глубины

**рекурсивных**

вызовов

Анализ

сложности

алгоритма

**Анализ результатов**

# В

**ы**

**ч**

**и**

**с**

**л**

**е**

**н**

**и**

**я**

Если база будет меняться в процессе вычислений, то должен быть указан алгоритм её изменения. Как правило, подобная динамическая база расширяется за счет получения решений промежуточных задач и облегчает выполнение процесса отложенных вычислений. Возможно и сужение рекурсивной базы.

**Декомпозиция общего случая** есть процесс последовательного разложения исходной задачи на серию более простых подзадач, аналогичных исходной задаче, каждая из которых обычно по тому или иному признаку более близка к тривиальному случаю, чем предыдущая. Декомпозиция предполагает наличие некоторых вычислений, предшествующих и способствующих переходам к более простым подзадачам. Их удобно называть предварительными вычислениями. Декомпозицию необходимо осуществлять так, чтобы несложно было доказать, что при любом допустимом наборе значений параметров, рано, или поздно, она приведет нас к одному из выделенных тривиальных случаев, то есть к базе.

**Проведение отложенных вычислений.** На последнем этапе, решая одну за другой полученные на этапе декомпозиции подзадачи в порядке обратном их получению, мы добираемся до решения исходной задачи. Этот этап непосредственно опирается на соответствующие предварительные вычисления (предвычисления).

Нелишне заметить, что некоторые преподаватели информатики в школах и вузах имеют стойкое предубеждение против рекурсии, неправомерно преувеличивают затраты ‘памяти-времени’ в рекурсивных алгоритмах и считают эти затраты весьма расточительными. Исходя из этой предпосылки, они и действуют, пропагандируя использование итерации даже в тех случаях, когда имеют дело по существу с рекурсивными алгоритмами или с данными, имеющими рекурсивную природу. Причины недостаточного внимания к рекурсии в текущем нормативном преподавании можно разделить на следующие.

**Исторические.** Устоявшиеся традиции преподавания математики и информатики и нерекурсивность начальных версий первых языков программирования высокого уровня Кобола и Фортрана. Тем не менее, стоит отметить, что многие известные авторы, ориентируясь в своих книгах, статьях и учебных пособиях тридцати-сорокалетней давности на Фортран, весьма широко использовали рекурсию в практике вычислений. При этом, нерекурсивность языка в каждом конкретном случае написания рекурсивного алгоритма требовала от них большой выдумки и изобретательности.

**Психологические.** Отсутствие диспозиционной и ситуационной мотиваций (побудительных причин) у большинства преподавателей и их неподготовленность как в школе, так и в вузе, к рекурсивным рассуждениям.

**Педагогические.** Консерватизм образовательной среды по отношению к содержанию предметной области;

**Методические.** Отсутствие устоявшейся рабочей терминологии и понятийного аппарата, а также полноценных и доступных методических разработок по рекурсивным методам решения задач.

**Технические.** Недостаточные ресурсы быстродействия и, в особенности, оперативной и дисковой памяти учебных компьютеров в недавнем прошлом, а зачастую и в настоящее время.

**Технологические.** Отсутствие средств отладки во многих языках программирования и полное отсутствие специализированных средств отладки и тестирования рекурсивных процедур и функций.

###### О терминологии

В предыдущем пункте, поясняя, что такое рекурсия, мы были вынуждены ввести в рассмотрение несколько специальных терминов. Этот ряд необходимо продолжить. К минимальному набору, требующих прочного усвоения студентами, понятий и терминов следуют отнести следующие смысловые единицы: рекурсия, рекурсивный алгоритм, прямая рекурсия, косвенная рекурсия, рекурсивные обращения, рекуррентные соотношения (возвратные последовательности), производящая функция, параметризация задачи, вспомогательные параметры рекурсии, рекурсивная база, индикаторы завершения рекурсивных вызовов, пространство параметров, полная рекурсивная траектория, рекурсивная траектория, глубина рекурсивных вызовов, декомпозиция, предварительные вычисления, отложенные вычисления, повторительная рекурсия, рекурсивная триада, рекурсивные вычисления, прямой и обратный ход рекурсии, рекурсивный стек, динамическая рекурсивная база, срез рекурсивных вычислений, формуляр, воплощение, рекурсограмма, рекурсивная машина обработки формуляров, рекурсивная тавтология, адаптивный рекурсивный алгоритм, визуальное мышление, рекурсивное мышление. С учетом пояснений некоторых из этих терминов, сделанных в предыдущем пункте, смысл большей части остальных терминов становиться интуитивно ясным. Тем не менее, дадим им короткие пояснения (неформальные определения). Это позволит в дальнейшем избегать неточностей или двусмысленностей при описании рекурсивных алгоритмов. Все эти краткие неформальные определения собраны в таблицу 2.1

**Таблица 2.1.** Понятия и термины, связанные с рекурсией

| **№** | **Понятие,**  **Термин** | **Неформальное определение,**  **пояснение** |
| --- | --- | --- |
|  | Рекурсия | 1. Введение в определение объекта ссылку на сам объект.  2. Прием сведения решения некоторой задачи к решению серии задач, подобных исходной.  3. Свойство алгоритмической системы на промежуточных этапах своего функционирования создавать другие системы, включая идентичные себе самой, и использовать результаты их функционирования в дальнейшей работе. При достаточно широкой трактовке понятия алгоритмической системы концепция рекурсивности отражает основные формы развития материи и является одним из важнейших методов познания. |
|  | Рекурсивный  алгоритм  (процедура,  функция) | 1. Алгоритм (функция, процедура) называется рекурсивным, если в его определении содержится прямой или косвенный вызов этого же алгоритма.  2. Рекурсивная функция − одно из математических уточнений интуитивного понятия вычислимой функции. |
|  | Прямая  рекурсия | Непосредственный вызов алгоритма (функции, процедуры) F из текста самого алгоритма F. |
|  | Косвенная  рекурсия | Циклическая последовательность вызовов нескольких алгоритмов (функций, процедур) F1, F2, … Fk друг друга: F1 вызывает F2, F2 вызывает F3, …, Fk вызывает F1 (k>1). |
|  | Рекурсивные  обращения  (рекурсивные  вызовы) | Прямая или косвенная рекурсия при рекурсивных вычислениях |
|  | Рекуррентное  соотношение  (рекуррентная  формула) | Формула вида an+p=F(an, an+1,…, an+p−1) (p≥1), позволяющая вычислять любой член бесконечной последовательности a1, a2,…, если заданы её первые p членов. Определяемая рекуррентной формулой последовательность называется возвратной. |
|  | Производящая  функция | Производящей функцией числовой бесконечной последовательности a1, a2,…, называют степенной ряд вида: , с вещественной или комплексной переменной z. |
|  | Параметризация  задачи | Выявление совокупности исходных величин, определяющих постановку и решение задачи. Значения этих параметров или некоторых из них влияют на трудоемкость решения задачи. |
|  | Вспомогательные  параметры  рекурсии | Параметры, напрямую с постановкой задачи не связанные, но помогающие изменить тип рекурсии или перейти к обобщенной задаче, где рекурсия проглядывается явно. |
|  | Рекурсивная база | Совокупность наборов значений параметров и соответствующих им решений задачи (или простых правил для получения этих решений). Выделение базы − один из основных этапов решения задачи с помощью рекурсии. |
|  | Индикаторы  завершения  рекурсивных  вызовов | Элементы постоянной или динамической рекурсивной базы. |
|  | Пространство  параметров | Пусть tk (k=1..n) параметры задачи (алгоритма, процедуры, функции), принимающие значения из некоторых множеств объектов Mk (k=1..n). Декартово произведение M множеств Mk (k=1..n) называется пространством параметров задачи. Таким образом, элементами M являются наборы (упорядоченные множества) объектов m1, m2, … mn, где mk∈Mk (k=1..n) вида: (m1, m2, … mn). Областью определения параметризованной задачи, является совокупность элементов пространства параметров, при которых она имеет решение. |
|  | Полная  рекурсивная  траектория | Пусть F(X), где X=(x1, x2,…xn) − рекурсивная функция, которую требуется вычислить в некоторой точке X0. Конечная последовательность аргументов F(X) вида: X0, X1, …Xn называется рекурсивной траекторией, если элементы Xk (k =1..n) − суть наборы параметров при последовательных рекурсивных вызовах, а Xn принадлежит базе рекурсии. |
|  | Рекурсивная  траектория | Любая начальная подпоследовательность полной рекурсивной траектории. |
|  | Глубина  рекурсивных  вызовов | Количество элементов полной рекурсивной траектории в пространстве параметров. |
|  | Декомпозиция  Предварительные  вычисления  (предвычисления) | Процесс последовательного разложения задачи на серию более простых подзадач, аналогичных исходной задаче, каждая из которых обычно по тому или иному признаку более близка к тривиальному случаю, чем предыдущая. Декомпозиция предполагает наличие некоторых вычислений, предшествующих и способствующих переходам к более простым подзадачам. Назовем их **предварительными вычислениями** или **предвычислениями**. Декомпозицию необходимо осуществлять так, чтобы несложно было доказать, что при любом допустимом наборе значений параметров, рано, или поздно, она приведет нас к одному из выделенных тривиальных случаев, то есть к задаче с набором параметров, являющемся индикатором завершения рекурсивных вызовов. |
|  | Отложенные  вычисления  Повторительная  рекурсия | Вычисления, проводимые после того, как рекурсивная траектория попала в базу, то есть стала полной. Возможно, что отложенные вычисления состоят лишь из серии передач значений и управления в порядке, обратном рекурсивным вызовам. В этом случае реальные отложенные вычисления отсутствуют, а соответствующая рекурсия называется повторительной. |
|  | Управляющие  параметры  рекурсии  (управляющий  параметр) | Параметры задачи, с помощью которых организуется её декомпозиция, обеспечивающая правила выполнения рекурсивных вызовов, а также предварительных и отложенных вычислений. |
|  | Рекурсивная  триада | Три основных этапа решения задач с помощью рекурсии: параметризация, выделение базы (или выделение начальной базы и правил её изменения), декомпозиция. |
|  | Рекурсивные  вычисления  Прямой и  обратный ход  вычислений | Вычисления, проводимые с помощью рекурсивных алгоритмов. Они состоят из двух стадий, называемых прямым ходом и обратным ходом. Первая из них соответствует совокупности всех предвычислений, реализуемых до входа рекурсивной траектории в базу, а вторая − совокупности отложенных вычислений, производимым после встречи с индикатором завершения рекурсивных вызовов. |
|  | Рекурсивный  стек | Область памяти, в которую заносятся значения всех локальных переменных алгоритма (программы) в момент рекурсивного обращения. Каждое такое обращение формирует один слой данных стека. При завершении вычислений по конкретному обращению α из стека считывается соответствующий ему слой данных, и локальные переменные восстанавливаются, снова принимая значения, которые они имели в момент обращения α. |
|  | Динамическая  рекурсивная база | Рекурсивная база, меняющаяся в процессе вычислений. Как правило, она расширяется за счет получения решений промежуточных задач и облегчает выполнение процесса отложенных вычислений. Возможно и сужение рекурсивной базы. |
|  | Срез рекурсивных  вычислений | При решении задачи каждое рекурсивное обращение, в том числе и начальный запуск вычислений, инициируют работу как бы со ‘своим экземпляром’ исходного алгоритма. Последовательность вычислений значений локальных и глобальных переменных, соответствующая одному конкретному ‘виртуальному экземпляру’ алгоритма и не включающая в себя вычисления по вызовам из данного экземпляра (но использующая их результаты!), называется срезом рекурсивных вычислений. |
|  | Формуляр | Специально разработанный расчетный бланк, в котором фиксируется протокол вычислений конкретного рекурсивного среза. Формуляр может быть задан таблицей, деревом Канторовича или иным способом. В нем должны указываться взаимосвязь шагов вычислений и, кроме того, предлагаться место для проведения вычислений. |
|  | Воплощение | Заполненный формуляр. Воплощение формируется для каждого рекурсивного среза на отдельном формуляре. Это же самое касается и всех вызовов нерекурсивных подпрограмм, для которых должны быть разработаны свои собственные формуляры. |
|  | Рекурсограмма | Последовательность воплощений, соответствующая последовательности рекурсивных вызовов. |
|  | Рекурсивная  машина  обработки  формуляров | Если правила заполнения формуляров при решении определенного круга задач с помощью рекурсии некоторым образом формализованы, то этот процесс может быть автоматизирован. В этом смысле можно говорить о виртуальной рекурсивной машине по заполнению формуляров. |
|  | Рекурсивная  тавтология | Прямое или косвенное обращение рекурсивной функции (алгоритма) к самой себе с набором значений параметров, с которого начиналось вычисление этой функции. |
|  | Адаптивный  рекурсивный  алгоритм | Алгоритм, который благодаря рекурсивности учитывает те или иные индивидуальные характеристики решаемой задачи из области своего определения. |
|  | Визуальное  мышление | 1. Способ решения интеллектуальных задач с опорой на внутренние визуальные образы.  2. Вид мышления, продуктом которого является порождение новых образов, создание новых визуальных форм, несущих определенную смысловую нагрузку. |
|  | Рекурсивное  мышление | 1. Способ решения прикладных задач с преимущественной опорой на рекурсивные алгоритмы.  2. Разновидность математического (диалектического, продуктивного) мышления, позволяющая видеть рекурсию там, где она на первый взгляд не просматривается. |

**Замечания.** В таблице описаны термины и понятия, связанные с рекурсией и используемые в информатике. Их оказалось чуть более 30. При этом многие понятия вводятся впервые. В то же время подобных слов и словосочетаний, активно используемых в математике порядка 300.

###### Корзина разнообразных задач

Как для конкретной задачи построить рекурсивный алгоритм её решения? − готовых рецептов не существует. Некоторые практические рекомендации на этот счет приведены в [6, стр. 144]. Однако лишь ознакомление с достаточным количеством учебных рекурсивных алгоритмов позволит выработать определенную интуицию в выборе тактики и стратегии поиска и обнаружения спасательной рекурсии в незнакомой обстановке и заложить фундамент для освоения, совершенствования и отработки техники рекурсивного программирования. Общие рекомендации здесь могли бы быть такими. Пытаясь искать рекурсивное решение какой-либо задачи, следует опираться на одну из предлагаемых ниже именованных схем.

* **Схема 1 − “увидеть”.** Увидеть непосредственную рекурсию в определении объекта. Во многих задачах условия не просто задают её постановку, но делают это рекурсивно. Отсюда и рекурсивные программы, являющиеся точной копией условий задачи. Смотри задачи ??? .
* **Схема 2 − “переформулировать”.** Часто в условиях задачи не только не проглядывается рекурсия, но и сама задача неявляется алгоритмически сформулированной. Иногда её простая перефразировка, а чаще построение математической модели позволяют вдруг обнаружить первоначально скрытую рекурсию. Смотри задачи ???.
* **Схема 3 − “обобщить (погрузить, вложить)”.** Если из постановки задачи рекурсию извлечь не удаётся, то за счет перехода к её некоторому обобщению иногда это сделать можно. При этом предполагается, что из решения обобщенной задачи без особого труда может быть получено решение исходной задачи. Как правило, переход к обобщенной задаче происходит за счет введения дополнительных параметров. В некоторых случаях рассматриваемая схема может быть использована для перехода от одного типа рекурсии к другому. Смотри задачи ???.
* **Схема 4 − “найти родственника”.** Иногда к исходной задаче удается найти одну или несколько вспомогательных родственных к ней задач так, что в совокупности, взаимно дополняя друг друга, они уже будут определять вполне просматриваемую косвенную рекурсию. Смотри задачи
* **Схема 5 − “обнаружить характеристическое свойство”.** Пусть совокупность всех или части условий задачи оформлена в виде некоторого предиката над наборами входных данных и возможных результатов. Такой предикат определяет некоторое характеристическое свойство задачи. Формальная запись предиката с одной стороны позволяет проводить независимую “экспертную” проверку правильности работы ранее разработанных алгоритмов решения данной задачи, а с другой стороны, может оказать существенную помощь для отыскания новых рекурсивных алгоритмов её решения. При этом иногда целесообразно преобразовать предикат, то есть переформулировать характеристическое свойство задачи так, чтобы из него можно было извлечь какой-либо иной алгоритм. В любом случае, следует помнить, что характеристические свойства не всегда определяют исходную задачу однозначно.
* **Схема 6** − “**перенести часть условий в проверку**”. Иногда при рассмотрении всех условий задачи рекурсия в явном виде сразу не обнаруживается, но удаление части условий приводит к новой вспомогательной задаче, рекурсивный алгоритм решения которой строится без особых затруднений. В этом случае, чтобы узнать, является ли полученный для новой задачи ответ (ответы) решением исходной задачи, необходимо проверить выполняются ли для него ранее удаленные условия или нет. Если решение задачи сводится к вычислению значения истинности некоторого предиката, непосредственно построенного из конъюнкции условий задачи на наборах входных данных, то описанная схема допускает возможность проверки выполнимости удаляемых условий как до использования рекурсивного алгоритма решения вспомогательной задачи, так и после этого. Смотри задачи ???.

Остановимся еще на одном важном моменте. Последовательность рекурсивных обращений за **конечное число шагов** обязательно должна приводить нас к одному из индикаторов завершения вычислений, расположенных в базе (см. табл. 3.1, где совокупность предварительных вычислений, соответствующая прямому ходу алгоритма ℜ, обозначена через f). В дальнейшем остается лишь провести отложенные вычисления. Этим рекурсивные вычисления по существу отличаются от метода последовательных приближений. Однако нельзя всегда рассчитывать на окончание рекурсивного алгоритма за конечное число шагов, как на нечто само собой разумеющееся. Иногда установление этого свойства для определенного подмножества значений пространства параметров требует значительных усилий в проведении подчас непростых рассуждений.

**Таблица 3.1.** Схематическое изображение последовательности рекур-

сивных обращений и формирования полной рекурсивной траектории

**Пространство параметров**

**Область определения**

**алгоритма ℜ**

**База**

**x1**

**x0**

**x2**

**x3**

**x4**

**x5**

**f**

**f**

**f**

**f**

**f**

В оставшейся части данного пункта рассматривается серия простых учебных демонстрационных задач, решения которых получаются с помощью рекурсивно определенных алгоритмов. Во многих случаях детально обсуждаются описанные выше схематические приемы поиска этих алгоритмов. В основном все программы-функции написаны на языке программирования вычислительной среды Mathcad. Часть программ написана на языке Object Pascal 5.0 системы объектного визуального программирования Delphi 5. Для некоторых задач предлагается несколько вариантов программ. Приводятся контрольные примеры. Заметим, что, ввиду разноплановости предложенных задач, многие из них могут служить отдельными темами, собирающими вокруг себя родственный содержательный материал по рекурсии для отработки техники рекурсивного программирования в рамках конкретного направления.

Результатом проработки материала данного пункта должно стать убеждение, что писать рекурсивные программы, как правило, несложно, а получаемые при этом тексты весьма компактны и, по причине отсутствия в них диких зарослей языковых украшательств, легко читаются. Нам представляется, что читатель вряд ли откажет себе в удовольствии написать собственные программы-функции решения многих из приведенных задач или их обобщений.

## Факториал

**Задача 1.** Составить программу-функцию вычисления факториала целого неотрицательного числа n.

**Решение.** Для целых неотрицательных чисел n факториал n обозначается через n! и определяется так:



В данном случае параметризация задачи осуществлена в её постановке. Остается лишь ввести более приемлемое для нас обозначение искомой функции. Пусть это будет facto(n). Тривиальные случаи, для которых задача решается без рекурсивных вызовов, также очевидны: facto(0)=1, facto(1)=1. Они и составляют базу рекурсии. Декомпозиция по параметру n реализуется по формуле: facto(n) = n⋅facto(n–1) (n = 1, 2, …). Поэтому вычисления facto(n) можно организовать так:



Контрольные примеры.



Понять процесс реализации рекурсивных вызовов, то есть декомпозицию, и возвратов управления при организации отложенных вычислений facto(n) можно из схемы рис. 3.1 (n = 3). Там около стрелок в круглых скобках жирными цифрами указаны номера последовательных шагов вычислений: (**1**), (**2**), (**3**) − декомпозиция; (**4**), (**5**), (**6**) − отложенные вычисления.



Рис. 3.1. Схематическое изображение рекурсивных вызовов и

отложенных вычислений при нахождении facto(3) = 3!.

**Замечание.** С помощью встроенной функции if() предложенный алгоритм удается записать еще короче. Это же касается и многих других примеров.



Для решения конкретной задачи иногда удается построить несколько различных рекурсивных алгоритмов. Например, функцию facto(n) можно было бы определить и так.



По сравнению с прежней реализацией facto() здесь количество рекурсивных вызовов уменьшается практически в 2 раза. В реальных ситуациях подобный фактор может оказаться решающим при выборе того или иного алгоритма.

Используем теперь для вычисления n! cформулированную выше схему 3 (обобщить). Если вместо исходной функции facto(n)=n!, ввести в рассмотрение функцию двух переменных fa(n, λ)=λ⋅n! (n=0,1,…), то получим равенства:

fa(n, λ)=λ⋅n⋅(n−1)⋅…⋅1=(λ⋅n)⋅(n−1)!=fa(λ⋅n)⋅(n−1)!,

fa(1, λ)=λ , fa(n,1)=n!.

Первое из этих соотношений может служить правилом декомпозиции, второе − определять рекурсивную базу, а третье − показывает, как вычислять n!. Соответствующая рекурсивная программа-функция могла бы выглядеть так:



В чем же отличие этой функции от первого варианта функции facto(n)? Дело в том, в facto(n) формирование n! реализуется при проведении отложенных вычислений, в то же время нахождение fa(n, λ) проводится вообще без отложенных вычислений. Особенно наглядно это видно на рисунках 3.1 и 3.2. На шагах 4, 5 и 6, отмеченных на рис. 3.2 жирными цифрами в круглых скобках, происходит лишь передача значений. Иными словами здесь мы имеем повторительную рекурсию.



Рис. 3.2. Схематическое изображение рекурсивных вызовов

при нахождении fa(3,λ) = λ⋅3!.

Еще один вариант вычисления n! можно реализовать с помощью рассмотренной ниже в задаче 20 функции Кадью.

## Сложный процент

**Задача 2.** Вкладчик положил в сбербанк сумму в sum единиц под p процентов за один период времени (год, месяц, неделя и т.д.). Составить программу-функцию возвращающую величину вклада по истечении n периодов времени (n = 1, 2,).

**Решение.** Пусть invest(sum,p,n) искомая функция. Для данной задачи вычисления значений invest() можно проводить по формуле

invest(sum,p,n) = sum⋅(1+p/100)n .

Однако, в учебных целях, нас интересует рекурсивный вариант алгоритма решения задачи. Рекурсию будем осуществлять по параметру n. Тривиальный случай очевиден. Если вклад положен на хранение и взят сразу, то есть до истечения первого периода времени начисления процентов, то возврату подлежит начальная сумма вклада − sum. Далее, декомпозиция может быть реализована исходя из следующего факта. Положить некоторую сумму в банк на n периодов – это то же самое, что положить эту сумму на n – 1 периодов и затем полученную сумму положить на 1 период. Соответствующий вариант программы-функции решения задачи выглядит так:



Контрольные примеры.



Схема рекурсивных вызовов здесь такая же, как при вычислении значений функции facto(n). Нетрудно видеть, что общее количество рекурсивных вызовов при вычислении invest(sum,p,n) равно n. При необходимости можно было бы уменьшить это значение до log2(n).

## Степень числа

**Задача 3.** Пусть a − вещественное число отличное от нуля и n − целое неотрицательное число. Составить программу-функцию возвращающую величину an.

**Решение.** Приведенная ниже функция power(a,n) дает решение задачи за n рекурсивных вызовов:



Уменьшить количество вызовов можно так. Организуем декомпозицию иначе, представив величину an в виде:



Отсюда сразу же получаем алгоритм вычисления an, требующий не более log2(n) рекурсивных вызовов. Реализуется он функцией pow(a,n):



Сумма элементов массива

**Задача 4.** Составить программу-функцию, возвращающую сумму S компонентов вектора v=(a0,a1,…,an−1)T: S= a0+a1+…+an−1, где n≥1 и ap (p=0..n−1) − вещественные или комплексные числа.

**Решение.** Определение суммы n слагаемых в виде:

S= a0+a1+…+an−1=(a0 +a1+…+an−2)+an−1

рекурсивно по своей сути. Сумма n слагаемых есть сумма первых (n−1)-го слагаемого плюс сумма последнего слагаемого. Этот факт и положен в основу определения функции summa(v), где v=(a0,a1,…,an−1)T.



## Произведение элементов массива

**Задача 5.** Составить программу-функцию, возвращающую произведение P компонентов вектора v=(a0,a1,…,an−1)T: P= a0⋅a1⋅…⋅an−1, где n≥1 и ap (p=0..n−1) − вещественные или комплексные числа.

**Решение.** Определение произведения n сомножителей в виде:

P= a0⋅a1⋅…⋅an−1= (a0⋅a1⋅…⋅an−2)⋅an−1 ,

как и соответствующее определение суммы, рекурсивно по своей сути. Произведение n сомножителей есть произведение первых (n−1)-го сомножителей умноженное на последний сомножитель. Отсюда и определение функции product(v), где v=(a0,a1,…,an−1)T.



## Числа Фибоначчи

**Задача 6.** Составить программу-функцию вычисления n−го числа Фибоначчи, исходя из рекуррентного определения этих чисел:

f(0)=f(1)=1, f(n)=f(n−1)+f(n−2) (n=2,3,…).(1)

**Решение.** Наличие рекуррентного соотношения вида (1) сразу же определяет и базу рекурсии, и способ декомпозиции. Программа-функция fib(n) написана строго в соответствии с определением (1).



Контрольные примеры.



Функция fib(n) вряд ли подходит для вычисления чисел Фибоначчи при больших n. И происходит это потому, что в данном случае с ростом n дерево рекурсивных вызовов очень быстро разрастается. На рис. 3.3 представлена схема рекурсивных вызовов для fib(5) (имя функции обозначено через f).

**f(5)**

**f(4)** **f(3)**

**f(3)** **f(2)** **f(2)** f(1)

**f(2)** f(1) f(1) f(0) f(1) f(0)

f(1) f(0)

Рис. 3.3. Схематическое изображение рекурсивных вызовов при

нахождении f(5)

Для ускорения вычислений можно было бы учесть, что



Это приводит к следующей рекурсивной программе функции:



Отметим, что теперь количество рекурсивных вызовов для fiboo(n) имеет порядок равный log2(n) и, скажем, fiboo(200) в символьной форме вычисляется практически мгновенно.

Контрольные примеры.

fiboo(0)=1 fiboo(1)=1 fiboo(10)=89

fiboo(200) → 453973694165307953197296969697410619233826

## Алгоритм Ламберта вычисления e

**Задача 7.** Составить программу вычисления приближения к основанию натуральных логарифмов, то есть к числу е, используя следующий алгоритм Ламберта:



**Решение.** Указанный алгоритм предложен Ламбертом в 1766 году [6, с.70]. Организовать по нему рекурсивные вычисления труда не составляет. Здесь величины a(k) и b(k) задаются рекуррентными соотношениями, похожими на определение чисел Фибоначчи, но нелинейными. Однако это обстоятельство не привносит каких-либо дополнительных затруднений в программную реализацию соответствующих функций. Дело в том, что задание последовательности любыми рекуррентными соотношениями сразу решает проблему триады: осуществлена параметризация задачи, выделена база и задана декомпозиция.

Программа e(k) приближенно вычисляет e, обращаясь к рекурсивным функциям-подпрограммам a(k) и b(k):



Учитывая, что последовательности a(k) и b(k) (k=0,1,2, …) определяются весьма схожим образом, можно построить одну общую функцию двух переменных для вычисления их членов. Отсюда ещё один вариант решения поставленной задачи:



Контрольный пример.



Мы видим, что уже е(5) ≈ e с точностью до 9 знаков после десятичной точки, а начиная с e(8) уже все 15 знаков после точки верные. Если была бы необходимость вычислять по алгоритму Ламберта число e с большей точностью, то пришлось бы программно реализовывать арифметику длинных чисел.

**Замечание.** Предложенные варианты вычисления e можно было бы сделать существенно более эффективными, организовав в них динамические базы, пополняемые в процессе вычислений уже найденными значениями. Тем самым, исключив повторные вычисления элементов последовательностей, можно было бы вычислять a(k), b(k) и d(k,t) за k рекурсивных обращений.

## Наибольший общий делитель

**Задача 8.** Составить программу-функцию возвращающую наибольший общий делитель двух натуральных чисел x и y.

**Решение.** Обозначим через nod(x,y) – наибольший общий делитель x и y. Известно, что

(2)



На этих утверждениях базируется известный итеративный алгоритм Евклида, нахождения наибольшего общего делителя двух целых чисел. Внимательный взгляд на соотношения (2) приводит нас к убеждению, что фактически мы имеем рекурсивное определение функции nod(x,y). На языке Mathcad это надо было бы записать так.



Контрольные примеры.



Обобщим решенную задачу, составив программу-функцию, возвращающую наибольший общий делитель нескольких натуральных чисел ap (p=0..n−1, n≥1), являющихся компонентами вектора v=(a0,a1,…,an−1)T.

Обозначим через nodd(a0,a1,…,an−1) – наибольший общий делитель чисел ap (p=0..n−1). Поскольку

nodd(a0,a1,…,an−1)=nod(nodd(a0,a1,…,an−2),an−1) ,

то соответствующая программа-функция, вычисляющая nodd(v) будет выглядеть так:



Контрольный пример.



## Наименьшее общее кратное

**Задача 9.** Составить программу-функцию, возвращающую наименьшее общее кратное натуральных чисел ap (p=0..n−1, n≥2), являющихся компонентами вектора v=(a0,a1,…,an−1)T.

**Решение.** Обозначим через nok(a0,a1,…,an−1) – наименьшее общее кратное чисел ap (p=0..n−1). Известно, что



и



Поэтому соответствующую программу-функцию можно было бы записать так:



где nod(x,y) − функция нахождения наибольшего общего делителя натуральных x и y.

Контрольный пример.



Функцию nok() можно записать в следующей более компактной форме:



## Биномиальные коэффициенты

**Задача 10.** Составить программу-функцию вычисления биномиальных коэффициентов С(n,m), где n m − целые и 0≤m≤n.

**Решение.** Известно, что



Отсюда и вытекает справедливость следующего рекурсивного определения С(n,m):



Обратите внимание на то, что здесь мы имеем рекурсию сразу по двум аргументам.

Опираясь на функцию C(n,m) как на подпрограмму, построим функцию binom(n,k), возвращающую для целого n≥0 вектор из k последовательных биномиальных коэффициентов: С(n,0), C(n,1),…,C(n,k) (k≤n).

Решение данной задачи можно записать так:



Контрольные примеры.



**Замечание.** Для отыскания всех биномиальных коэффициентов при заданном n не обязательно вычислять binom(n,n). Учитывая, что C(n,k)=C(n,n−k), достаточно вычислить binom(n,(n−mod(n/2))/2).

И еще один способ вычисления биномиальных коэффициентов. Рекурсивная программа-функция tripas(n) вычисляет треугольник Паскаля, то есть значения величин C(i,j) для (0≤i≤n , 0≤j≤i), исходя из формул непосредственно определяющих и декомпозицию и базу:



Справа от функции просчитан контрольный пример для n=4.



Вычисления по tripas(n) реализуются не более чем за n рекурсивных обращений, при этом общее количество операций сложения не превосходит величины



## Задача о Ханойских башнях

Рассмотрим следующую весьма популярную у студентов задачу.

**Задача о Ханойских башнях.** На одном из трех алмазных шпилей надето 64 круглых золотых диска. Диски имеют разные радиусы и расположены на шпиле в порядке убывания радиусов от основания к вершине. Трудолюбивые буддийские монахи день и ночь переносят диски с первого шпиля на второй, используя при необходимости и третий шпиль. При этом неукоснительно соблюдаются следующие правила.

• за один раз можно перемещать только один диск.

• больший диск нельзя располагать на меньшем.

• снятый диск необходимо надеть на какой-либо шпиль перед тем как будет снят другой диск.

Легенда утверждает, что когда монахи закончат свою работу, наступит конец света. Можно было бы подсчитать, что для решения задачи с 64 дисками потребуется 264 – 1 перемещений (около 1020). Поэтому, что касается конца света, то он произойдет по истечении пяти миллиардов веков, если считать, что один диск перемещается за одну секунду. Впрочем, и задачу и легенду для неё придумал в 1883 году математик Э. Люка. Это дает нам право отложить заботы о конце света в сторону и перейти к решению следующей задачи.

**Задача 11.** Составить рекурсивную программу-функцию, которая бы решала поставленную выше задачу о Ханойских башнях при количестве дисков, равном n (n = 1, 2, …).

**Решение.** Введем имена для шпилей: **a** , **b** , **c**. Пусть hanoi(n, **a**, **b**, **c**) искомая функция, возвращающая последовательность перемещений дисков с **a** на **b** c использованием **c** по вышеописанным правилам. При n = 1 решать задачу мы умеем. Необходимо просто произвести операцию “переместить **a** → **b**”. Предположим, что мы умеем решать эту задачу для n – 1 диска. Тогда общая схема рекурсии могла бы выглядеть следующим образом.



Иными словами, переместим n – 1 диск с **a** на **с**. Далее, переместим один оставшийся диск с **a** на **b** и, наконец, переместим n – 1 диск с **c** на **b**. Что нам мешает реализовать эту схему на языке программирования Mathcad? По-видимому то, что в процессе вычисления функции hanoi(n, **a**, **b**, **c**), мы не в состоянии организовать вывод сообщений типа “переместить **a** → **b**”. Остается одно средство. Организовать рекурсивные обращения так, чтобы все подобные ходы-перемещения запоминались в массиве, который и будет возвращаться функцией hanoi(n, **a**, **b**, **c**).

Вот один из возможных вариантов определения функции hanoi():



Функция возвращает матрицу размера k×2, в каждой строчке которой фиксируется перемещение одного диска (откуда, куда). Величина k равна общему количеству перемещений.

**Контрольный пример.** При трех дисках с именами шпилей 1, 2 и 3 получаем следующее решение:



## Экзотические средние

Теперь решим задачу, связанную с экзотическими средними. Рассмотрим два положительных числа а0 и b0 и составим их среднее арифметическое и среднее геометрическое. Продолжим этот процесс и, если числа an и bn уже построены, то определим an+1 и bn+1 следующим образом:

(3)



Известно, что последовательности {an} и {bn} стремятся к общему пределу и, следуя Гауссу, его называют средним арифметико-геометрическим исходных чисел а0 и b0.

**Задача 12.** Составить рекурсивную программу-функцию, по которой для неотрицательных чисел a и b можно было бы приближенно вычислять их **арифметико-геометрическое среднее**.

**Решение.** Параметрами задачи естественно считать исходные величины a, b и количество итераций n по формулам (3). Рекурсию организуем по n, a решением задачи будем считать матрицу (an, bn). Построить соответствующую функцию несложно и выглядеть она может, например, т



Контрольные примеры.

age(100,30,3)=[59.77675556213139 59.77665550389991],

age(100,30,5)=[59.77670553300519 59.77670553300519].

Снова отправляясь от двух положительных чисел а0 и b0 станем последовательно составлять средние арифметические и средние гармонические:

(4)



Известно, что последовательности {an} и {bn}, строящиеся по рекуррентным формулам (4), стремятся к общему пределу. Его называют средним арифметико-гармоническим исходных чисел а0 и b0. Оказывается, что среднее арифметико-гармоническое двух чисел совпадает с их средним геометрическим.

**Задача 13.** Составить рекурсивную программу-функцию, по которой для неотрицательных чисел a и b можно было бы приближенно вычислять их **арифметико-гармоническое среднее**, то есть приближенное значение



**Решение.** Как и в предыдущем случае, параметрами задачи естественно считать исходные величины a, b и количество итераций n по формулам (4). Рекурсию организуем по n, a решением задачи будем считать матрицу (an, bn). Соответствующая функция может выглядеть так:



Контрольный пример.

aga(100,20,7)=[44.7213595499958 44.7213595499958],



## Итерация функции в точке

**Задача 14.** Составить программу для нахождения n-ой итерации (n = 0, 1, 2,…) функции F(x) в точке a.

**Решение.** В соответствии с условиями задачи программа должна вычислять значение выражения вида F(F(F…F(a)…)) при n-кратном использовании операции F. Функция iter(F,a,n) решает поставленную задачу.



Контрольный пример.



## Вещественный корень f(x)

**Задача 15.** Пусть функция f(x) вещественной переменной x непрерывна на отрезке [a, b] и f(a)⋅f(b) ≤ 0. Составить программу нахождения на [a, b] какого-либо вещественного корня f(x).

**Решение.** Во первых, при перечисленных выше условиях по крайней мере один корень f(x) на [a, b] существует. Во вторых, договоримся о том, как понимать слова “найти корень”? Будем считать, что корень ищется с точностью ε > 0, то есть должен быть найден отрезок [α, β] (β – α < 2⋅ε), на котором корень имеется. Тогда в качестве приближенного значения корня может быть взята точка x0 = (β + α)/2.

Для отыскания решения многих задач часто используется метод дихотомии, называемый также методом последовательного деления пополам, бисекции или вилки. В некоторых ранее рассмотренных задачах мы уже сталкивались с этим методом. В нашем случае, когда ищется корень уравнения, суть его в следующем. Пусть ε > 0 задано. Делим отрезок [a, b] точкой с=(b+a)/2 на две равные части и в качестве нового отрезка [a, b] берем ту из его половин, для которой снова f(a)⋅f(b) ≤ 0 и т.д. Ясно, что на некотором шаге будем иметь отрезок [a, b] такой, что b – a < 2⋅ε и f(a)⋅f(b) ≤ 0. Следовательно, приближенное решение найдено и оно равно (b + a)/2.

А как записать предложенный алгоритм с использованием рекурсии? Оказывается все достаточно просто.



Контрольные примеры.

1. y(x):= x3dicho(y, -1, 1, 0.01) = -0.008

1. f(u)=u⋅(u + sin(u) – 3)⋅exp(cos(u))

dicho(f, 1, 3, 0.0001) = 2.18f(2.18)=0

**Задача 16**. Пусть функция f(x) вещественной переменной x непрерывна на отрезке [a, b]. Составить программу нахождения на [a, b] любого вещественного корня f(x). При отсутствии корней, должно быть выдано значение ∞ (10307).

**Решение.** Отличие постановки этой задачи от предыдущей в том, что здесь априори ничего неизвестно о знаках функции на концах отрезка и, следовательно, корней f(x) уже может и не быть. Однако метод дихотомии с успехом может быть применен и в данном случае. Соответствующий алгоритм может быть записан так.



Контрольные примеры.

Рассмотрим функции примеров из предыдущей задачи. Имеем:

dichot(y,1,7,0.001)=10307 , dichot(f,2.17,3,0.0001)=2.18 .

Периодическое продолжение

**Задача 17.** Составить программу, которая для функции g(x), определенной при x ∈ [a,b), строит функцию peri(g,a,b,x), являющуюся периодическим продолжением g(x) на всю действительную ось c периодом ω = b – a.

**Решение.** Нам, очевидно, требуется определить функцию следующего вида.



На языке Mathcad это будет выглядеть практически так же:



Заметим, что при x находящемся вдали от промежутка [a,b) вычисления значения функции peri() требует значительного количества рекурсивных вызовов. Происходит это по той причине, что за один такой вызов мы продвигаемся в направлении [a,b) лишь на расстояние ω=b−a.

Значительно эффективней проводятся вычисления по функции F(g,x,a,b) также являющейся периодическим продолжением g(x) на всю числовую ось.



Контрольные примеры.

1. Пусть y(x) = x2⋅sin(x). Тогда:

peri(y,1,0,2) = 0.841 peri(y,3,0,2) = 0.841

peri(y,−1,0,2) = 0.841 peri(y,1001,0,2)=0.841

2. На рис. 3.4 изображен график функции H(t), являющейся периодическим продолжением функции y(x)= x2⋅sin(x) для x ∈ [−10, 0). H(t) построено с помощью программы-функции F(), а график выведен на промежутке [−10,20) с шагом h=0.1.

t:= −10,−9.9..20 H(t):=perri(y,t,−10,0)

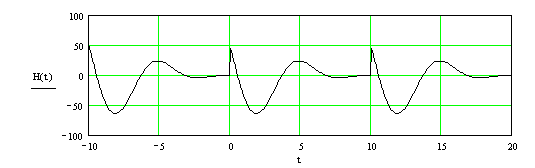


Рис. 3.4 Периодическое продолжение функции y(x)= x2⋅sin(x)

для x ∈ [-10, 0).

## Функция Аккермана

**Задача 18.** Пусть n и m целые неотрицательные числа. Написать программу, вычисляющую классическую в теории рекурсии функцию Аккермана:

(5)



**Решение.** Вычислить функцию Аккермана, исходя непосредственно из определения (5), удается лишь для некоторых малых n и m. Связано это со сложностью и необычностью рекурсивного определения. В общем случае не только ak() вычисляется через ak(), но и второй из аргументов функции также требует рекурсивного вызова ak(). Соответствующая программа-функция может быть записана так:



Контрольные примеры.



Следующий вариант программы для вычисления функции Аккермана включает в себя лишь один рекурсивный вызов:



Замечание. Для m=0..4 справедливы соотношения [5, с. 69]:



Эти формулы могут оказаться полезными при построении контрольных примеров для отладки новых более эффективных вариантов программ вычисления функции Аккермана.

В работе [5, c. 256−260] приведен нерекурсивный вариант алгоритма вычисления значений функции Аккермана.

## Функция Маккарти

**Задача 19. Функция Маккарти.** Показать, что для приведенной ниже рекурсивной программы-функции



при целочисленных значениях n справедлива формула:

(6)



**Решение.** Относительно параметра n возможны три случая:

n > 100,90≤ n ≤100, -∞ < n<90.

В первом из них в силу базы рекурсии следует, что makkarti(n)=n−10. Во втором случае:



Наконец, всякое начальное n<90 в соответствии с декомпозицией через конечное число рекурсивных вызовов приводит ко второму случаю. Отсюда опять makkarti(n)=91. Таким образом (6) справедливо во всех случаях.

Заметим, что из проведенных рассуждений вытекает, что рассматриваемая функция может быть определена более просто, например, так:



## Функция Кадью

**Задача 20.** Показать, что для приведенной ниже рекурсивной программы-функции



при целочисленных значениях x справедлива формула:

(7)



**Решение.** При y=0 и z=1 имеем z=y!. Далее из характера декомпозиции функции cadiou() ясно, что z=y! остается инвариантом в ходе рекурсивных вызовов. Вместе с условием завершения x=y это и дает z=x! и (7) установлено.

## Количество делителей

**Задача 21.** **Количество делителей.** Составить программу-функцию подсчета для натурального числа n количества всех его делителей.

**Решение.** Перейдем к более общей задаче. Подсчитаем для натурального числа n количества всех его делителей, меньших или равных заданному натуральному числу x. Пусть dn(n) и dnx(n,x) − соответственно функции для решения исходной и обобщенной задач. Очевидно, что dn(n)=dnx(n,n).

Рекурсивную функцию dnx(n,x), по которой последовательно подвергаются испытанию на делители n все числа от 1 до x включительно, можно определить так:



Контрольные примеры.



Далее, если n ≥2 и dn(n)=2, то число n – простое. Однако проверка n на простоту этим способом весьма неэкономна.

## Простые числа

**Задача 22.** Составить программу-функцию проверяющую, является ли заданное натурально число n простым?

**Решение.** Пусть рекурсивная функция isprim(n) является решением задачи и



Дальнейшие рассуждения являются иллюстрацией использования классического приема “вложения” (Дж. Маккарти, 1962). Перейдем к рассмотрению следующей обобщенной задачи. **Пусть a, b, n − натуральные числа и 2≤a≤b≤n. Верно ли, что заданное n не делится ни на одно целое из отрезка [a, b]?** Пусть эту задачу решает функция



Ниже приведено три рекурсивных варианта реализации этой функции. По первому из них проверке на делитель n последовательно подвергаются числа: a, a+1, … , b; по второму − эти же числа в обратном порядке и, наконец, по третьему − a, b, a+1, b−1, … .



Контрольные примеры.



Далее, натуральное число n≥2 является простым, если оно не имеет делителей на отрезке , поэтому характеристическая функция isprim(n) через функцию nodiv(n,a,b) может быть выражена так:



Контрольные примеры.



**Задача 23.** Составить программу-функциюpi(x), которая подсчитывает **количество простых чисел**, не превосходящих заданное число x.

**Решение.** С помощью функций nodiv() или isprim() рекурсивный вариант функции pi(x) строится достаточно просто, исходя из такого декомпозиционного утверждения. Количество простых чисел, не превосходящих x, составляется из количества простых чисел меньших или равных x−1 плюс значение функции isprim(x). Поэтому:



Контрольные примеры.



**Задача 24.** Составить программу pn(n), которая вычисляет **n-ое простое число** (n – натуральное).

**Решение.** Предварительно напишем рекурсивную подпрограмму-функ-цию minp(m) нахождения наименьшего простого числа, большего или равного m (m – натуральное число). Сделать это можно, например, так:



Контрольные примеры.



Тогда искомая функция pn(n) может быть определена так:



Контрольные примеры.



## Схема Горнера

**Задача 25.** Составить программу-функцию вычисления значений многочлена по схеме Горнера.

**Решение.** Пусть f(x) многочлен с вещественными или комплексными коэффициентами и v – вектор этих коэффициентов:

(8)



(9)



Вычисление f(x) в точке x по схеме Горнера проводится от самых внутренних скобок и далее в соответствии с представлением:



Отсюда ясно, как записать рекурсивный и не рекурсивный варианты алгоритмов вычисления f(x). Нас интересует только первый из них. Параметрами задачи можно считать вектор v и точку a для вычисления f(a). Тривиальный случай – многочлен нулевой степени: f(a) = a0. Декомпозиция получается из указанной выше расстановки скобок в записи f(x). Соответствующая программа-функция выглядит, например, так.



Контрольные примеры.



**Замечание.** Параметр n – это константа, равная степени многочлена. По вектору v она определяется однозначно. Однако в каждом рекурсивном вызове проводится перевычисление n. Избежать лишней вычислительной работы в подобных ситуациях можно двумя способами. Или определять такие константы заранее вне текста программы-функции или передавать их значения через дополнительные аргументы функции.

## Деление многочлена на двучлен

**Задача 26.** Составить программу-функцию, по которой для многочлена (8) с коэффициентами, составляющими вектор (9) и двучлена x−x0 (x0 − вещественное или комплексное число), вычисляется вектор c:



такой, что



и



**Решение.** Если в соотношении, связывающем многочлены f(x) и q(x), осуществить приведение к общему знаменателю и приравнивание коэффициентов при одинаковых степенях x, то для определения компонент вектора c получим следующую систему линейных уравнений.



Отсюда вытекает, что c = qu(v,x0), где рекурсивное определение функции qu() может выглядеть, например, так.



Контрольный пример.



Иными словами:



## Произведение двух многочленов

**Задача 27.** Пусть коэффициенты многочленов fn(x) и gm(x) заданы компонентами векторов v и w:

(10)



(11)



Составить рекурсивную программу-функцию вычисляющую коэффициенты многочлена hn+m(x)=fn(x)⋅gm(x) и возвращающую их в виде компонентов вектора:



**Решение.** Поскольку



где



то вполне можно организовать рекурсию по параметру m − степени второго сомножителя. И в качестве решения может быть предложена следующая функция:



Ясно, что аналогично можно было бы реализовать рекурсию и по параметру n − степени первого сомножителя. В любом случае величины m и n определяют количество рекурсивных обращений. Поэтому в данной задаче рекурсию выгодно реализовывать по параметру со значением, равным min(n,m).

Контрольный пример.



## Произведение биномов

**Задача 28.** Составить программу-функцию возвращающую коэффициенты многочлена g(x), который получается в результате перемножения биномов (x−vk): (k=0,1,…n−1; n≥1):



**Решение.** Будем считать, что свободные члены биномов заданы в виде компонентов некоторого вектора v: а результат вычислений должен возвращаться также в виде вектора. Поскольку



то несложно организовать рекурсию по количеству перемножаемых биномов. Соответствующая программа функция могла бы выглядеть так:



Контрольный пример.



## Деление многочлена на многочлен

**Задача 29.** Пусть выполнены соотношения (10)−(11), то есть многочлены fn(x) и gm(x) степеней n и m (n,m≥0) соответственно заданы своими коэффициентами в виде компонентов векторов v и w. Пусть, далее, при m=0, то есть если gm(x) есть константа, gm(x)≠0. Составить программу-функцию нахождения частного q(x) и остатка r(x) при делении fn(x) на gm(x):

fn(x)=q(x)⋅gm(x)+r(x), (12)

где степень r(x) меньше m (при m=0 r(x)≡0).

**Решение**. Представление (12) единственно. В дальнейшем нам удобно считать, что степень r(x) равна m−1 с возможно нулевыми коэффициентами при старших степенях x.

При m=0 решение задачи очевидно:

q(x)=fn(x)/w0,r(x)=0. (13)

Далее, при n<=m имеем

(14)



Пусть n>m и q1(x) и r1(x) − частное и остаток от деления (fn(x)−an-1)/x на gm(x):



Из этого соотношения вытекает, что



Вместе с (12) это дает:

(15)



Если соотношения (13)−(14) рассматривать в качестве базы рекурсии, то равенства (15) определяют декомпозицию. Считаем, что в результате вычислений должен быть сформирован и возвращен составной вектор qr=[q r]T, где q и r соответственно векторы коэффициентов многочленов q(x) и r(x). Соответствующая программа-функция, реализующая эти идеи, выглядит так:



Контрольные примеры.



**Замечание.** Функция poldiv() возвращает составной вектор [q r]T , в котором второй компонент-вектор r может содержать “ведущие” нули. При необходимости эти нули можно погасить, то есть выделить из r подвектор, начинающийся с первого из ненулевых компонент r. Сделать это можно, например, с помощью приведенной ниже рекурсивной функции nulera(u). Если все компоненты u равны нулю, то nulera(u) возвращает 0.



## Разбиение целого на части

**Задача 30.** Составить программу-функцию подсчета количества ξ(m) разбиений натурального числа m, то есть его представления в виде суммы натуральных чисел.

**Решение.** Пусть, например, m=6. Тогда разбиениями m являются его представления в виде:

6;

5+1;

4+2, 4+1+1;

3+3, 3+2+1, 3+1+1+1;

2+2+2, 2+2+1+1, 2+1+1+1+1;

1+1+1+1+1+1;

Таким образом, ξ(m)=11 и понятно, что простым перебором возможных случаев уже при m>10 справиться с задачей достаточно сложно.

Для решения исходной задачи перейдем к рассмотрению обобщенной задачи. Составить программу-функцию подсчета количества P(m,n) разбиений натурального числа m со слагаемыми, не превосходящими n.Ясно, что ξ(m)=P(m,m). Поэтому, достаточно научиться вычислять значения функции P(m,n). Но для неё нетрудно выделить рекурсивную базу и указать правило декомпозиции. Сделать это можно исходя из следующих вполне прозрачных свойств этой функции.

1. P(m,1)=1 − существует только одно разбиение m, в котором слагаемые не превосходят единицы, а именно: m=1+1+…+1.
2. P(1,n)=1 − число единица имеет только одно представление при любом n.
3. P(m,n)=P(m,m) при n>m − слагаемые, большие m в разбиениях отсутствуют.
4. P(m,m)=P(m,m−1)+1 − существует лишь одно разбиение со слагаемым равным m. Все иные разбиения имеют слагаемые не превосходящие m−1.
5. P(m,n)=P(m,n−1)+P(m−n,n) (n<m). Обоснование этого соотношения проводится так. Все разбиения m на сумму слагаемых, не превосходящих n можно разбить на два непересекающихся класса: суммы, не содержащие n в качестве слагаемого и суммы, содержащие такое n. Количество элементов первого класса равно P(m,n−1), а количество элементов второго класса подсчитаем так. Без учета слагаемого n суммы элементов второго класса равны m−n. Значит их общее количество равно P(m−n,n) и, следовательно, общее количество элементов второго класса также равно этой величине. Тем самым свойство 5 установлено.

Первые два свойства определяют базу рекурсии, а три следующие задают декомпозицию. Строго в соответствии с этими утверждениями и составлена рекурсивная программа-функция deco(n,m) для вычисления величины P(m,n).



Контрольные примеры.



## Максимальный и минимальный элементы

Пусть одномерный массив задан вектором Часто возникает задача поиска максимального (минимального) элемента v. Вне зависимости от того, идет ли речь о нахождении максимального по значению элемента или его позиции в v, поиск можно реализовать за один просмотр массива. Правда в последнем случае решение может оказаться неоднозначными и постановка задачи требует уточнения. Например, отыскивается позиция максимального элемента с наименьшим индексом. В любом случае сравнение характеристик различных алгоритмов поиска проводят по количеству тех или иных выполняемых ими операций. Чаще всего это операции сравнения.



Рекурсивная функция maxv(v) находит максимальный элемент v ровно за n−1 операцию сравнения. Декомпозиция реализуется по длине вектора и опирается на такое утверждение. Решить исходную задачу для v − это то же самое, что решить её для подвектора, получаемого из v удалением его первого компонента, сравнить полученное значение с удаленным элементом и, наконец, выбрать не меньшее из них.



Контрольные примеры.



**Замечание.** При подсчете количества операций сравнения элементов массива не учитываются сравнения с управляющими переменными рекурсии или цикла. Это же самое касается и других программ, приведенных ниже.

Аналогично строится и функция minv(v) − нахождения за n−1 операцию сравнения минимального по значению элемента массива. Нетрудно написать рекурсивную функцию одновременного поиска минимального и максимального значений v за 2⋅n−2 операции сравнения. Выглядеть она может, например, так:



Контрольные примеры.



Однако одновременное нахождение максимального и минимального значений v может быть реализовано гораздо эффективней. Соответствующий результат зафиксирован в следующей задаче.

**Задача 31.** Написать рекурсивную программу-функцию, находящую одновременно максимальный и минимальный элементы массива v за операций сравнения.



**Решение.** Если длина вектора v равна единице, то решением задачи можно считать вектор При длине вектора v, равной двум, решение находится за одно сравнение. Пусть длина вектора больше двух. Тогда из двух первых компонентов непосредственно найдем наибольшее и наименьшее (одно сравнение), решим исходную задачу для подвектора, получающегося из v удалением этих компонентов и, наконец, осуществим правку полученного решения для подвектора за счет первых двух компонентов (два сравнения). Именно эти идеи и реализуются рекурсивной функцией minmax():



Контрольный пример.



Подсчитаем теперь S − общее количество сравнений:



Тем самым решение задачи завершено полностью.

В следующем варианте minmax() − функции minmax1() используются три вспомогательных параметра: a, b, i. Первые два из них служат для фиксации текущих значений минимального и максимального элементов массива, а последний − для нумерации рекурсивных обращений. Как и в предыдущем случае, каждый рекурсивный вызов уменьшает обрабатываемый массив на два элемента. Но здесь, в отличие от minmax(), они удаляются с разных его концов. Использование вспомогательной функции mima(v) избавляет нас от сложного обращения к minmax1().



Контрольный пример.



**Задача 32.** Написать рекурсивную программу-функцию, находящую одновременно позиции максимального и минимального элементов массива v за операций сравнения.



**Решение.** Для решения данной задачи можно использовать тот же самый алгоритм, который реализуется функцией minmax1(), подвергая запоминанию не текущие значения минимального и максимального элементов массива, а их индексы. Соответствующие функции posmima() и pmima() можно записать так:



Контрольный пример.



**Замечание.** Рекурсивная функция posmima() позволяет реализовать сортировку массива v по такому алгоритму:



Контрольные примеры.



## Абракадабра

**Задача 33.** Последовательность из латинских букв строится следующим образом. На нулевом шаге она пуста. На каждом последующем шаге последовательность удваивается, то есть приписывается сама к себе, и к ней слева добавляется очередная буква алфавита (a,b,c,…). По заданному числу n определить символ, который стоит на n-ом месте последовательности, получившейся после шага 26.

**Решение.** Эта задача предлагалась участникам областных туров Всероссийской олимпиады школьников по информатике в 1998 году. Приведем первые шаги формирования последовательности: 0 − пустая последовательность, 1 − a, 2 − baa,3 − cbaabaa, 4 − dcbaabaacbaabaa, … . Мы имеем явно рекурсивный процесс.

Построим более общую программу-функцию, чем это требуется по условиям задачи. Пусть abra(k,n) − n-ая буква в последовательности, полученной на шаге k (k=1..26). Тогда ясно, что abra(k,1) равно k-ой букве латинского алфавита. Этот факт можно взять в качестве базы рекурсии, а функцию для определения этой буквы записать так:



Декомпозицию удобно организовать по k, проводя “раскрутку” последовательности по шагам в обратном направлении. Это приводит к следующей функции:



Контрольные примеры.



## Вложенные многоугольники

**Задача 34.** **Вложенные квадраты.** Пусть на плоскости первый квадрат задан точками: M1(-1,-1), M2(-1,1), M3(1,1), M4(1,-1). Второй квадрат строится так, что вершины первого квадрата являются серединами его сторон и т.д. Составить рекурсивную программу-функцию, которая по заданному натуральному n строит систему из 2⋅n вложенных друг в друга описанным образом квадратов, точнее создает массив точек, последовательное соединение которых на плоскости отрезками и формирует эту систему (см. рис. 3.5).

**Решение.** Если учесть тот факт, что по условию задачи всегда строится четное число вложенных квадратов, то последовательность точек для построения первых двух из них можно задать непосредственно и считать соответствующую матрицу beg базой рекурсии:



Обратите внимание, что в beg каждый из квадратов задается не четырьмя, а пятью точками. При этом первая и последняя из них совпадают. Это поможет впоследствии правильной прорисовке квадратов.

Далее, декомпозиция определяется таким утверждением. Если уже построена матрица ma точек для 2⋅(n−1) вложенных квадратов, то, пополнив её точками матрицы , получим матрицу точек для 2⋅n таких квадратов. Соответствующая рекурсивная функция, базирующаяся на этих фактах, может выглядеть так:



Контрольный пример. Результат вычислений представлен на рис. 3.5 c неодинаковым масштабом по осям.

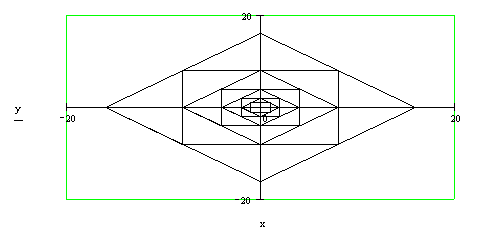


Рис. 3.5. Вложенные 2⋅n квадратов (n=4)

**Задача 35.** **Вложенные многоугольники.** Пусть на плоскости задан правильный n-угольник, вписанный в единичную окружность одна из вершин которого имеет координаты (cos(α), sin(α)), где α − некоторый угол. Второй правильный n-угольник строится так, что его вершины являются серединами сторон первого многоугольника и т.д. Составить рекурсивную программу-функцию, которая по заданному натуральному n строит систему из n вложенных друг в друга описанным образом многоугольников, точнее создает массив точек, последовательное соединение которых на плоскости отрезками и формирует эту систему (см. рис. 3.6).

**Решение.** Это задача в каком-то смысле является обобщением предыдущей. Здесь выводятся правильные вписанные n-угольники, количество их не обязательно четно и первая из вершин начального n-угольника может лежать в любой точке единичной окружности. Правда, здесь многоугольники не “описываются” один вокруг другого, а “вписываются” друг в друга. Но существа дела это не меняет.

Прежде всего, составим программу, которая формирует массив вершин правильного n-угольника, вписанного в окружность радиуса r и содержащего вершину (r⋅cos(α) r⋅sin(α)). Сделать это можно, например, так:



Тогда массив polyone(n,1,α) может служить базой рекурсии. Более того, эта функция при правильном выборе r и α позволяет получить массив вершин любого из следующих вписанных n-угольников, что помогает организовать и декомпозицию. Если уже построена матрица ma точек для (n−1)-го вписанного правильного n-угольника, то, пополнив её точками матрицы:



получим матрицу точек для 2⋅n таких многоугольников. Соответствующая рекурсивная функция, реализующая эти идеи, может выглядеть так:



Контрольный пример. Результат вычислений представлен на рис. 3.6 c неодинаковым масштабом по осям.

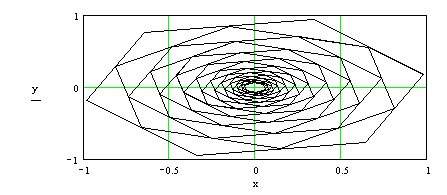


Рис. 3.6. k вложенных правильных n-угольников (k=20, n=6)

Несколько слов перед формулировкой новой задачи. В основах анализа [8, с. 17−38] операции **сложения** и **умножения** над натуральными числами определяются рекурсивным способом, опираясь на аксиомы Пеано “о существовании последующего числа” и “индукции”. Первая из названных аксиом звучит так: “для каждого натурального x имеется и притом только одно натуральное число, называемое его последующим и обозначаемое число x′ ”.Постараемся промоделировать указанные выше и некоторые иные операции.

Моделирование арифметических операций

**Задача 36.** Для целых неотрицательных чисел n, m разрешены операции:

нахождения последующего числа (n+1) и предыдущего числа n−1 (n>0). Промоделировать с помощью рекурсивных функций операции нахождения суммы (n+m), разности (n−m (n≥m)), умножения (n⋅m), возведения в степень nm (n>0), частного и остатка при делении n на m (n/m).

**Решение.**

**A. Сумма: a+b.** Очевидноесоотношение



задает одновременно и базу рекурсии и правило декомпозиции. По нему и построена следующая рекурсивная программа-функция:



Контрольный пример:



**B. Разность: a−b (a≥b).** В данном случае база рекурсии и декомпозиция могут быть извлечены из соотношения:



Соответствующая рекурсивная программа-функция выглядит так.



Контрольный пример:



**C. Умножение: a⋅b.** Будем считать, что операция сложения уже определена. Тогда соотношение, из которого легко извлекаются база и декомпозиция для данного случая выглядит следующим образом:



Соответствующая рекурсивная программа-функция может быть записана, например, так:



Контрольный пример:



**D. Возведение в степень: ab (a≠0).** Будем считать, что операция умножения уже определена. Тогда:



и рекурсивная программа-функция возведения в степень может быть задана так:



Контрольный пример:



**E. Частное и остаток: a/b (b>0).** В данной задаче речь идет об отыскании величин q и r из представления: a=q⋅b+r (0≤r<b, q≥0). Будем предполагать, что операция вычитания уже определена, а ответ должен быть возвращен в виде вектора с двумя компонентами: (q r)T. Если a<b, то q=0 и r=a. Этот факт и определяет базу рекурсии. Если же b>a, то a/b=1+(a−b)/b, что позволяет организовать декомпозицию. Все сказанное учтено в рекурсивной функции divi(q,a,b). В ней первый аргумент q является вспомогательным параметром. При обращениях к divi() он должен определять базовое значение частного, то есть быть равен нулю. Однако проще решать задачу с помощью вспомогательной функции div(a,b), возлагая на неё решение вопроса о правильном значении вспомогательного параметра вызовах divi(). Это довольно типичный случай при обобщениях исходной задачи за счет введения дополнительных параметров.



Контрольный пример.



**Замечание.** Моделирование различных операций возможно не только для целых неотрицательных чисел. Их можно было бы определить и для множества всех целых чисел и даже для множества вещественных чисел. Ограничимся рассмотрением одного примера. Напишем рекурсивную программу-функцию нахождения дробной части вещественного числа, если разрешены лишь операции x+1 и x−1. Вот один из возможных достаточно ясных вариантов её определения:



Контрольный пример.



Синтаксические языковые конструкции

**Задача 37.** Составить программу-функцию проверяющую, является ли данная последовательность символов **идентификатором** языка Фортран.

**Решение.** Будем считать, что речь идет о версиях Фортрана, где идентификатор определялся как последовательность из шести заглавных латинских букв и (или) цифр, начинающаяся с буквы и для символов была принята ASCII кодировка. Превратить последовательность символов в вектор соответствующих им ASCII-кодов можно с помощью встроенной функции str2vec(). Например:



Дополнительное ограничение на первый символ идентификатора (буква, но не цифра) усложняет общую проверку (или буква, или цифра). Чтобы действия были стандартными для всех символов, проверку первого из них будем осуществлять на допустимость отдельно в головной программе. Естественно там же располагать и проверку длины слова, которая должна находиться в пределах от 1 до 6. Тогда в подпрограмме останется решить вполне рекурсивную подзадачу: “если первый символ исходного слова не буква и не цифра, то формируем ответ “не идентификатор”. В противном случае, если длина слова равна единице, то возвращаем ответ “идентификатор”, а иначе укорачиваем слово на первый символ и снова решаем эту же подзадачу. Все сказанное и реализуется головной программой identity(word) и рекурсивной подпрограммой iden(v):



Контрольные примеры.



**Замечание.** В любом языке программирования все базовые языковые конструкции (идентификаторы, константы, переменные, выражения, метки, типы и т.п.) определяются рекурсивно. Особенно наглядно это видно, когда они представлены с помощью синтаксических диаграмм [7, c. 685−703] или в форме Бэкуса-Наура. Подобные определения в рамках конкретного языка программирования могут служить наборами тренировочных заданий по написанию рекурсивных программ и даже простейших рекурсивных трансляторов.

Приведем пример. Идентификатор в Паскале определяется, как и в Фортране, но без ограничений на длину последовательности символов. С помощью синтаксической диаграммы это выглядит так, как это указано на рис.3.7., а в форме Бэкуса-Наура следующим образом (значок “::=” читается как “есть по определению”):

<идентификатор>::=<буква>

<идентификатор>::=<идентификатор><буква>

<идентификатор>::=<идентификатор><цифра>

**Идентификатор**

**буква**

**буква**

**цифра**

Рис. 3.7. Синтаксическая диаграмма идентификатора (Паскаль)

И в том и в другом случаях идентификатор определяется сам через себя.

## Проблемная ситуация

**Задача 38.** При любом ли натуральном n ли рекурсивная функция



равна 1?

**Решение.** Хотя данная строка и начата со слова “решение”, ответа на поставленный вопрос мы не знаем и, по-видимому, на сегодняшний день его не знает никто. Более того, неизвестно, для любого ли n problem(n) вычисляется за конечное число шагов. Рассмотренную задачу называют 3⋅n+1 проблемой. Мы включили её в список задач для того, чтобы обратить внимание читателя на следующий факт. Достаточно простые с виду рекурсивные определения функций могут таить в себе глубокие проблемы, решения которых лежат совсем не на поверхности. Тем не менее, конкретные вычисления problem(n) при разных n приводят к одному и тому же значению, равному 1. Ниже приведена рекурсивная программа для проверки истинности утверждения “problem(n)=1” при значениях n из диапазона k1..k2.



Контрольные примеры.



1. Задача Иосифа Флавия

С именем известного историка первого века Иосифа Флавия связывают следующую задачу-легенду. В ходе иудейской войны он в составе отряда из 41 воина был загнан римлянами в пещеру. Не желая сдаваться, осажденные воины решили покончить жизнь самоубийством и разработали для этого следующую процедуру. Они выстроились в круг и, начиная отсчет с конкретной позиции, каждый третий должен был убивать себя, пока не останется ни одного человека. Математически одаренный Иосиф считал подобный конец бессмысленным и потому поставил себя и своего друга на такие позиции, что после серии из 39 самоубийств они остались вдвоем, чем и спасли себе жизнь. Что это были за позиции?

Дадим этой задаче-легенде более точную и обобщенную формулировку, освободившись от суицида. При этом будем искать рекурсивные решения двух ниже приведенных задач, различающихся завершением соответствующей процедуры. Подробнее рекурсия как метод решения практических задач обсуждается в следующем параграфе.

**Задача 1.** По окружности в направлении движения часовой стрелки расположены n последовательных натуральных чисел от 1 до n. При перемещении по числам 1, 2, ... каждое k-ое число (k>1) вычеркивается (удаляется). Этот процесс продолжается до тех пор, пока чисел не станет меньше k. Определить оставшиеся числа.

**Задача 2.** По окружности в направлении движения часовой стрелки расположены n последовательных натуральных чисел от 1 до n. При перемещении по числам 1, 2, ... каждое k-ое число (k>1) вычеркивается (удаляется). Этот процесс продолжается до тех пор, пока не останется одно число. Определить его.

**A. Решение первой задачи Флавия при k=2***.*

Ниже рассмотрены три варианта A1−A3 решения задачи 1 при k=2, опирающиеся на разные идеи.

A1. Если n=2⋅s, то после первого прохода по кругу останутся числа: 1, 3, ... 2⋅s−1 и следующий проход начнется с вычеркивания числа 3. Это все равно, как если бы мы начинали с s последовательных натуральных чисел от 1 до s, но каждое уцелевшее число удваивали и результат уменьшали на 1. Отсюда, если fla1(n) − функция, решающая поставленную задачу, то fla1(1)=1 и

fla1(2⋅s)=2⋅fla1(s) − 1 (s≥1). (16)

Аналогичные рассуждения показывают, что

fla1(2⋅s+1)=2⋅fla1(s) + 1 (s≥1). (17)

Величины fla1(n) назовем числами Флавия. Соотношения (16) и (17) сразу же позволяют написать следующую рекурсивную программу-функцию вычисления значений fla1(n).



A2. Исследование рекуррентных соотношений (16)−(17) показывает, что

fla1(2s + q)=2⋅q+1 (s≥0, 0≤q<2⋅s) (18)

Отсюда получаем еще один рекурсивный алгоритм для вычисления чисел Флавия (см. ниже). При этом вспомогательная рекурсивная функция power(n,0) вычисляет значение s, удовлетворяющее соотношению (18), то есть уменьшенное на 1 количество цифр двоичного разложения n, а функция fla2(n) непосредственно вычисляет число Флавия для заданного n.



A3. Еще один способ нахождения чисел Флавия дается программой-функцией flavec(v), где v=(1 2 3 ... n)T − вектор. Подавать такой вектор в качестве аргумента необязательно. Проще обращаться к flavec(v) c помощью функции fla3(n), где по заданному n генерируется соответствующий вектор v. Отметим, что в flavec(v) используется рекурсивный алгоритм непосредственного вычеркивания каждого второго числа. При этом вектор v перестраивается при каждом новом перемещении по кругу.



Контрольные примеры.

1. fla1(6)=5fla2(6)=5fla3(6)=5

2. fla1(11)=7 fla2(11)=7 fla3(11)=7

3. fla1(1000)=997fla2(1000)=997fla3(1000)=997

**B. Решение первой задачи Флавия в общем случае.**

Ниже рассмотрены два варианта B1−B2 решения задачи 1 в общем случае. Первый из них представляет прямое обобщение алгоритма из пункта A3 и реализует рекурсию по каждому вычеркнутому элементу. Во втором варианте рекурсия организована по отдельным проходам по окружности.

B1. Способ A3 решения задачи 1 при k=2 хоть и несколько громоздкий, но он достаточно просто переносится на общий случай. При этом естественно считать k аргументом функции**.** Тогда решение задачи дает рекурсивная программа-функция flave(v,k), где v=(1 2 3 ... n)T − вектор. Однако проще использовать для этих целей функцию fla4(n,k), формирующую по заданному целому n требуемый вектор v, а затем уже обращающуюся k flave(v,k).



B2. Приведенный ниже способ решения первой задачи Флавия отличается от предыдущего лишь способом организации рекурсии. Здесь она реализована не по каждому вычеркнутому элементу, а по отдельным проходам по окружности. Решение задачи дается программой-функцией flavmek(v,k), где v=(1 2 3 ... n)T − вектор. Однако проще использовать для этих целей функцию fla5(n,k), формирующую по заданному целому n требуемый вектор v, а затем уже обращающуюся к flavmek(v,k). Обратите внимание на используемый в fla5(n,k) метод формирования вектора v.



Контрольные примеры.

1. fla4(6,2)=5 fla5(6)=5

2. fla4(41,3)T =[16 31] fla5(11) T =[16 31]

3. fla4(1000,5)T=[563 763 802 73] fla5(1000) T=[563 763 802 73]

**С. Решение второй задачи Флавия в общем случае.**

Пусть функция flav(v,k), где v=(1 2 ... n)T решает поставленную задачу и пусть w − вектор, полученный из v вычеркиванием одного k-го компонента. После каждой такой операции будем организовывать рекурсивный вызов flav(w,k), прекращая вычисления тогда, когда длина вектора станет равной единице. Пусть le - длина вектора v и s=mod(k,le). Нетрудно видеть, что после одного вычеркивания получим:

w=(1 2 ... le−1)T при s=0 ,

w=(2 3 ... le)T при s=1 ,

w=(s+1,s+2,...,le,1,2,...,s−1) при s>=2 .

Поэтому функцию flav(v,k) и обращающуюся к ней функцию fla6(n,k) можно определить следующим образом:



Контрольные примеры.

fla6(10,5)=3 fla6(5,10)=4 fla6(1000,7)=404 .

**13.4 Системы счисления**

**A. Перевод чисел из десятичной системы в p-ичную систему**

Не ограничивая общности речь можно вести о неотрицательных числах.Пусть p∈{2,3,…} и цифры p-ичной системы − это последовательные десятичные числа: 0, 1, ... p−1. Рассмотрим 6 конкретных задач. В трех первых из них речь идет о переводе естественным образом заданных десятичных чисел в p-ичную систему счисления. В следующих трех задачах речь идет о переводе десятичных чисел, цифры которых заданы в виде последовательных компонентов векторов, в p-ичную систему счисления. Во всех случаях результат формируется в виде вектора, компоненты которого p-ичные цифры исходного числа.

**Задача 1.** Составить программу-функцию перевода целых неотрицательных десятичных чисел m в систему счисления по основанию p.

**Решение.** Функция dec\_p\_i(m,p) решает поставленную задачу, используя рекурсивный алгоритм последовательного деления. Результат формируется в виде вектора, компоненты которого p-ичные цифры m.



Контрольные примеры.



.



Замечания.

1. Если разряды p-ичного числа необходимо формировать не от старшего разряда, а от младшего и далее, то в программе dec\_p\_i() первый и второй аргументы функции stack() необходимо поменять местами.

2. При переводе неотрицательных десятичных чисел в конкретную систему счисления, в функции dec\_p\_i() достаточно иметь один аргумент. Например, перевод в двоичную систему можно осуществлять следующей программой-функцией dec\_b\_i(m).



Контрольные примеры.



Как мы уже отмечали при реализации функций dec\_p\_i(m,p) и dec\_b\_i(m) использован рекурсивный вариант алгоритма последовательного деления − выделения цифр p-ичной системы для целых чисел. Пояснений требуют лишь фрагменты вида identity(1)⋅x. Дело в том, что функция stack() в качестве своих аргументов использует векторы или матрицы. И смысл записи identity(1)⋅x состоит в превращении скаляра х в матрицу размера 1×1 с элементом x.

**Задача 2.** Составить программу-функцию перевода правильной неотрицательной десятичной дроби y в систему счисления по основанию p.

**Решение.** Функция dec\_p\_f(y,p,k) решает поставленную задачу, используя рекурсивный алгоритм последовательного умножения. Результат формируется в виде вектора с не более чем k (k=1,2,…) компонентами, которые суть p-ичные цифры числа y, начиная от старших разрядов и далее.



Контрольные примеры.



**Задача 3.** Составить программу-функцию перевода неотрицательного действительного десятичного числа a в систему счисления по основанию p (p=2, 3, …).

**Решение.** Функция dec\_p(a,p,k) решает поставленную задачу, осуществляя перевод десятичного числа a в p-ичную систему счисления. Результат вычислений формируется в виде составного вектора. Компоненты этого вектора снова векторы, содержащие соответственно цифры целой и дробной частей числа a в p-ичной системе счисления. Цифры целой и дробной части a возвращаются, начиная со старших разрядов и далее. В дробной части присутствует не более чем k (k=1,2,...) цифр. При вычислениях функция dec\_p() обращается к двум рекурсивным функциям dec\_p\_i() и dec\_p\_f().



Контрольные примеры.



**Задача 4.** Пусть m=(v0v1…vn−1)10 − целое десятичное неотрицательное число, цифры которого от старшей и далее заданы последовательными компонентами вектора v=(v0, v1, …,vn−1)T . Составить программу-функцию перевода m в систему счисления по основанию p.

**Решение.** Функция dec\_p\_iv(v,p) осуществляет перевод v в p-ичную систему счисления. Результат вычислений формируется, начиная от старших разрядов и далее, в виде вектора, компоненты которого суть p-ичные цифры исходного числа. Функция dec\_p\_iv(v,p) отличается от ранее рассмотренной функции dec\_p\_i(v,p) лишь формой представления первого аргумента. Поэтому её вычисление можно свести к вычислению dec\_p\_i(v,p) с предварительным обращением к рекурсивной функции dv\_norm(v), переводящей десятичное число из векторного представления в нормальную форму. Это и сделано ниже.



Контрольные примеры.



**Задача 5.** Пусть y=(.v0v1…vn−1)10 − правильная неотрицательная десятичная дробь, цифры которой от старшей и далее заданы последовательными компонентами вектора v=(v0, v1, …,vn−1)T. Составить программу-функцию перевода y в систему счисления по основанию p.

**Решение.** Функция dec\_p\_fv(v,p,k) осуществляет перевод v в p-ичную систему счисления. Результат вычислений формируется, начиная от старших разрядов и далее, в виде вектора длины не более k, компоненты которого суть p-ичные цифры исходного числа. Функция dec\_p\_fv(v,p,k) отличается от ранее рассмотренной функции dec\_p\_f(v,p,k) лишь формой представления первого аргумента. Поэтому её вычисление можно свести к вычислению dec\_p\_f(v,p,k) с предварительным обращением к рекурсивной функции dv\_normf(v), переводящей десятичную дробь из векторного представления в нормальную форму. Это и сделано ниже.



Контрольные примеры.



**Задача 6.** Пусть действительное неотрицательное десятичное число a представлено двумя векторами in и fr, компоненты которых последовательные десятичные цифры, начиная от старшей и далее, соответственно целой и дробной части а. Составить программу-функцию перевода a в систему счисления по основанию p.

**Решение.** Функция dec\_pv(in,fr,p,k) решает поставленную задачу, осуществляя перевод десятичного числа a в p-ичную систему счисления. Результат вычислений формируется в виде составного вектора. Компоненты этого вектора снова векторы, содержащие соответственно цифры целой и дробной частей числа a в p-ичной системе счисления. Цифры целой и дробной части a возвращаются, начиная со старших разрядов и далее. В дробной части присутствует не более чем k (k=1,2,...) цифр. При вычислениях функция dec\_p() обращается к двум рекурсивным функциям dec\_p\_iv() и dec\_p\_fv().



Контрольные примеры.



**B. Перевод чисел из p-ичной системы в десятичную систему**

Пусть p∈{2,3,…} и цифрами p-ичной системы являются десятичные числа 0,1,... ,p−1. Будем считать, что рассматриваются неотрицательные p-ичные числа, а их цифры от старшего разряда и далее задаются последовательными компонентами векторов. Рассмотрим 3 конкретные задачи.

**Задача 7.** Пусть m=(v0v1…vn−1)p − целое p-ичное неотрицательное число, цифры которого от старшей и далее заданы последовательными компонентами вектора v=(v0, v1, …,vn−1)T . Составить программу-функцию перевода m в десятичную систему счисления.

**Решение.** Функция p\_dec\_i(v,p) решает поставленную задачу, используя рекурсивный алгоритм последовательного деления. Результат формируется в естественной форме.



Контрольные примеры.



**Задача 8.** Пусть y=(.v0v1…vn−1)p − правильная неотрицательная десятичная дробь, цифры которой от старшей и далее заданы последовательными компонентами вектора v=(v0, v1, …,vn−1)T . Составить программу-функцию перевода y в десятичную систему счисления.

**Решение.** Функция p\_dec\_f(v,p) решает поставленную задачу, используя рекурсивный алгоритм последовательного умножения. Результат формируется в естественной форме.



Контрольные примеры.



**Задача 9.** Пусть действительное неотрицательное p-ичное число а представлено двумя векторами in и fr, компоненты которых последовательные p-ичные цифры, начиная от старшей и далее, соответственно целой и дробной части а. Составить программу-функцию перевода a в десятичную систему счисления.

**Решение.** Функция p\_dec(in,fr,p) решает поставленную задачу, осуществляя перевод p-ичного числа a в десятичную систему счисления. Результат вычислений формируется в естественной форме. При вычислениях функция p\_dec() обращается к двум рекурсивным функциям p\_dec\_i() и p\_dec\_f().



Контрольные примеры.



Замечание. Перевод чисел из p-ичной системы счисления в q-ичную систему (p,q∈{2,3,…}) можно осуществлять через промежуточную десятичную систему.

**13.5 Генераторы перестановок**

Ранее в разделе 12.9 мы описали один из алгоритмов получения n! перестановок из элементов множества S={a0, a1, …, an−1}, где ak (k=0..n−1) − попарно различные действительные числа. Считая, что S={1,2,…,n}, обсудим еще несколько вариантов рекурсивных алгоритмов генерирования перестановок. Отличаются друг от друга они разными характеристиками: быстродействием, компактностью записи, количеством транспозиций при получении очередной перестановки и т.д.

**A.** **Метод вертикальной прогонки.** При S={1} имеем одну перестановку − 1. Если уже имеется последовательность перестановок из n−1 элемента {1,2,…,n−1}, то получить все перестановки из n элементов {1,2,…n} можно способом, который мы будем именовать как “метод вертикальной прогонки” элемента n. Суть его в следующем. Рассмотрим n идентичных экземпляров (групп) последовательностей перестановок из элементов {1,2,…,n−1}. В первом экземпляре в конец каждой перестановки поместим элемент n. Во втором экземпляре этот элемент поместим на предпоследнем месте и т.д. Наконец, в последнем экземпляре поместим n перед первым элементом. На рисунке 13.4 описанная процедура демонстрируется при переходе от n=3 к n=4. Нетрудно понять, что указанные действия любую перестановку из n−1 элемента переводят в некоторую перестановку из n элементов. При этом общее количество полученных перестановок равно n!. Остается показать, что среди них нет совпадающих. Если взять две сгенерированные перестановки внутри одной группы, то они будут различаться друг от друга позициями элементов {1,2,…,n−1} и, значит, не могут быть совпадающими. Если же взять перестановки из разных групп, то позиции элемента n в них будут разными и перестановки опять не могут быть совпадающими. Таким образом, все полученные n! перестановок различны.



Рис. 13.6. Генерирование перестановок методом вертикальной прогонки.

Описанный алгоритм вертикальной прогонки реализуется рекурсивной программой-функцией permut1(n):



Контрольный пример



**B.** **Метод последовательного замещения.** При S={1} имеем одну перестановку − 1. Если уже имеется последовательность перестановок из n−1 элемента {1,2,…,n−1}, то получить все перестановки из n элементов {1,2,…n} можно способом, который мы будем именовать как “метод последовательного замещения” элемента n. Суть его в следующем. Рассмотрим n идентичных экземпляров (групп) последовательностей перестановок из элементов {1,2,…,n−1}. В первом экземпляре, как и в методе вертикальной прогонки, в конец каждой перестановки поместим элемент n. В каждой перестановке второго экземпляра поменяем местами элементы n и 1. В каждой перестановке третьего экземпляра поменяем местами элементы n и 2 и т.д. Наконец, в последнем экземпляре поменяем местами элементы n и n−1. На рисунке 13.5 описанная процедура демонстрируется при переходе от n=3 к n=4. Нетрудно понять, что указанные действия любую перестановку из n−1 элемента переводят в некоторую перестановку из n элементов. При этом общее количество полученных перестановок равно n!. Остается показать, что среди них нет совпадающих. Если взять две сгенерированные перестановки внутри одной группы, то они обязательно будут различаться друг от друга расположением элементов на позициях от 1 до n−1 и, значит, не могут быть совпадающими. Если же взять перестановки из разных групп, то на последних позициях у них стоят разные элементы и перестановки опять не могут быть совпадающими. Таким образом, все полученные n! перестановок различны.



Рис. 13.5. Генерирование перестановок методом последовательного

замещения

Описанный алгоритм последовательного замещения реализуется рекурсивной программой-функцией permut2(n):



**С. Перестановки в антилексикографическом порядке.** На множестве P всех перестановок <x0,x1,…,xn-1> из элементов {1,2,…,n} определим два типа порядка.

Лексикографический порядок на P вводится следующим образом. Для

∀(<a0,a1,…,an−1>,<b0,b1,…,bn−1>∈P) (19)

положим

<a0,a1,…,an−1> <’ <b0,b1,…,bn−1> ⇔

∃(k ≥ 0) [(ak ≤ bk) & ∀(s < k) [(as = bs)]], (20)

где символ <’ использован в качестве знака лексикографического сравнения перестановок.

Заметим, что если вместо чисел 1,2,…,n использовать буквы с естественным порядком следования их в алфавите, то лексикографический порядок определяет стандартную последовательность, в которой слова длины n появляются в словаре. И это справедливо для алфавитов любых языков.

Аналогично вводится и антилексикографический порядок на P. Для (19) положим

<a0,a1,…,an−1> <” <b0,b1,…,bn−1> ⇔

∃(k ≤ n−1) [(ak > bk) & ∀(s > k) [(as = bs)]], (21)

где символ <” использован в качестве знака антилексикографического сравнения перестановок.

Построим рекурсивную программу-функцию, генерирующую перестановки элементов {1,2,…n} в антилексикографическом порядке. Соответствующий алгоритм может базироваться на следующих двух утверждениях, непосредственно вытекающих из определения 21.

Утверждение 1. Если множество перестановок P упорядочено антилексикографически, то начальная и конечная перестановки − это соответственно <1,2,…,n> и <n,n−1,…,1>.

Утверждение 2. Упорядоченная антилексикографически последовательность перестановок по значению последнего элемента в них может быть разбита на n блоков длины (n−1)!. При этом q-й блок на последнем месте имеет элемент равный (n−q+1), а первые n−1 позиций этого блока определяют последовательность перестановок множества {1,2,…,n}\{q} в антилексикографическом порядке.

Головная функция permut3(n) решает поставленную задачу. В ней по значению n для рекурсивной программы-функции permu(p) формируется начальная перестановка p. В permu(p) и проводятся все вычисления. На рис. 13.6 представлен результат выполнения permut3(n) при n=3 и n=4. Для n=4 полученные перестановки расположены по столбцам.



Рис. 13.6. Генерирование перестановок в антилексикографическом порядке

**D. Перестановки в лексикографическом порядке.** В предыдущем разделе мы ввели понятие лексикографического порядка. Алгоритм генерирования перестановок в таком порядке и соответствующие программы-функции могут быть построены на идеях, близких к тем, которые использовались при генерировании перестановок в антилексикографическом порядке. Поэтому здесь мы ограничимся лишь приведением соответствующих функций permut(p) и permut4(n) и представим полученный по ним результат вычислений для n=3 и n=4 (см. рис. 13.7). Для n=4 полученные перестановки расположены по столбцам.



Рис. 13.7. Генерирование перестановок в лексикографическом порядке

**E. Перестановки c одной транспозицией соседних элементов.** В этом пункте рассматривается алгоритм генерирования последовательности перестановок U из элементов множества {1,2,…,n} такой, что любая перестановка U, кроме начальной, получается из предыдущей перестановки одной транспозицией соседних элементов. Проиллюстрируем на примере идею соответствующего алгоритма, приписываемого Джонсону [] и Троттеру []. Предположим, что для элементов {1,2,…,n−1} уже построена требуемая последовательность перестановок U. Тогда из элементов {1,2,…,n} необходимая последовательность может быть построена перемещениями элемента n между начальной и конечной позициями каждой перестановки U. При этом перемещения должны производиться попеременно вперед и назад (n−1)! раз так, как это показано на рис. 13.8.

Рекурсивная программа-функция permut5(n) реализует этот, схематично описанный, алгоритм. На рис. 13.8 приведены полученные по permut5(n) перестановки для n=3 и n=4.



Рис. 13.8. Генерирование перестановок c транспозицией соседних элементов

В заключение автор выражает признательность профессорам. Ваграменко Я.А. и Добровольскому Н.М. за консультации и советы при написании пособия.

Литература

1. Кнут Д. Искусство программирования для ЭBM. Основные алгоритмы: т. 1, M.: Мир, 1976.
2. Кнут Д. Искусство программирования для ЭBM. Получисленные алгоритмы: т. 2, М.: Мир, 1977.
3. Кнут Д. Искусство программирования для ЭBM. Сортировка и поиск: т. 3, М.: Мир, 1978.
4. Лекции лауреатов премии Тьюринга. М.: Мир, 1985.
5. Бауэр Ф.Л., Гнац Р., Хилл У. Информатика. Задачи и решения. М.: Мир, 1978 г.
6. Бауэр Ф.Л., Гооз Г. Информатика. T. 1, М.: Мир, 1990 г.
7. Бауэр Ф.Л., Гооз Г. Информатика. T. 2, М.: Мир, 1990 г.
8. Ландау Э. Основы анализа. М.: изд. Иностранной литературы, 1947.

9. http://www-cs-staff.stanford.edu/~knuth/taocp.html#vol4-{volume4}).