

# ИНФОРМАТИКА

4

**Динамическое программирование**  
Вести из будущего, или Хит ЕГЭ-2012?

34

**Методика разноуровневого обучения**  
Под одну гребенку?  
На одну вершину!

55

**1+1=?**  
Традиционная тема:  
двоичный сумматор



На диске

**"От слов к делу"**  
Полные решения задач С4 прошлых лет

► Дорогие коллеги! Настроение у вас уже наверняка предновогоднее. Надеемся, что, получив этот номер, вы его отложите, отметите Новый год, хорошенько отдохнете, а уже затем вернетесь к работе и серьезному чтению. А там уже и первый январский номер придет — кому-то по почте, кому-то прямо 1 января в Личный кабинет. С праздником! Здоровья вам, удачи, вдохновения и желаний работать — желаний вести уроки, биться с задачами, возиться с детьми.

- 4** ПРОФИЛЬ  
► Алгоритмизация и программирование
- 34** МЕТОДИКА  
► Методика обучения информатике в разноуровневых группах (10–11-е классы)
- 44** ВНЕКЛАССНАЯ РАБОТА  
► Вопросы по информатике для конкурсов “Что? Где? Когда?” и “Брейн-ринг”
- 47** ИНФОРМАЦИЯ  
► Годовая подшивка газеты “Информатика” на компакт-диске за 2010 год
- 48** ЗАНИМАТЕЛЬНЫЕ МАТЕРИАЛЫ ДЛЯ ПЫТЛИВЫХ УЧЕНИКОВ И ИХ ТАЛАНТЛИВЫХ УЧИТЕЛЕЙ  
► “В мир информатики” № 171
- 63** ИНФОРМАЦИЯ  
► Педагогический университет “Первое сентября” предлагает дистанционные курсы для учителей информатики



## ЭЛЕКТРОННЫЕ МАТЕРИАЛЫ:

- Исходные коды программ
- Методические материалы
- Презентации к материалам номера
- Д.М. Златопольский. Полные решения задач С4 ЕГЭ прошлых лет

Уважаемые подписчики бумажной версии журнала “Информатика”!

В течение второго полугодия 2011 г. вы можете получать и электронную версию нашего журнала в Личном кабинете. Электронная версия в Личный кабинет всегда доставляется точно в срок — 1-го числа каждого месяца.

1. Зайдите на интернет-сайт [www.1september.ru](http://www.1september.ru).
2. Зарегистрируйте Личный кабинет (если у вас его еще нет).
3. В Личном кабинете в разделе “Издания/Коды доступа” введите код SE-00332-68580. Номера будут доступны в разделе “Издания/Получение”.

## ИНФОРМАТИКА

ПОДПИСНЫЕ ИНДЕКСЫ: по каталогу “Роспечати”: 32291 (бумажная версия), 19179 (электронная версия); “Почта России”: 79066 (бумажная версия), 12684 (электронная версия)

<http://inf.1september.ru>

Учебно-методический журнал для учителей информатики  
*Основан в 1995 г.*  
Выходит один раз в месяц

## РЕДАКЦИЯ:

гл. редактор С.Л. Островский  
редакторы

Е.В. Андреева,  
Д.М. Златопольский  
(редактор вкладки  
“В мир информатики”)

Дизайн макета И.Е. Лукьянов  
верстка Н.И. Пронская

корректор Е.Л. Володина  
секретарь Н.П. Медведева

Фото: фотобанк Shutterstock  
Журнал распространяется по подписке

Цена свободная

Тираж 3000 экз.

Тел. редакции: (499) 249-48-96

E-mail: [inf@1september.ru](mailto:inf@1september.ru)

<http://inf.1september.ru>

ИЗДАТЕЛЬСКИЙ ДОМ  
“ПЕРВОЕ СЕНТЯБРЯ”

Главный редактор:  
Артем Соловейчик  
(генеральный директор)

Коммерческая деятельность:  
Константин Шмарковский  
(финансовый директор)

Развитие, IT  
и координация проектов:  
Сергей Островский  
(исполнительный директор)

Реклама и продвижение:  
Марк Сартан

Мультимедиа, конференции  
и техническое обеспечение:  
Павел Кузнецов

Производство:  
Станислав Савельев

Административно-  
хозяйственное обеспечение:  
Андрей Ушков

Главный художник:  
Иван Лукьянов

Педагогический университет:  
Валерия Арсланьян (ректор)

ГАЗЕТА  
ИЗДАТЕЛЬСКОГО ДОМА

Первое сентября – Е.Бирюкова

ЖУРНАЛЫ

ИЗДАТЕЛЬСКОГО ДОМА

Английский язык – А.Громушкина

Библиотека в школе – О.Громова

Биология – Н.Иванова

География – О.Коротова

Дошкольное

образование – М.Аромштам

Здоровье детей – Н.Сёмина

Информатика – С.Островский

Искусство – М.Сартан

История – А.Савельев

Классное руководство

и воспитание школьников –

О.Леонтьева

Литература – С.Волков

Математика – Л.Рослова

Начальная школа – М.Соловейчик

Немецкий язык – М.Бузоева

Русский язык – Л.Гончар

Спорт в школе – О.Леонтьева

Управление школой – Я.Сартан

Физика – Н.Козлова

Французский язык – Г.Чесновицкая

Химия – О.Блохина

Школьный психолог – И.Вачков

УЧРЕДИТЕЛЬ:  
ООО “ЧИСТЫЕ ПРУДЫ”

Зарегистрировано

ПИ № ФС77-44341

от 22.03.2011

в Министерстве РФ

по делам печати

Подписано в печать:

по графику 10.11.2011,

фактически 10.11.2011

Заказ №

Отпечатано в ОАО “Чеховский

полиграфический комбинат”

ул. Полиграфистов, д. 1,

Московская область,

г. Чехов, 142300

АДРЕС ИЗДАТЕЛЯ:

ул. Киевская, д. 24,

Москва, 121165

Тел./факс: (499) 249-31-38

Отдел рекламы:

(499) 249-98-70

<http://1september.ru>

ИЗДАТЕЛЬСКАЯ ПОДПИСКА:

Телефон: (499) 249-47-58

E-mail: [podpiska@1september.ru](mailto:podpiska@1september.ru)

Документооборот

Издательского дома

“Первое сентября” защищен

антивирусной программой

DrWeb

# Год уходящий

► Подходит к концу 2011 год. И хотя этот номер сдается в печать в конце ноября, уже можно начинать подводить итоги. Каждый день в околокомпьютерном мире происходит множество событий. Выделить какие-либо из них — дело заведомо неблагодарное и субъективное. Редакция “Информатики” представляет свой список запомнившихся событий. Он честно составлен по памяти. Конечно, большинство дат мы поставили “задним числом”, но события выписывали без шпаргалки — что запомнилось.

Египет полностью отключен от Интернета. Чем все закончилось, мы знаем	28.01
Android стал самой популярной мобильной ОС в мире	01.02
Крупный сбой Gmail	28.02
Стив Джобс представил iPad 2	02.03
Вышел Google Chrome 10.0	09.03
Вышел IE 9.0	15.03
Вышел Mozilla Firefox 4.0	23.03
В возрасте 84 лет скончался Пол Бэран, создатель первой компьютерной сети пакетной технологии передачи данных в сетях	28.03
Мощнейшая атака на ЖЖ	30.03
День учителя информатики на Московском педагогическом марафоне	01.04
В России вступил в силу закон об электронной подписи	08.04
Протокол ICQ открыли для альтернативных клиентов	11.04
Apple разъяснила назначение функции “слежки” в iPhone	05.05
Microsoft покупает Skype	10.05
Google представила первые серийные “хромбуки”	12.05
Стала достоянием гласности история “очернения” Google в СМИ за деньги Facebook	14.05
Google запустил кнопку “+1”	16.05
На Amazon электронные книги обошли по количеству продаж бумажные	20.05
Microsoft показала Mango — новую версию Windows Phone 7	25.05
Тотальные перебои в работе Skype	26.05
Российские студенты выиграли “золото” на чемпионате мира по программированию. Сам чемпионат выиграли китайцы	31.05
Google Apps прекратил поддержку устаревших браузеров. К таковым отнесен и IE 7	02.06
Стив Джобс анонсировал iOS5 и Apple iCloud	06.06
Вышел Google Chrome 12	08.06
Упал ICQ	11.06
Google запустила голосовой поиск на компьютерах	15.06
Вышел Mozilla Firefox 5.0	21.06
Исполнилось 100 лет IBM	21.06
У LinkedIn появилась русская версия	22.06
Ежемесячная аудитория Google превысила миллиард	22.06
Рамблер начал использовать поиск Яндекс	23.06
Минобрнауки открыло блог в ЖЖ	24.06

Открылась Google+, пока по приглашениям	28.06
Вышел Opera 11.50	28.06
Google изменила дизайн почты и календаря	01.07
Facebook и Skype запустили видеочат	07.07
SMS Мегафона нашлись в поисковой выдаче Яндекс	18.07
Livejournal упал из-за сбоя в дата-центре	25.07
Livejournal снова подвергся атаке	28.07
Mail.ru изменила дизайн почты	04.08
В Google+ появились on-line-игры	12.08
Вышел Mozilla Firefox 6.0	15.08
Крупный сбой Яндекс	19.08
Апорт перешел на поиск Яндекс	24.08
Стив Джобс ушел с поста генерального директора Apple	25.08
Google Docs снова обеспечила работу в off-line	01.09
Яндекс включил “бесконечную прокрутку” поисковых результатов	05.09
Стартовал проект “Школа цифрового века”	15.09
На Youtube появился видеоредактор с эффектами	15.09
Google Chrome 15.00	19.09
В Google+ открылась регистрация для всех желающих	20.09
Вышел Mozilla Firefox 7.0	28.09
Firefox вслед за Chrome изъял http из адресной строки	28.09
Apple представила iPhone 4S. С ним можно поговорить...	04.10
Скончался Стив Джобс	06.10
Предзаказы на книгу о Джобсе на Amazon выросли на 41 000%	06.10
Глобальный сбой Blackberry	10.10
Google представила язык Dart — альтернативу Javascript	10.10
В ICQ можно входить под одним аккаунтом с разных устройств	12.10
Скончался Деннис Ритчи, создатель языка программирования Си	13.10
Kindle Fire — очередной “убийца iPad” — поступил в продажу	14.10
Скончался Джон Маккарти, создатель языка программирования Лисп	25.10
Facebook приступила к строительству дата-центра у Полярного круга	27.10
Закрылся сайт Вебпланета	19.11
В популярных моделях смартфонов появилась поддержка ГЛОНАСС	19.11



HELLO  
WORLD!

## Алгоритмизация и программирование

### Целочисленные алгоритмы

К.Ю. Поляков,  
А.П. Шестаков,  
Е.А. Еремин

► Во многих задачах все исходные данные и необходимые результаты — целые числа. При этом всегда желательно, чтобы все промежуточные вычисления тоже выполнялись с только целыми числами. На это есть по крайней мере две причины:

- процессор, как правило, выполняет операции с целыми числами значительно быстрее, чем с вещественными;
- целые числа всегда точно представляются в памяти компьютера, и вычисления с ними также выполняются без ошибок (если, конечно, не происходит переполнение разрядной сетки).

### Решето Эратосфена

Простые числа широко используются во многих прикладных задачах, например, при шифровании с помощью алгоритма RSA. Основные задачи при работе с простыми числами — это проверка числа на простоту и нахождение всех простых чисел в заданном интервале.

Пусть задано некоторое натуральное число  $N$  и требуется найти все простые

числа в интервале от 1 до  $N$ . Самое простое (но неэффективное) решение этой задачи состоит в том, что в цикле перебираются все числа от 1 до  $N$ , и каждое из них отдельно проверяется на простоту. Например, можно проверить, есть ли у числа  $k$  делители в интервале от 2 до  $\sqrt{k}$ . Если ни одного такого делителя нет, то число  $k$  простое.

Описанный метод при больших  $N$  работает очень медленно. Греческий математик Эратосфен Киренский (275–194 гг. до н. э.) предложил другой алгоритм, который работает намного быстрее:

- 1) выписать все числа от 2 до  $N$ ;
- 2) начать с  $k = 2$ ;
- 3) вычеркнуть все числа, кратные  $k$  ( $2k, 3k, 4k$  и т.д.);
- 4) найти следующее не вычеркнутое число и присвоить его переменной  $k$ ;
- 5) повторять шаги 3 и 4, пока  $k < N$ .

Покажем работу алгоритма при  $N = 16$ :

2	3	4	5	6	7	8	9	10	11
		12	13	14	15	16			

Рис. 1

Первое не вычеркнутое число — 2, поэтому вычеркиваем все четные числа:

```

2 3 4 5 6 7 8
9 10 11 12 13 14 15 16

```

Рис. 2

Далее вычеркиваем все числа, кратные 3:

```

2 3 4 5 6 7 8
9 10 11 12 13 14 15 16

```

Рис. 3

А все числа, кратные 5 и 7, уже вычеркнуты. Таким образом, получены простые числа 2, 3, 5, 7, 11 и 13.

Классический алгоритм можно улучшить, уменьшив количество операций. Заметьте, что при вычеркивании чисел, кратных трем, нам не пришлось вычеркивать число 6, так как оно уже было вычеркнуто. Кроме того, все числа, кратные 5 и 7, к последнему шагу тоже оказались вычеркнуты.

Предположим, что мы хотим вычеркнуть все числа, кратные некоторому  $k$ , например,  $k = 5$ . При этом числа  $2k$ ,  $3k$  и  $4k$  уже были вычеркнуты на предыдущих шагах, поэтому нужно начать не с  $2k$ , а с  $k^2$ . Тогда получается, что при  $k^2 > N$  вычеркивать уже будет нечего, что мы и увидели в примере. Поэтому можно использовать улучшенный алгоритм:

- 1) выписать все числа от 2 до  $N$ ;
- 2) начать с  $k = 2$ ;
- 3) вычеркнуть все числа, кратные  $k$ , начиная с  $k^2$ ;
- 4) найти следующее не вычеркнутое число и присвоить его переменной  $k$ ;
- 5) повторять шаги 3 и 4, пока  $k^2 \leq N$ .

Чтобы составить программу, нужно определить, что значит “выписать все числа” и “вычеркнуть число”. Один из возможных вариантов хранения данных — массив логических величин с индексами от 2 до  $N$ . Как и в учебнике 10-го класса, будем использовать два языка: алгоритмический язык и Паскаль.

Объявление переменных в программе будет выглядеть так (для  $N = 100$ ):

```

цел i, k, N = 100
логтаб A[2..N]

const N = 100;
var i, k: integer;
    A: array[2..N] of boolean;

```

Если число  $i$  не вычеркнуто, будем хранить в элементе массива  $A[i]$  истинное значение, если вычеркнуто — ложное. В самом начале нужно заполнить массив истинными значениями:

```

нц для i от 2 до N
    A[i] := да
кц

for i:= 2 to N do
    A[i] := True;

```

В основном цикле выполняется описанный выше алгоритм:

```

k:= 2
нц пока k*k <= N
    если A[k] то
        i:= k*k
        нц пока i <= N
            A[i] := нет
            i:= i + k
        кц
    все
    k:= k + 1
кц

```

```

k:= 2;
while k*k <= N do begin
    if A[k] then begin
        i:= k*k;
        while i <= N do begin
            A[i] := False;
            i:= i + k
        end
    end;
    k:= k + 1
end;

```

Обратите внимание, что, для того чтобы вообще не применять вещественную арифметику, мы заменили условие  $k \leq \sqrt{N}$  на равносильное условие  $k^2 \leq N$ , в котором используются только целые числа.

После завершения этого цикла не вычеркнутыми остались только простые числа, для них соответствующий элемент массива содержит истинное значение. Эти числа нужно вывести на экран:

```

нц для i от 2 до N
    если A[i] то
        вывод i, нс
    все
кц

for i:= 2 to N do
    if A[i] then
        writeln(i);

```

### “Длинные” числа

Современные алгоритмы шифрования используют достаточно длинные ключи, которые представляют собой числа длиной 256 бит и больше. С ними нужно выполнять разные операции: складывать, умножать, находить остаток от деления. Вопрос состоит в том, как хранить такие числа в памяти, где для хранения целых чисел отводятся ячейки значительно меньших размеров (обычно до 64 бит). Ответ достаточно очевиден: нужно разбить “длинное” число на части так, чтобы использовать несколько ячеек памяти.

“Длинные” числа — это числа, которые не помещаются в одну переменную одного из стандартных типов данных языка программирования. Алгоритмы работы с “длинными” числами называют “длинной арифметикой”.

Для хранения “длинного” числа удобно использовать массив целых чисел. Например, число 12345678 можно записать в массив с индексами от 0 до 9 таким образом:

	0	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	0	0

Такой способ имеет ряд недостатков:

1) нужно где-то хранить длину числа, иначе числа 12345678, 123456780 и 1234567800 будет невозможно различить;

2) неудобно выполнять арифметические операции, которые начинаются с младшего разряда;

3) память расходуется неэкономно, потому что в одном элементе массива хранится только один разряд — число от 0 до 9.

Чтобы избавиться от первых двух проблем, достаточно “развернуть” массив наоборот, так чтобы младший разряд находился в  $A[0]$ . В этом случае на рисунке удобно применять обратный порядок элементов:

	9	8	7	6	5	4	3	2	1	0
A	0	0	1	2	3	4	5	6	7	8

Теперь нужно найти более экономичный способ хранения “длинного” числа. Например, разместим в одной ячейке массива три разряда числа, начиная справа:

	9	8	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	0	12	345	678

Здесь использовано равенство

$$12345678 = 12 \cdot 1000^2 + 345 \cdot 1000^1 + 678 \cdot 1000^0.$$

Фактически мы представили исходное число в системе счисления с основанием 1000!

Сколько разрядов можно хранить в одной ячейке массива? Это зависит от ее размера. Например, если ячейка занимает 4 байта и число хранится со знаком, допустимый диапазон ее значений

$$\text{от } -2^{32} = -4\,294\,967\,296$$

$$\text{до } 2^{32} - 1 = 4\,294\,967\,295.$$

В такой ячейке можно хранить до 9 разрядов десятичного числа, то есть использовать систему счисления с основанием 1 000 000 000. Однако нужно учитывать, что с такими числами нужно будет выполнять арифметические операции, результат которых должен “помещаться” в ячейку памяти. Например, если надо умножать разряды этого числа на  $k \leq 100$  и в языке программирования нет 64-битных целочисленных типов данных, можно хранить не более 7 разрядов в ячейке.

**Задача 1.** Вычислить точно значение факториала  $100! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100$  (это число состоит более чем из сотни цифр и явно не помещается в одну ячейку памяти).

Для хранения “длинного” числа будем использовать целочисленный массив  $A$ . Определим необходимую длину массива. Заметим, что

$$1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100 < 100^{100}$$

Число  $100^{100}$  содержит 201 цифру, поэтому число  $100!$  содержит не более 200 цифр. Если в каждом

элементе массива записано 6 цифр, для хранения всего числа требуется не более 34 ячеек:

```
цел N = 33
целтаб A[0:N]
```

```
const N = 33;
var A: array[0..N] of integer;
```

Чтобы найти  $100!$ , нужно сначала присвоить “длинному” числу значение 1, а затем последовательно умножать его на все числа от 2 до 100. Запишем эту идею на псевдокоде, обозначив через  $A$  “длинное” число, находящееся в массиве  $A$ :

```
[A] := 1
нц для k от 2 до 100
  [A] := [A] * k
кц
```

Записать в “длинное” число единицу — это значит присвоить элементу  $A[0]$  значение 1, а в остальные ячейки записать нули:

```
A[0] := 1
нц для i от 1 до N
  A[i] := 0
кц
```

```
A[0] := 1;
for i:=1 to N do
  A[i] := 0;
```

Таким образом, остается научиться умножать “длинное” число на “короткое” ( $k \leq 100$ ). “Короткими” обычно называют числа, которые помещаются в один из стандартных типов данных. Попробуем выполнить такое умножение на примере. Предположим, что в каждой ячейке массива хранится 6 цифр “длинного” числа, то есть используется система счисления с основанием  $d = 1000000$ . Тогда число  $A = 12345678901734567$  хранится в трех ячейках

	2	1	0
A	1234	568901	734567

Пусть  $k = 3$ . Начинаем умножать с младшего разряда:  $734567 \cdot 3 = 2203701$ . В нулевом разряде может находиться только 6 цифр, значит, старшая двойка перейдет в перенос в следующий разряд. В программе для выделения переноса  $r$  можно использовать деление на основание системы счисления  $d$  с отбрасыванием остатка. Сам остаток — это то, что остается в текущем разряде. Поэтому получаем

```
s := A[0] * k
A[0] := mod(s, d)
r := div(s, d)
```

```
s := A[0] * k;
A[0] := s mod d;
r := s div d;
```

Для следующего разряда будет все то же самое, только в первой операции к произведению нужно добавить перенос из предыдущего разряда, кото-

рый был записан в переменную  $r$ . Приняв в самом начале  $r = 0$ , запишем умножение “длинного” числа на “короткое” в виде цикла по всем элементам массива, от  $A[0]$  до  $A[N]$ :

```
r:=0
нц для i от 0 до N
  s:=A[i]*k+r
  A[i]:=mod(s,d)
  r:=div(s,d)
кц
```

```
r:=0;
for i:=0 to N do begin
  s:=A[i]*k+r;
  A[i]:=s mod d;
  r:=s div d;
end;
```

В свою очередь, эти действия нужно выполнить в другом (внешнем) цикле для всех  $k$  от 2 до 100:

```
нц для k от 2 до 100
  ...
кц
```

```
for k:=2 to 100 do
  ...
end;
```

После этого в массиве  $A$  будет находиться искомое значение  $100!$ , остается вывести его на экран. Нужно учесть, что в каждой ячейке хранятся 6 цифр, поэтому в массиве

	2	1	0
A	1	2	3

хранится значение 1000002000003, а не 123. Кроме того, старшие нулевые разряды выводить на экран не надо. Поэтому при выводе требуется

- 1) найти первый (старший) ненулевой разряд числа;
- 2) вывести это значение без лидирующих нулей;
- 3) вывести все следующие разряды, добавляя лидирующие нули до 6 цифр.

Поскольку мы знаем, что число не равно нулю, старший ненулевой разряд можно найти в таком цикле<sup>1</sup>:

```
i:=N
нц пока A[i] = 0
  i:=i-1
кц
```

```
i:=N;
while A[i] = 0 do
  i:=i-1;
```

Старший разряд выводим обычным образом (без лидирующих нулей):

```
вывод A[i]
write(A[i]);
```

<sup>1</sup> Подумайте, что изменится, если выводимое число может быть нулевым.

Для остальных разрядов будем использовать специальную процедуру **Write6**:

```
нц для k от i-1 до 0 шаг -1
  Write6(A[k])
кц
```

```
for k:=i-1 downto 0 do
  Write6(A[k]);
```

Эта процедура последовательно выводит цифры десятичного числа, начиная с сотен тысяч и кончая единицами:

```
алг Write6(цел x)
нач
  цел M, xx
  xx:=x
  M:=100000
  нц пока M > 0
    вывод div(xx, M)
    xx:=mod(xx, M)
    M:=div(M, 10)
  кц
кон
```

```
procedure Write6(x: integer);
var M: integer;
begin
  M:=100000;
  while M > 0 do begin
    write(x div M);
    x:=x mod M;
    M:=M div 10;
  end;
end;
```

Для того чтобы разобраться, как она работает, выполните “ручную прокрутку” при различных значениях  $x$  (например, возьмите  $x = 1$ ,  $x = 123$  и  $x = 123456$ ).

### Контрольные вопросы

1. Какие преимущества и недостатки имеет алгоритм “решето Эратосфена” в сравнении с проверкой каждого числа на простоту?
2. Что такое “длинные” числа?
3. В каких случаях необходимо применять “длинную арифметику”?
4. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
5. Можно ли использовать для хранения “длинного” числа символьную строку? Какие проблемы при этом могут возникнуть?
6. Почему неудобно хранить “длинное” число, записывая первую значащую цифру в начало массива?
7. Почему неэкономично хранить по одной цифре в каждом элементе массива?
8. Сколько разрядов числа можно хранить в одной 16-битной ячейке?
9. Объясните, какие проблемы возникают при выводе “длинного” числа. Как их можно решать?
10. Объясните работу процедуры **Write6**.



## Задачи

1. Докажите, что если у числа  $k$  нет ни одного делителя в интервале от 2 до  $\sqrt{k}$ , то оно простое.

2. Напишите две программы, которые находят все простые числа от 1 до  $N$  двумя разными способами:

1) проверкой каждого числа из этого интервала на простоту;

2) используя “решето Эратосфена”.

Сравните число шагов цикла (или время работы) этих программ для разных значений  $N$ . Постройте для каждого варианта зависимость количества шагов от  $N$ , сделайте выводы о сложности алгоритмов.

3. Докажите, что в приведенной программе вычисления  $100!$  не будет переполнения при использовании 32-битных целых переменных.

4. Можно ли в той же программе в одной ячейке массива хранить 9 цифр “длинного” числа?

5. Без использования программы определите, сколько нулей стоит в конце числа  $100!$

6. Соберите всю программу и вычислите  $100!$ . Сколько цифр входит в это число?

7. Оформите вывод “длинного” числа на экран в виде отдельной процедуры. Учтите, что число может быть нулевым.

8. \*Придумайте другой способ вывода “длинного” числа, использующий символьные строки.

9. Напишите процедуру для ввода “длинных” чисел из файла.

10. Напишите процедуры для сложения и вычитания “длинных” чисел.

11. \*Напишите процедуры для умножения и деления “длинных” чисел.

12. \*\*Напишите процедуру для извлечения квадратного корня из “длинного” числа.

## Структуры (записи)

## Зачем нужны структуры?

Представим себе базу данных библиотеки, в которой хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляров и т.д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но в массиве используются элементы одного типа, тогда как информация о книгах разнородна, она содержит целые числа и символьные строки разной длины. Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т.д.), так чтобы  $i$ -й элемент каждого массива относился к книге с номером  $i$ . Но такой подход оказывается слишком неудобен и ненадежен. Например, при сортировке нужно переставлять элементы всех массивов (отдельно!) и можно легко ошибиться и нарушить связь данных.

Возникает естественная идея — объединить все данные, относящиеся к книге, в единый блок памяти, который в программировании называется структурой.

**Структура** — это тип данных, который может включать в себя несколько *полей* — элементов разных типов (в том числе и другие структуры).

В Паскале структуры по традиции называют записями (англ. *record* — “запись”). В этой главе мы будем использовать возможности свободнораспространяемого компилятора *FreePascal*.

## Объявление структур

Как и любые переменные, структуры необходимо объявлять. До этого мы работали с простыми типами данных (целыми, вещественными, логическими и символьными), а также с массивами этих типов. Вы знаете, что при объявлении переменных и массивов указывается их тип, поэтому, для того чтобы работать со структурами, нужно ввести новый тип данных.

Построим структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только<sup>2</sup>

- фамилию автора (строка не более 40 символов);
- название книги (строка не более 80 символов);
- имеющееся в библиотеке количество экземпляров (целое число).

Объявление такого *составного* типа имеет вид:

```
type
  TBook = record
    author: string[40]; {автор, строка}
    title: string[80]; {название, строка}
    count: integer; {количество, целое}
  end;
```

Объявления типов данных начинаются с ключевого слова **type** (от англ. “тип”) и располагаются выше блока объявления переменных. Имя нового типа — **TBook** — это удобное сокращение от английских слов *Type Book* (*тип книга*), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования. Слово **record** означает, что этот тип данных — структура (запись); далее перечисляются поля и указывается их тип. Объявление структуры заканчивается ключевым словом **end**.

Обратите внимание, что для строк **author** и **title** указан максимальный размер. Это сделано для того, чтобы точно определить, сколько места нужно выделить на них в памяти.

В результате такого объявления никаких структур в памяти не создается: мы просто описали новый тип данных, чтобы транслятор знал, что делать, если мы захотим его использовать.

Теперь можно использовать тип **TBook** так же, как и простые типы, для объявления переменных и массивов:

```
const N = 100;
var B: TBook;
    Books: array[1..N] of TBook;
```

Здесь введена переменная **B** типа **TBook** и массив **Books**, состоящий из элементов того же типа.

<sup>2</sup> Конечно, в реальной ситуации данных больше, но принцип не меняется.



Иногда бывает нужно определить размер одной структуры. Это можно сделать с помощью стандартной функции `sizeof`, которой можно передать имя типа, а также переменную или массив:

```
writeln(sizeof(TBook));
writeln(sizeof(B));
writeln(sizeof(Books));
```

Первые две команды выведут на экран размер одной структуры (124 байта), а последняя — размер выделенного массива из 100 структур. Размер структуры вызывает некоторые вопросы: каждый элемент строки занимает 1 байт, а целое число — 2 байта, поэтому простой подсчет дает значение  $40 + 80 + 2 = 122$ . Откуда появились еще 2 байта? Дело в том, для строка `author` из 40 символов фактически занимает 41 байт, а строковое поле `title` — 81 байт: один дополнительный байт расходуется на хранение размера строки.

### Обращение к полю структуры

Для того чтобы работать не со всей структурой, а с отдельными полями, используют так называемую *точечную нотацию*, разделяя точкой имя структуры и имя поля. Например, `B.author` обозначает “поле `author` структуры `B`”, а `Books[5].count` — “поле `count` элемента массива `Books[5]`”. Например, для определения размера полей в байтах можно снова использовать функцию `sizeof`:

```
writeln(sizeof(B.author));
writeln(sizeof(B.title));
writeln(sizeof(B.count));
```

— и мы увидим на экране числа 41, 81 и 2.

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить их с клавиатуры (или из файла):

```
readln(B.author);
readln(B.title);
readln(B.count);
```

присваивать новые значения:

```
B.author:= 'Пушкин А.С.';
B.title:= 'Полтава';
B.count:= 1;
```

использовать при обработке данных:

```
p:= Pos(' ', B.author);
fam:= Copy(B.author, 1, p-1); {только фамилия}
B.count:= B.count - 1; {одну книгу взяли}
if B.count = 0 then
  writeln('Этих книг больше нет!');
```

и выводить на экран:

```
writeln(sizeof(B.author));
writeln(sizeof(B.title));
writeln(sizeof(B.count));
```

### Работа с файлами

В программах, работающих с базами данных, необходимо читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля одной структуры в одну строчку и разделяя

их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет выполнять файловые операции проще и надежнее (с меньшей вероятностью ошибки). Для этого нужно использовать файлы специального типа, которые называются *типизированными*. Все записываемые в них данные должны иметь одинаковый тип. В отличие от текстовых файлов данные в типизированных файлах хранятся во внутреннем формате, то есть так, как они представлены в памяти компьютера во время работы программы. Например, можно сделать файл целых чисел или логических величин.

В данном случае нас интересует файл структур типа `TBook`, так что файловая переменная `F` для работы с типизированным файлом должна быть объявлена так:

```
var F: file of TBook;
```

Запись структуры в файл выполняется стандартным способом:

```
Assign(F, 'books.dat');
Rewrite(F);
B.author:= 'Тургенев И.С.';
B.title:= 'Муму';
B.count:= 2;
write(F, B);
Close(F);
```

Напомним, что процедура `Assign` связывает файловую переменную с файлом на диске, процедура `Rewrite` открывает файл на запись, а процедура `Close` — закрывает (освобождает) файл.

Процедура `write`, определив, что файловая переменная `F` связана с типизированным файлом структур, записывает в файл одну структуру во внутреннем формате. При попытке передать этой процедуре переменную другого типа произойдет ошибка и аварийный останов программы.

С помощью цикла можно записать в файл весь массив структур:

```
for i:=1 to N do
  write(F, Books[i]);
```

Прочитать из файла одну структуру и вывести ее поля на экран можно следующим образом:

```
Assign(F, 'books.dat');
Reset(F);
Read(F, B);
writeln(B.author, ' ', B.title, ' ', B.count);
Close(F);
```

Функция `read`, получив ссылку `F` на типизированный файл, может принимать в качестве следующих параметров только структуры типа `TBook`.

Если заранее известно, сколько структур записано в файле, при чтении их в массив можно применить цикл с переменной:

```
for i:=1 to N do
  read(F, Books[i]);
```

Если же число структур неизвестно, нужно выполнять чтение до тех пор, пока файл не закончится, то есть функция `Eof` не вернет истинное значение:

```
i:= 0;
while not eof(F) do begin
  i:= i + 1;
  Read(F, Books[i]);
end;
```

Здесь целая переменная *i* играет роль счетчика: в ней на каждом шаге записано количество фактически прочитанных структур.

### Сортировка

Для сортировки массива структур можно использовать те же методы, что и для массива простых переменных. Разница только в том, по какому критерию сортировать. Структуры обычно сортируют по возрастанию или убыванию одного из полей, которое называют ключевым полем, или ключом, хотя можно, конечно, использовать и сложные условия, зависящие от нескольких полей.

Отсортируем массив **Books** (типа **TBook**) по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле **author**. Предположим, что фамилия состоит из одного слова, а за ней через пробел следуют инициалы. Тогда сортировка методом “пузырька” выглядит так:

```
for i:=1 to N-1 do
  for j:=N-1 downto i do
    if Books[j+1].author <
      Books[j].author then
      begin
        B:=Books[j]; Books[j]:=Books[j+1];
        Books[j+1]:=B;
      end;
```

Здесь *i* и *j* — целочисленные переменные, а **B** — вспомогательная структура типа **TBook**.

Как вы знаете из курса 10-го класса, при сравнении двух символьных строк они рассматриваются посимвольно до тех пор, пока не будут найдены первые отличающиеся символы. Далее сравниваются коды этих символов по кодовой таблице. Так как код пробела меньше, чем код любой русской (и латинской) буквы, строка с фамилией “Волк” окажется выше в отсортированном списке, чем строка с более длинной фамилией “Волков”, даже с учетом того, что после фамилии есть инициалы. Если фамилии одинаковы, сортировка происходит по первой букве инициалов, затем — по второй букве.

Возможно, что структуры нужно отсортировать так, чтобы не перемещать их в памяти. Например, они очень большие, и многократное копирование целых структур занимает много времени. Или по каким-то другим причинам перемещать структуры нельзя. При таком ограничении нужно вывести на экран или в файл отсортированный список. В этом случае применяют “сортировку по указателям”, в которой используется дополнительный массив переменных специального типа — указателей.

**Указатель** — это переменная, в которой можно сохранить адрес любой переменной заданного типа.

То есть содержимое указателя — это адрес памяти. Чтобы избежать случайных ошибок, каждому указателю при объявлении присваивается тип данных, адреса которых он может хранить. Например, объявление

```
type pBook = ^TBook;
```

вводит новый тип данных — указатель на структуру типа **TBook**. Адреса переменных других типов в такой указатель записывать нельзя. Имя типа-указателя удобно начинать с буквы **p**, от английского слова *pointer* — “указатель”.

Для сортировки массива **Books** нужно разместить в памяти массив таких указателей и одну вспомогательную переменную, которая будет использована при сортировке:

```
var p: array[1..N] of pBook;
    p1: pBook;
```

Следующий этап — расставить указатели так, чтобы *i*-й указатель был связан с *i*-й структурой из массива **Books**:

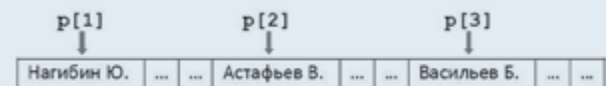
```
for i:=1 to N do p[i]:= @Books[i];
```

Знак “@” обозначает операцию взятия адреса, то есть в указатель записывается адрес структуры.

Для того чтобы от указателя перейти к объекту, на который он ссылается, используют оператор **^**. Например, в нашем случае (после показанной выше начальной установки указателей) запись **p[i]^** обозначает то же самое, что и **Books[i]**, а **p[i]^ .title** — то же самое, что и **Books[i].title**.

Теперь можно перейти к сортировке. Рассмотрим идею на примере массива из трех структур. Сначала указатели стоят по порядку (рис. 3а). В результате сортировки нужно переставить их так, чтобы **p[1]** указывал на первую структуру в отсортированном списке, **p[2]** — на вторую и т.д. (рис. 3б).

а)



б)

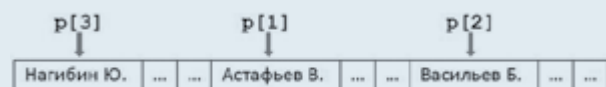


Рис. 3

Обратите внимание, что при этом сами структуры в памяти не перемещались.

Теперь можно вывести отсортированные данные, обращаясь к ним через указатели (а не через массив **Books**):

```
for i:=1 to N do
  writeln(p[i]^ .author, '; ',
          p[i]^ .title, '; ',
          p[i]^ .count);
```

### Контрольные вопросы

1. Что такое структура? В чем ее отличие от массива?
2. В каких случаях использование структур дает преимущества? Какие именно?
3. Как объявляется новый тип данных в Паскале? Выделяется ли при этом память?
4. Как обращаются к полю структуры? Расскажи-те о точечной нотации.
5. Как определить, сколько байт памяти выделяется на структуру?

6. Что такое типизированный файл? Чем он отличается от текстового?
7. Как работать с типизированными файлами?
8. Как можно сортировать структуры?
9. В каких случаях при сортировке желательно не перемещать структуры в памяти?
10. Что такое указатель?
11. Как записать в указатель адрес переменной?
12. Как обращаться к полям структуры через указатель?
13. Как используются указатели при сортировке?

## Задачи

1. Опишите структуру, в которой хранится информация о
  - а) видеозаписи;
  - б) сотруднике фирмы “Рога и Копыта”;
  - в) самолете;
  - г) породистой собаке.
2. Постройте программу, которая работает с базой данных в виде типизированного файла. Ваша СУБД (система управления базой данных) должна иметь следующие возможности:
  - а) просмотр записей;
  - б) добавление записей;
  - в) удаление записей;
  - г) сортировка по одному из полей (через указатели).

## Динамические массивы

### Что это такое?

Когда мы объявляем массив, место для него выделяется во время трансляции, то есть до выполнения программы. Такой массив называется *статическим*. В то же время, иногда размер данных заранее неизвестен.

Например, в файле записан массив чисел, которые нужно отсортировать. Их количество неизвестно, но известно, что такой массив помещается в оперативную память. В этом случае есть два варианта: 1) выделить заранее максимально большой блок памяти, и 2) выделять память уже *во время выполнения программы* (то есть динамически), когда стал известен необходимый размер массива.

Другой пример — задача составления *алфавитно-частотного словаря*. В файле находится некоторый текст. Нужно вывести в другой файл все различные слова, которые встречаются в файле, и количество этих слов в тексте. Здесь проблема состоит в том, что нужный размер массива можно узнать только тогда, когда все различные слова будут найдены и, таким образом, задача решена. Поэтому нужно сделать так, чтобы массив мог “расширяться” в ходе работы программы.

Эти задачи приводят к понятию “динамических структур данных”, которые позволяют во время выполнения программы:

- создавать новые объекты в памяти;
- изменять их размер;
- удалять их из памяти, когда они не нужны.

Память под эти объекты выделяется в специальной области, которую обычно называют “*кучей*” (англ. *heap*).

### Размещение в памяти

**Задача 2.** Ввести с клавиатуры целое значение *N*, затем — *N* целых чисел, и вывести на экран эти числа в порядке возрастания.

Поскольку для сортировки все числа необходимо удерживать в памяти, нужно заводить массив, в который будут записаны все элементы. Поэтому алгоритм решения задачи на псевдокоде выглядит так:

```
прочитать данные из файла в массив
отсортировать их по возрастанию
вывести массив на экран
```

Все эти операции для обычных массивов подробно рассматривались в курсе 10-го класса, поэтому здесь мы остановимся только на главной проблеме: как разместить в памяти массив, размер которого до выполнения программы неизвестен?

Для подобных случаев в версии языка Паскаль, которая поддерживается в среде *FreePascal*, существуют динамические массивы, которые объявляются без указания размера:

```
var A: array of integer;
```

Использовать сразу такой массив нельзя, поскольку его размер неизвестен. Попытка обращения к элементу, скажем, **A[1]** вызывает ошибку и аварийный останов программы.

Когда значение переменной *N* введено, можно фактически разместить массив в памяти, используя процедуру **SetLength** (англ. *set length* — “установить длину”):

```
SetLength(A, N);
```

Далее можно использовать массив **A** так же, как и обычный (статический) массив. Остается один вопрос: в каком диапазоне находятся его индексы?

Вы помните, что границы изменения индексов обычного (статического) массива задаются при его объявлении, причем начальный индекс может быть любым. Индексы динамического массива *всегда начинаются с нуля*, так что к начальному элементу нужно обращаться как **A[0]**, а к последнему — как **A[N-1]**. Например, чтение данных с клавиатуры выполняется в цикле:

```
for i:=0 to N-1 do read(A[i]);
```

Кроме того, массив “знает” свою длину, которая вычисляется с помощью стандартной функции **Length** (англ. *длина*), и максимальный индекс, который возвращает функция **High** (англ. *высокий*). Поэтому предыдущий цикл можно заменить на такой:

```
for i:=0 to High(A) do read(A[i]);
```

Размер массива (количество элементов в нем) можно вычислить как **Length(A)** или **High(A)+1**.

Как только массив стал не нужен, можно удалить его из памяти, установив нулевую длину:

```
SetLength(A, 0);
```

Для такого (удаленного) массива нулевой длины функция **Length(A)** вернет значение 0.

Таким же образом можно работать и с динамическими матрицами. Они объявляются как “массив массивов”:

```
var A: array of array of integer;
```

Для определения ее размеров в процедуре `SetLength` нужно указать два параметра — количество строк и количество столбцов:

```
SetLength(A, 4, 3);
```

Функция `High` возвращает максимальный индекс строки (минимальный индекс всегда равен 0):

```
writeln(High(A)); { = 3 }
```

Для определения границ изменения второго индекса (максимального номера столбца) нужно вызывать эту функцию для отдельной строки:

```
writeln(High(A[0])); { = 2 }
```

### Использование в подпрограммах

Динамические массивы можно передавать как параметры подпрограмм (процедур и функций). Например, процедуру для вывода на экран целочисленного массива можно написать так:

```
procedure printArray(X: array of integer);
begin
  for i:=0 to High(X) do write(X[i], ' ');
end;
```

Динамические массивы можно передать как изменяемые параметры (с помощью ключевого слова `var`). В этом случае все изменения, сделанные в подпрограмме, применяются к массиву, переданному вызывающей программой, а не к его копии.

### Расширение массива

**Задача 3.** С клавиатуры вводятся натуральные числа, ввод заканчивается числом 0. Нужно вывести на экран эти числа в порядке возрастания.

Как и в предыдущей задаче, для сортировки нужно предварительно сохранить все числа в оперативной памяти (в массиве). Но проблема в том, что размер этого массива становится известен только тогда, когда будут введены все числа. Что же делать?

В первую очередь приходит в голову такой вариант: при вводе каждого следующего ненулевого числа расширять массив на один элемент и записывать введенное число в последний элемент массива:

```
read(x);
while x <> 0 do begin
  SetLength(A, Length(A)+1);
  A[High(A)] := x;
  read(x)
end;
```

Здесь `x` — это целая переменная. К счастью, при таком расширении массива значения всех существующих элементов сохраняются.

Чем плох такой подход? Дело в том, что память в куче выделяется блоками. Поэтому при каждом увеличении длины массива последовательно выполняются три операции:

- 1) выделение блока памяти нового размера;
- 2) копирование в этот блок всех “старых” элементов;
- 3) удаление “старого” блока памяти из кучи.

Видно, что “накладные расходы” очень велики, то есть мы заставляем компьютер делать слишком много вспомогательной работы.

Ситуацию можно немного исправить, если увеличивать массив не каждый раз, а скажем, после каждых 10 введенных элементов. То есть, когда все свободные элементы массива заполнены, к нему добавляется еще 10 новых ячеек. При этом нужно считать фактическое количество записанных в массив значений, потому что определить их, как в предыдущей программе, через функцию `Length(A)` будет невозможно:

```
N:=0;
read(x);
while x <> 0 do begin
  if N > High(A) then
    SetLength(A, Length(A)+10);
  A[N] := x;
  N:=N+1;
  read(x)
end;
```

Здесь целая переменная `N` — это счетчик введенных чисел.

Теперь, когда все числа записаны в массив, можно отсортировать их любым известным методом, например, методом “пузырька” или с помощью “быстрой сортировки” (эти алгоритмы изучались в 10-м классе). Закончить программу вы можете самостоятельно.

### Как это работает?

Чтобы грамотно применять динамические массивы, необходимо разобраться в том, как они работают. Для этого выведем на экран размер массива из 100 элементов (целых чисел):

```
SetLength(A, 100);
write(sizeof(A));
write(100*sizeof(integer));
```

Вы с удивлением обнаружите, что если программа выводит числа 4 и 200, то функция `sizeof` считает, что размер массива равен 4 байтам, хотя на самом деле 100 целых переменных должны занимать 200 байт. Более того, величина `sizeof(A)` не зависит от фактического размера массива. Поэтому можно сделать вывод, что размер элементов тут вообще не учитывается.

На самом деле в переменной `A` хранится адрес массива в памяти, который действительно занимает 4 байта. Таким образом, фактически `A` — это указатель. Если массив имеет нулевой размер, этот указатель равен нулю (нулевой адрес в Паскале обозначается как `nil`).

Что из этого следует? Представьте, например, что мы построили структуру, которая состоит из массива (поле `data`) и количества используемых элементов в нем (поле `size`):

```
type TArray = record
  data: array of integer;
  size: integer;
end;
```



Допустим, создана переменная типа **TArray**:

```
var A: TArray;
```

для которой выделен в памяти внутренний массив **data** и заполнен натуральными числами:

```
SetLength(A.data, 10);  
for i:=0 to 9 do A.data[i]:=i;  
A.size:=10;
```

Что будет, если мы попытаемся сохранить такую структуру в файле? Несложно понять, что в файл запишутся только элементы структуры, то есть *адрес массива data* и количество элементов **size**. Сами элементы не входят в структуру, поэтому сохранены не будут. Если после этого мы прочитаем из файла такую структуру, адрес **data** будет недействителен, и использовать его нельзя. Поэтому при сохранении в файле структур с динамическими полями нужно принимать специальные меры по сохранению содержимого массивов.

Динамическая матрица — это указатель на массив указателей:

```
var A: array of array of integer;
```

Если применить к ней процедуру **SetLength** с одним параметром

```
SetLength(A, 10);
```

то в памяти будет выделен массив указателей на строки матрицы, причем память под сами строки не выделяется. То есть **A[1]** (строка матрицы с индексом 1) — это указатель, на который можно “навесить” динамический массив любого размера, например, так:

```
for i:=0 to 9 do  
  SetLength(A[i], i+1);
```

Таким образом, мы получили матрицу, где все строки имеют разную длину:

```
writeln(High(A[0])); { = 1 }  
writeln(High(A[9])); { = 10 }
```

## Контрольные вопросы

1. Приведите примеры задач, в которых использование динамических массивов дает преимущества (какие именно?).
2. Что такое динамические структуры данных? Где выделяется память под эти данные?
3. Как объявить в программе динамический массив и задать его размер?
4. Как расширить массив в ходе работы программы? Не потеряются ли при этом уже записанные в нем данные?
5. Как определить границы изменения индексов динамического массива? Нужно ли хранить его размер в отдельной переменной?
6. Как удалить массив из памяти?
7. Как разместить в памяти динамическую матрицу?
8. Как передать динамический массив в подпрограмму?
9. Какие проблемы могут возникнуть при сохранении динамических массивов и матриц в файлах? Как вы предлагаете их решать?

## Задачи

1. Напишите полные программы для решения задач, рассмотренных в тексте параграфа.

2. Введите с клавиатуры число  $N$  и вычислите все простые числа в интервале от 2 до  $N$ .

3. Введите с клавиатуры число  $N$  и вычислите первые  $N$  простых чисел.

4. Введите с клавиатуры число  $N$  и вычислите первые  $N$  чисел Фибоначчи ( $F_n = F_{n-1} + F_{n-2}$ ,  $F_1 = F_2 = 1$ ).

5. Напишите функцию, которая находит максимальный элемент переданного ей динамического массива.

6. Напишите подпрограмму, которая находит максимальный и минимальный элементы переданного ей динамического массива (используйте изменяемые параметры).

7. Напишите рекурсивную функцию, которая считает сумму элементов переданного ей динамического массива.

8. Напишите функцию, которая сортирует значения переданного ей динамического массива, используя алгоритм “быстрой сортировки” (см. учебник 10-го класса).

## Списки

### Что такое список?

**Задача 4.** В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Построить алфавитно-частотный словарь: все различные слова должны быть записаны в другой файл в алфавитном порядке, справа от каждого слова указано, сколько раз оно встречается в исходном файле.

Для решения задачи нам нужно составить список, в котором хранить пары “слово — количество таких слов”. Список составляется по мере чтения файла, то есть это динамическая структура.

**Список** — это упорядоченный набор элементов одного типа, для которых введены операции вставки (включения) и удаления (исключения).

Обычно используют *линейные* списки, в которых для каждого элемента (кроме первого) можно указать предыдущий, а для каждого элемента, кроме последнего, — следующий.

Вернемся к нашей задаче построения алфавитно-частотного словаря. Алгоритм, записанный в виде псевдокода, может выглядеть так:

```
нц пока есть слова в файле  
  прочитать очередное слово  
  если оно есть в списке то  
    увеличить на 1 счетчик для этого слова  
  иначе  
    добавить слово в список  
    записать 1 в счетчик слова  
все  
кц
```

Теперь нужно выразить все шаги этого алгоритма через операторы языка программирования.

## Использование динамического массива

В нашем случае каждый элемент списка должен содержать пару значений: слово (символьную стро-

ку) и счетчик этих слов (целое число). Поэтому элементы списка — это структуры, тип которых можно описать так:

```
type TPair = record
    word: string;      { слово }
    count: integer;   { счетчик }
end;
```

Для организации списка будем использовать динамические массивы *FreePascal*.

Здесь ситуация похожа на задачу 2: размер массива становится известен только в конце работы программы. Поэтому требуется динамический массив, состоящий из описанных выше структур. С ним нужно выполнять следующие операции:

- искать заданное слово в списке;
- увеличивать счетчик заданного слова на 1;
- вставлять слово в определенное место списка (так, чтобы сохранить алфавитный порядок).

Как и в предыдущем параграфе, будем расширять размер массива сразу на 10 элементов, чтобы не выделять память слишком часто.

Объявим структуру-список:

```
type TWordList = record
    { динамический массив }
    data: array of TPair;
    { количество элементов }
    size: integer;
end;
```

Напомним, что количество фактически используемых элементов массива **size** может быть меньше, чем количество элементов, размещенных в памяти.

Введем переменную **L** типа **TWordList**:

```
var L: TWordList;
```

В начале основной программы очистим список и установим для него нулевую длину:

```
SetLength(L.data, 0);
L.size := 0;
```

Основной цикл (чтение данных из файла и построение списка) можно записать так (для последующего объяснения строки в теле цикла пронумерованы):

```
while not eof(F) do begin
    readln(F, s);           {1}
    p := Find(L, s);        {2}
    if p >= 0 then          {3}
        L.data[p].count := L.data[p].count + 1 {4}
    else begin              {5}
        p := FindPlace(L, s); {6}
        InsertWord(L, p, s); {7}
    end;                     {8}
end;
```

Здесь используются две вспомогательные переменные: символьная строка **s** (типа **string**) и целая переменная **p**. В строке 1 очередное слово читается из файла в строку **s**. Затем с помощью функции **Find** определяем, есть ли оно в списке (строка 2). Если есть (функция **Find** вернула существующий индекс), увеличиваем его счетчик (строки 3–4).

Если в списке слова еще нет (функция **Find** вернула **-1**), нужно найти место, куда его вставить, так чтобы не нарушился алфавитный порядок. Это делает функция **FindPlace**, которая должна возвращать номер элемента массива, перед которым нуж-

но вставить прочитанное слово. Вставку выполняет процедура **InsertWord**.

Когда список готов, остается вывести его в выходной файл:

```
Assign(F, 'output.dat');
Rewrite(F);
for p:=0 to L.size-1 do
    writeln(F, L.data[p].word, ': ',
            L.data[p].count);
```

```
Close(F);
```

Для каждого элемента списка в файл выводится хранящееся в нем слово и через двоеточие — сколько раз оно встретилось в тексте.

Таким образом, нам остается написать функции **Find** и **FindPlace**, а также процедуру **InsertWord**.

Функция **Find** принимает список и слово, которое нужно искать. В цикле проходим все элементы (не забывая, что их нумерация в динамическом массиве начинается с нуля). Как только очередное слово списка совпало с образцом, возвращаем в качестве результата функции номер этого элемента. Если просмотрены все элементы и совпадения не было, функция вернет **-1**.

```
function
    Find(L: TWordList; word: string): integer;
var i: integer;
begin
    Find := -1;
    for i:=0 to L.size-1 do
        if L.data[i].word = word then begin
            Find := i;
            break;
        end;
    end;
```

Здесь встретилось обозначение с двумя точками: **L.data[i].word**. Вспомним, что **L** — это структура-список, у него есть поле-массив **data**. В этом массиве идет обращение к элементу с номером **i**. Этот элемент — структура типа **TPair**, в составе которой есть поле **word**. Таким образом, **L.data[i].word** означает “поле **word** в составе *i*-го элемента массива **data**, который входит в состав структуры **L**”.

Функция **FindPlace** также принимает в параметрах список и слово. Она находит место вставки нового слова в список, при котором сохраняется алфавитный порядок расположения слов. Результат функции — номер слова, перед которым нужно вставить заданное. Для этого нужно найти в списке слово, которое “больше” заданного. Если такое слово не найдено, новое слово вставляется в конец списка:

```
function FindPlace(L: TWordList; word:
string): integer;
var i, p: integer;
begin
    p := -1;
    for i:=0 to L.size-1 do
        if L.data[i].word > word then begin
            p := i;
            break;
        end;
    if p < 0 then p := L.size;
    FindPlace := p;
end;
```

Заметим, что обе функции можно было описать с использованием двоичного поиска.

Процедура `InsertWord` вставляет слово `word` в позицию `k` в список `L`:

```
procedure InsertWord(var L: TWordList;
                    k: integer; word: string);
var i: integer;
begin
  IncSize(L); {1}
  for i:=L.size-1 downto k+1 do {2}
    L.data[i]:=L.data[i-1]; {3}
  L.data[k].word:=word; {4}
  L.data[k].count:=1; {5}
end;
```

Поскольку в список добавляется новый элемент, его размер увеличивается. Для этого введена процедура `IncSize`, которая вызывается в строке 1 (мы напишем ее позже). Далее в цикле сдвигаем все последние элементы, включая элемент с номером `k`, на одну ячейку к концу массива (строки 2–3). Таким образом, элемент с номером `k` освобождается. В строке 4 в него записывается новое слово, а в строке 5 — счетчик этого слова устанавливается равным 1.

Процедура `IncSize` расширяет список на 1 элемент. Когда нужный размер становится больше, чем размер динамического массива, массив увеличивается сразу на 10 элементов:

```
procedure IncSize (var L: TWordList);
begin
  L.size:=L.size+1;
  if L.size > Length(L.data) then
    SetLength(L.data, Length(L.data)+10);
end;
```

Процедура `IncSize` в программе должна располагаться выше вызывающей ее процедуры `InsertWord`.

Приведем окончательную структуру программы:

```
program AlphaList;
  { объявления типов TPair и TWordList }
var F: text;
    w: string;
    L: TWordList;
    p: integer;
  { процедуры и функции }
begin
  SetLength(L.data, 0);
  L.size := 0;
  Assign(F, 'input.dat');
  Reset(F);
  { основной цикл: составление списка слов }
  Close(F);
  { вывод результата в файл }
end.
```

Блоки, выделенные синим цветом, уже были написаны ранее в этом параграфе.

Заметим, что если известно максимальное количество разных слов в файле (скажем, не более 1000), то же самое можно сделать и на основе обычного (статического) массива, в котором память выделена заранее на максимальное число элементов.

## Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (*модуль*). Все подпрограммы, входящие в модуль, должны быть связаны друг с другом, но слабо связаны с другими процедурами и функциями. В нашей программе в отдельный модуль можно вынести все подпрограммы, работающие со списком слов.

Модуль в языке Паскаль, в отличие от основной программы, начинается со слова `unit`, после которого ставится название модуля.

```
unit WordList;
interface
  ...
implementation
  ...
end.
```

В модуле два основных раздела: `interface` (интерфейс, общедоступная часть) и `implementation` (реализация, недоступная другим модулям). В разделе `interface` обычно размещают объявления типов данных, функций и процедур, а в разделе `implementation` — программный код. В нашей программе модуль может выглядеть так:

```
unit WordList;
interface
  type TPair = record
    word: string;
    count: integer;
  end;
  TWordList = record
    data: array of TPair;
    size: integer;
  end;
  function Find(L: TWordList;
               word: string): integer;
  function FindPlace(L: TWordList;
                   word: string): integer;
  procedure InsertWord(var L: TWordList;
                      k: integer; word: string);
implementation
  { процедуры и функции }
end.
```

В секции `interface` мы расположили объявление типов данных, которые будут нужны основной программе, и заголовки подпрограмм этого модуля, которые могут вызываться извне. Все, что находится в секции `implementation`, скрыто от “внешнего мира”. В частности, там могут быть внутренние подпрограммы, которые “видны” только внутри модуля (в нашем случае это процедура `IncSize`).

Модуль подключается к основной программе (или к другому модулю) с помощью ключевого слова `uses`. Если программа использует несколько модулей, все они перечисляются через запятую после слова `uses`.

Наша основная программа, использующая модуль `WordList`, выглядит так:

```
program AlphaList;
uses WordList; { подключение модуля }
var F: text;
    s: string;
```

```
L: TWordList;
p: integer;
begin
  { тело основной программы }
end.
```

Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других. Кроме того, такой подход ускоряет трансляцию больших программ, так как каждый модуль транслируется отдельно, причем только в том случае, если он был изменен.

**Связные списки**

Линейный список иногда представляется в программе в виде *связного списка*, в котором каждый элемент может быть размещен в памяти в произвольном месте, но должен содержать ссылку (указатель) на следующий элемент. У последнего элемента эта ссылка нулевая (в Паскале — `nil`), она показывает, что следующего элемента нет. Кроме того, нужно хранить где-то (в указателе `Head`) адрес первого элемента (“голова”) списка, иначе список будет недоступен:

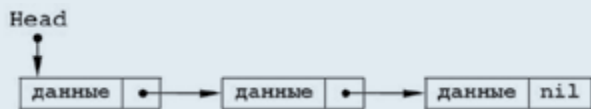


Рис. 4

Связный список часто рассматривают как рекурсивную структуру данных. Рекурсивное определение может быть дано так:

- 1) пустой список (состоящий из 0 элементов) — это список;
  - 2) список — это узел и связанный с ним список.
- Здесь вторая часть определяет рекурсию, а первая — условие окончания рекурсии.

Если замкнуть связный список в кольцо, так чтобы последний элемент содержал ссылку на первый, получается *циклический список*:



Рис. 5

Поскольку элементы связного списка содержат ссылки только на следующий элемент, к предыдущему перейти нельзя. Поэтому перебор возможен только в одном направлении. Этот недостаток устранен в двусвязном списке, где каждый элемент хранит адрес как следующего, так и предыдущего:



Рис. 6

Для такого списка обычно хранятся два адреса: “голова” списка (указатель `Head`) и его “хвост”

(указатель `Tail`). Можно организовать и циклический двусвязный список. Использование двух указателей для каждого элемента приводит к дополнительному расходу памяти и усложнению всех операций со списком, потому что при добавлении и удалении элемента нужно правильно расставить оба указателя.

Применение связанных списков приводит к более сложным алгоритмам, чем работа с динамическими массивами, поэтому приводить соответствующие программы мы не будем.

**Контрольные вопросы**

1. Что такое список? Какие операции он допускает?
2. Верно ли, что элементы в списке упорядочены?
3. Какой метод поиска в списке можно использовать? Обсудите разные варианты.
4. Как добавить элемент в линейный список, сохранив заданный порядок сортировки?
5. Как можно представить список в программе? В каких случаях для этого можно использовать обычный массив?
6. Объясните запись `L.data[i].word`.
7. Что такое модуль? Зачем используют модули?
8. Как оформляется текст модуля? Как по нему отличить модуль от основной программы?
9. Что размещается в секциях `interface` и `implementation`?
10. Можно ли все переменные и подпрограммы поместить в секцию `interface`? Чем это плохо?
11. Как подключается модуль к основной программе или другому модулю?
12. Что такое связный список?
13. Дайте рекурсивное определение связного списка и объясните его.
14. Что такое циклический список? Попробуйте придумать задачу, где после завершения просмотра списка нужно начать просмотр заново.
15. Сравните односвязный и двусвязный списки. В чем достоинства и недостатки одного и второго типов?

**Задачи**

1. Постройте программу, которая составляет алфавитно-частотный словарь заданного файла. Используйте модуль, содержащий все операции со списком.
2. В предыдущей программе объедините функции `Find` и `FindPlace`, заменив их на одну функцию. Если слово найдено в списке, функция работает так же, как `Find`: возвращает номер слова в списке. Если слово не найдено, функция должна вернуть отрицательное число: номер элемента массива, перед которым нужно вставить слово, со знаком минус.
3. \*В предыдущей задаче вывести все найденные слова в файл в порядке убывания частоты, то есть в начале списка должны стоять слова, которые встречаются в файле чаще всех.



## Стек, очередь, дек

### Что такое стек?

Представьте себе стопку книг (подносов, кирпичей и т.п.). С точки зрения информатики ее можно воспринимать как список элементов, расположенных в определенном порядке. Этот список имеет одну особенность — удалять и добавлять элементы можно только с одной (“верхней”) стороны. Действительно, для того чтобы вытащить какую-то книгу из стопки, нужно сначала снять все те книги, которые находятся на ней. Положить книгу сразу в середину тоже нельзя.

**Стек** (англ. *stack* — “стопка”) — это линейный список, в котором элементы добавляются и удаляются только с одного конца (“последним пришел — первым ушел”<sup>3</sup>).

На рис. 7 показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка:

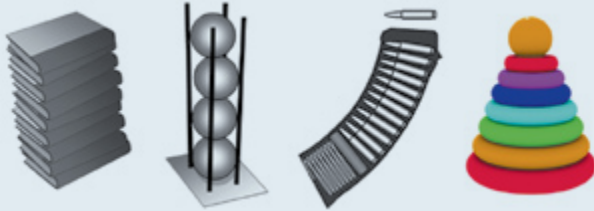


Рис. 7

Как вы знаете из учебника 10-го класса, стек используется при выполнении программ: в нем хранятся (временно) адреса возврата; параметры, передаваемые функциям и процедурам, а также локальные переменные.

**Задача 5.** В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке.

В этой задаче очень удобно использовать стек. Для стека определены две операции:

- добавить элемент на вершину стека (англ. *push* — “втолкнуть”);
- получить элемент с вершины стека и удалить его из стека (англ. *pop* — “вытолкнуть”).

Запишем алгоритм решения на псевдокоде. Сначала читаем данные и добавляем их в стек:

```
нц пока файл не пуст
  прочитать x
  добавить x в стек
кц
```

Теперь верхний элемент стека — это последнее число, прочитанное из файла. Поэтому остается “вытолкнуть” все записанные в стек числа, они будут выходить в обратном порядке:

```
нц пока стек не пуст
  вытолкнуть число из стека в x
  записать x в файл
кц
```

<sup>3</sup> Англ. LIFO = *Last In – First Out*.

## Использование динамического массива

Поскольку стек — это линейная структура данных с переменным количеством элементов, для создания стека в программе мы можем использовать динамический массив. Конечно, можно организовать стек из обычного (статического) массива, но его будет невозможно расширить сверх размера, выделенного при трансляции.

Для рассмотренной выше задачи 5 структура стека содержит динамический целочисленный массив и количество используемых в нем элементов:

```
type TStack = record
  data: array of integer;
  size: integer;
end;
```

Будем считать, что стек “растет” от начала к концу массива, то есть вершина стека — это последний элемент.

Для работы со стеком нужны две подпрограммы:

- процедура **Push**, которая добавляет новый элемент на вершину стека;
- функция **Pop**, которая возвращает верхний элемент стека и убирает его из стека.

Приведем эти подпрограммы:

```
procedure Push(var S: TStack;
  x: integer);
begin
  if S.size > High(S.data) then
    SetLength(S.data, Length(S.data) + 10);
  S.data[S.size] := x;
  S.size := S.size + 1;
end;

function Pop(var S: TStack): integer;
begin
  S.size := S.size - 1;
  Pop := S.data[S.size];
end;
```

Обратите внимание, что здесь структура типа **TStack** изменяется внутри подпрограмм, поэтому этот параметр должен быть изменяемым (описан с помощью **var**).

Заметим, что если нам понадобится стек, который хранит данные другого типа (например, символы, символьные строки или структуры), в объявлении типа и в приведенных подпрограммах нужно просто заменить **integer** на нужный тип.

Теперь несложно написать цикл ввода данных в стек из файла:

```
SetLength(S.data, 0);
S.size := 0;
while not eof(F) do begin
  read(F, x);
  Push(S, x);
end;
```

Здесь **s** — переменная типа **TStack**; **F** — файловая переменная, связанная с файлом, открытым на чтение; **x** — целая переменная. Вывод результата в файл выполняется аналогично:

```
for i:=0 to S.size-1 do begin
  x := Pop(S);
  writeln(F, x);
end;
```

Здесь *i* — целая переменная, а *F* — файловая переменная, связанная с файлом, открытым на запись.

### Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, записанные в такой форме:  $(5+15) / (4+7-1)$ ? Такая запись называется *инфиксной* — в ней знак операции расположен между операндами (данными, участвующими в операции). Инфиксная форма неудобна для автоматических вычислений из-за того, что выражение содержит скобки и его нельзя вычислить за один проход слева направо.

В 1920 году польский математик Ян Лукашевич предложил *префиксную* форму, которую стали называть польской нотацией. В ней знак операции расположен перед операндами. Например, выражение  $(5+15) / (4+7-1)$  может быть записано в виде  $/ + 5 15 - + 4 7 1$ . Скобок здесь не требуется, так как порядок операций строго определен: сначала выполняются два сложения ( $+ 5 15$  и  $+ 4 7$ ), затем вычитание и, наконец, деление. Первой стоит последняя операция. Таким образом, выражение в префиксной форме нужно вычислять с конца.

В середине 1950-х годов была предложена *обратная польская нотация*, или *постфиксная* форма записи, в которой знак операции стоит после операндов:

$5 15 + 1 4 7 + - /$

В этом случае также не нужны скобки, и выражение может быть вычислено за один просмотр с помощью стека следующим образом:

- если очередной элемент — число (или переменная), он записывается в стек;
- если очередной элемент — операция, то она выполняется с верхними элементами стека, и после этого в стек вталкивается результат выполнения этой операции.

Покажем, как работает этот алгоритм (стек “растет” снизу вверх):

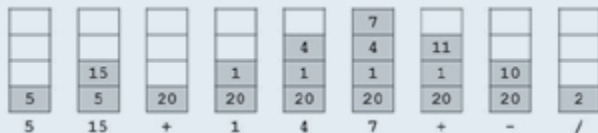


Рис. 8

В результате в стеке остается значение заданного выражения.

### Скобочные выражения

**Задача 6.** Вводится символьная строка, в которой записано некоторое (арифметическое) выражение, использующее скобки трех типов:  $()$ ,  $[\ ]$  и  $\{ \}$ . Проверить, правильно ли расставлены скобки.

Например, выражение  $() \{ ( ) [\ ] \}$  — правильное, потому что каждой открывающей скобке соответствует закрывающая, и вложенность скобок не нарушается. Выражения  $[ ( )$ ,  $[\ [ ( )$ ,  $[ ( ) ]$ ,  $) ($ ,  $( [ ]$  неправильные. В первых трех есть непарные скобки, а в последних двух не соблюдается вложенность скобок.

Начнем с аналогичной задачи, в которой используется только один вид скобок. Ее можно решить с помощью счетчика скобок. Сначала счетчик равен нулю. Строка просматривается слева направо; если очередной символ — открывающая скобка, то счетчик увеличивается на 1, если закрывающая — уменьшается на 1. В конце просмотра счетчик должен быть равен нулю (все скобки парные), кроме того, во время просмотра он не должен становиться отрицательным (должна соблюдаться вложенность скобок).

В исходной задаче (с тремя типами скобок) хочется завести три счетчика и работать с каждым отдельно. Однако это решение неверное. Например, для выражения  $\{ ( [ ] \}$  условия “правильности” выполняются отдельно для каждого вида скобок, но не для выражения в целом.

Задачи, в которых важна вложенность объектов, удобно решать с помощью стека. У нас объекты — это открывающие и закрывающие скобки, на остальные символы можно не обращать внимания. Строка просматривается слева направо. Если очередной символ — открывающая скобка, нужно втолкнуть ее на вершину стека. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там соответствующая открывающая скобка, то ее нужно просто снять со стека. Если стек пуст или на вершине лежит открывающая скобка другого типа, выражение неверное и нужно закончить просмотр. В конце обработки правильной строки стек должен быть пуст. Кроме того, во время просмотра не должно быть ошибок. Такой алгоритм иллюстрируется на рис. 9 (для правильного выражения):



Рис. 9

Введем следующие переменные:

```
type TStack = record
    data: array of char;
    size: integer;
end;
var S: TStack;
    p, i: integer;
    str, L, R: string;
    err: boolean;
    c: char;
```

Отличие стека *S* от предыдущего примера только в том, что он содержит не целые числа, а символы (типа *char*). Поэтому приводить подпрограммы *Push* и *Pop* мы не будем, вы можете их переделать самостоятельно. Для удобства добавим логическую функцию, которая возвращает значение *True* (истина), если стек пуст:

```
function isEmpty(S: TStack): boolean;
begin
    isEmpty := (S.size = 0);
end;
```

Переменная *str* — исходная строка, в переменную *L* запишем все виды открывающих скобок, а в

переменную **R** — соответствующие им закрывающие скобки (в том же порядке):

```
L := ' ( { ' ;
R := ' ) ] } ' ;
```

Логическая переменная **err** будет сигнализировать об ошибке. Сначала ей присваивается значение **False** (ложь).

В основном цикле меняется целая переменная **i**, которая обозначает номер текущего символа, переменная **p** используется как вспомогательная:

```
for i:=1 to Length(str) do begin
  p:= Pos(str[i], L); {1}
  if p > 0 then Push(S, str[i]); {2}
  p := Pos(str[i], R); {3}
  if p > 0 then begin {4}
    if isEmpty(S) then err:=True {5}
    else begin
      c:= Pop(S); {6}
      if p <> Pos(c,L) then err:= True {7}
    end;
    if err then break {8}
  end
end;
```

Сначала мы ищем символ **str[i]** в строке **L**, то есть среди открывающих скобок (строка 1). Если это действительно открывающая скобка, вталкиваем ее в стек (2). Далее ищем символ среди закрывающих скобок (3). Если нашли, то в первую очередь проверяем, не пуст ли стек. Если стек пуст, выражение неверное и переменная **err** принимает истинное значение. Если в стеке что-то есть, снимаем символ с вершины стека в символьную переменную **c** (6). В строке (7) сравнивается тип (номер) закрывающей скобки **p** и номер открывающей скобки, найденной на вершине стека. Если они не совпадают, выражение неправильное, и в переменную **err** записывается значение **True**. Если при обработке текущего символа обнаружено, что выражение неверное (переменная **err** установлена в **True**), нужно закончить цикл досрочно с помощью оператора **break** (8).

В конце программы остается вывести результат на экран:

```
if not err then
  writeln('Выражение правильное. ');
else writeln('Выражение неправильное.');
```

### Очереди, деки

Все мы знакомы с принципом очереди: первый пришел — первый обслужен (англ. *FIFO* = *First In - First Out*). Соответствующая структура данных в информатике тоже называется очередью.

**Очередь** — это линейный список, для которого определяются две операции:

- добавление нового элемента в конец очереди;
- удаление первого элемента из очереди.

Очередь — это не просто теоретическая модель. Операционные системы используют очереди для

организации сообщения между программами: каждая программа имеет свою очередь сообщений. Контроллеры жестких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создается очередь из пакетов данных, ожидающих отправки.

**Задача 7.** Рисунок задан в виде матрицы **A**, в которой элемент **A[y, x]** определяет цвет пикселя на пересечении строки **y** и столбца **x**. Перекрасить в цвет 2 одноцветную область, начиная с пикселя  $(x_0, y_0)$ . На рисунке показан результат такой заливки для матрицы из пяти строк и пяти столбцов с начальной точкой (2,1).

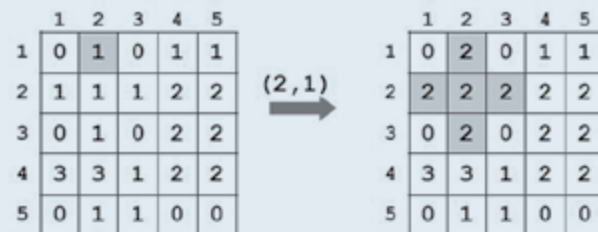


Рис. 10

Эта задача актуальна для графических программ. Один из возможных вариантов решения использует очередь, элементы которой — координаты пикселей (точек):

```
добавить в очередь точку (x0, y0)
запомнить цвет начальной точки
нц пока очередь не пуста
  взять из очереди точку (x, y)
  если A[y, x] = цвет начальной точки то
    A[y, x] := 2;
    добавить в очередь точку (x-1, y)
    добавить в очередь точку (x+1, y)
    добавить в очередь точку (x, y-1)
    добавить в очередь точку (x, y+1)
все
кц
```

Конечно, в очередь добавляются только те точки, которые находятся в пределах рисунка (матрицы **A**).

Две координаты точки связаны между собой, поэтому в программе лучше объединить их в структуру **TPoint** (от англ. *point* — “точка”), а очередь составить из таких структур:

```
Type TPoint = record
  x, y: integer;
end;
TQueue = record
  data: array of TPoint;
  size: integer;
end;
```

Для удобства построим функцию **Point**, которая формирует структуру типа **TPoint** по заданным координатам:

```
function Point(x, y: integer): TPoint;
begin
  Point.x := x;
  Point.y := y;
end;
```

Для работы с очередью, основанной на динамическом массиве, введем две подпрограммы:

- процедура **Put** добавляет новый элемент в конец очереди; если нужно, массив расширяется блоками по 10 элементов;

- функция **Get** возвращает первый элемент очереди и удаляет его из очереди; все следующие элементы сдвигаются к началу массива.

```
procedure Put(var S: TQueue; x: TPoint);
begin
  if S.size > High(S.data) then
    SetLength(S.data, Length(S.data) + 10);
  S.data[S.size] := x;
  S.size := S.size + 1;
end;

function Get(var S: TQueue): TPoint;
var i: integer;
begin
  Get := S.data[0];
  S.size := S.size - 1;
  for i := 0 to S.size - 1 do
    S.data[i] := S.data[i + 1];
  end;
end;
```

Остается написать основную программу. Объявляем константы и переменные:

```
const XMAX = 5; YMAX = 5;
var S: TQueue;
    x0, y0, color: integer;
    A: array[1..YMAX, 1..XMAX] of integer;
    pt: TPoint;
```

Сначала задаем исходную точку и запоминаем ее “старый” цвет:

```
x0 := 2; y0 := 1;
color := A[y0, x0];
Put(S, Point(x0, y0));
```

Основной цикл практически повторяет алгоритм на псевдокоде:

```
while not isEmpty(S) do begin
  pt := Get(S);
  if A[pt.y, pt.x] = color then begin
    A[pt.y, pt.x] := 2;
    if pt.x > 1 then Put(S,
      Point(pt.x - 1, pt.y));
    if pt.x < XMAX then Put(S,
      Point(pt.x + 1, pt.y));
    if pt.y > 1 then
      Put(S, Point(pt.x, pt.y - 1));
    if pt.y < YMAX then Put(S,
      Point(pt.x, pt.y + 1));
  end;
end;
```

Здесь функция **isEmpty** — такая же, как и для стека (возвращает **True**, если очередь пуста (поле **size** равно нулю)).

Существует еще одна линейная динамическая структура данных, которая называется *дек*.

**Дек** — это линейный список, в котором можно добавлять и удалять элементы как с одного, так и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью стека можно, например, моделировать колоду игральнх карт.



Рис. 11

## Контрольные вопросы

1. Что такое стек? Какие операции со стеком разрешены?
2. Как используется системный стек при выполнении программ?
3. Какие ошибки могут возникнуть при использовании стека?
4. В каких случаях можно использовать обычный массив для моделирования стека?
5. Как использовать стек на основе динамического массива?
6. Почему при передаче стека в подпрограмму соответствующий параметр должен быть изменяемым?
7. Что такое очередь? Какие операции она допускает?
8. Приведите примеры задач, в которых можно использовать очередь.

## Задачи

1. Напишите программу, которая “переворачивает” массив, записанный в файл, с помощью стека. Размер массива неизвестен. Все операции со стеком вынесите в отдельный модуль.
2. Напишите программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме. Выражение вводится с клавиатуры в виде символьной строки.
3. Напишите программу, которая проверяет правильность скобочного выражения с четырьмя видами скобок:  $()$ ,  $[]$ ,  $\{\}$  и  $\langle \rangle$ . Все операции со стеком вынесите в отдельный модуль.
4. Напишите программу, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки. Все операции с очередью вынесите в отдельный модуль.

## Деревья

### Что такое дерево?

Как вы знаете из учебника 10-го класса, дерево — это структура, отражающая иерархию (отношения подчиненности, многоуровневые связи). Напомним некоторые основные понятия, связанные с деревьями.

Дерево состоит из узлов и связей между ними (они называются дугами). Самый первый узел, рас-



положенный на верхнем уровне (в него не входит ни одна стрелка-дуга), — это *корень дерева*. Конечные узлы, из которых не выходит ни одна дуга, называются *листьями*. Все остальные узлы, кроме корня и листьев, — это промежуточные узлы.

Высота дерева — это наибольшее расстояние (количество дуг) от корня до листа.

Из двух связанных узлов тот, который находится на более высоком уровне, называется “*родителем*”, а другой — “*сыном*”. Корень — это единственный узел, у которого нет “родителя”; у листьев нет “сыновей”.

Используются также понятия “*предок*” и “*потомок*”. “*Потомок*” какого-то узла — это узел, в который можно перейти по стрелкам от узла-предка. Соответственно, “*предок*” какого-то узла — это узел, из которого можно перейти по стрелкам в данный узел.

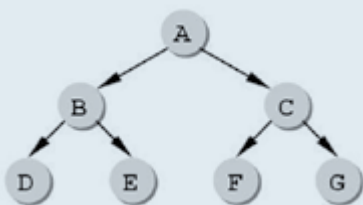


Рис. 12

В дереве на рисунке справа родитель узла E — это узел B, а предки узла E — это узлы A и B, для которых узел E — потомок. Потомками узла A (корня) являются все остальные узлы.

Высота дерева — это наибольшее расстояние (количество ребер) от корня до листа. Высота дерева, приведенного на рис. 12, равна 2.

Формально **дерево** можно определить следующим образом:

- 1) пустая структура — это дерево;
- 2) дерево — это корень и несколько связанных с ним отдельных деревьев.

Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Такой прием называется *рекурсией* (см. главу 8 учебника 10-го класса). Согласно этому определению, дерево — это рекурсивная структура данных. Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

Чаще всего в информатике применяются *двоичные* (или *бинарные*) деревья, то есть такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

**Двоичное дерево:**

- 1) пустая структура — это двоичное дерево;
- 2) дерево — это корень и два связанных с ним двоичных дерева (“левое” и “правое” поддерева).

В информатике деревья широко применяются в следующих задачах:

- поиск в большом массиве неменяющихся данных;

- сортировка данных;
- вычисление арифметических выражений;
- оптимальное кодирование данных (метод сжатия Хаффмана).

## Деревья поиска

Известно, что, для того чтобы найти заданный элемент в неотсортированном массиве из  $N$  элементов, может понадобиться  $N$  сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева, например:

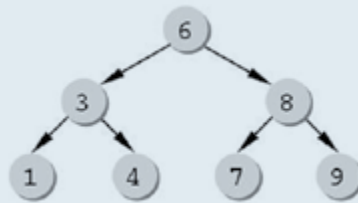


Рис. 13

Значения, связанные с каждым из узлов дерева, по которым выполняется поиск, называются *ключами* этих узлов. Кроме ключа, узел может содержать множество других данных. Перечислим важные свойства показанного дерева:

- слева от каждого узла находятся узлы с меньшим ключом;
- справа от каждого узла находятся узлы, ключ которых больше или равен ключу данного узла.

Дерево, обладающее такими свойствами, называется *двоичным деревом поиска*.

Например, нужно найти узел, ключ которого равен 4. Начинаем поиск по дереву с корня — 6 (больше заданного), поэтому дальше нужно искать только в левом поддереве. Если при линейном поиске в массиве за одно сравнение отсекается один элемент, здесь — сразу примерно половина оставшихся. Количество операций сравнения в этом случае пропорционально  $\log_2 N$ , то есть алгоритм имеет асимптотическую сложность  $O(\log N)$ . Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм поиска выгодно применять, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

## Обход дерева

Обойти дерево — это значит “посетить” все узлы по одному разу. Если перечислить узлы в порядке их посещения, мы представим данные в виде списка.

Существует несколько способов обхода дерева:

- КЛП = “корень — левый — правый” (обход в прямом порядке):

**посетить корень**  
**обойти левое поддерево**  
**обойти правое поддерево**

- ЛКП = “левый — корень — правый” (симметричный обход):

**обойти левое поддерево**  
**посетить корень**  
**обойти правое поддерево**

• ЛПК = “левый — правый — корень” (обход в обратном порядке):

- обойти левое поддерево
- обойти правое поддерево
- посетить корень

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень — пустое дерево.

Рассмотрим дерево, которое может быть составлено для вычисления арифметического выражения  $(1+4) * (9-5)$ :

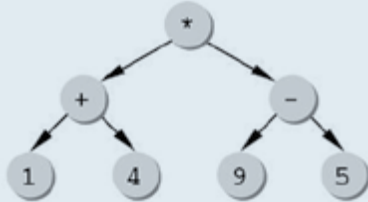


Рис. 14

Выражение вычисляется по такому дереву снизу вверх, то есть корень дерева — это последняя выполняемая операция.

Различные типы обходов дают последовательность узлов:

- КЛП: \* + 1 4 - 9 5
- ЛКП: 1 + 4 \* 9 - 5
- ЛПК: 1 4 + 9 5 - \*

В первом случае мы получили префиксную форму записи арифметического выражения, во втором — привычную нам инфиксную форму (только без скобок), а в третьем — постфиксную форму. Напомним, что в префиксной и постфиксной формах скобки не требуются.

Существует еще один способ обхода, который называют “обходом в ширину”. Сначала посещают корень дерева, затем — всех его “сыновей”, затем — “сыновей сыновей” (“внуков”) и т.д., постепенно спускаясь на один уровень вниз. Обход в ширину для приведенного выше дерева даст такую последовательность посещения узлов:

обход в ширину: \* + - 1 4 9 5

Для того чтобы выполнить такой обход, применяют очередь. В очередь записывают узлы, которые необходимо посетить. На псевдокоде обход в ширину можно записать так:

```

записать в очередь корень дерева
нц пока очередь не пуста
  V := выбрать первый узел из очереди
  посетить узел V
  если у узла V есть левый сын то
    добавить в очередь левого сына V
  все
  если у узла V есть правый сын то
    добавить в очередь правого сына V
  все
кц
    
```

### Вычисление арифметических выражений

Один из способов вычисления арифметических выражений основан на использовании дерева. Сначала выражение, записанное в линейном виде

(в одну строку), нужно “разобрать” и построить соответствующее ему дерево. Затем в результате прохода по этому дереву от листьев к корню вычисляется результат.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырех арифметических действий: +-\*/. Построим дерево для выражения

$$40 - 2 * 3 - 4 * 5$$

Так как корень дерева — это последняя операция, нужно сначала найти эту последнюю операцию, просматривая выражение слева направо. Здесь последнее действие — это второе вычитание, оно оказывается в корне дерева.

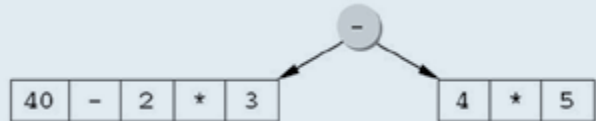


Рис. 15

Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке *приоритета* (старшинства): сначала операции с более высоким приоритетом (слева направо), потом — с более низким (также слева направо). Отсюда следует важный вывод.

В корень дерева нужно поместить последнюю из операций с наименьшим приоритетом.

Теперь нужно построить таким же способом левое и правое поддерева:

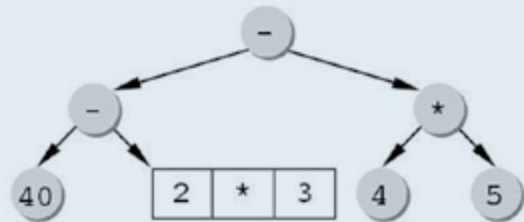


Рис. 16

Левое поддерево требует еще одного шага:

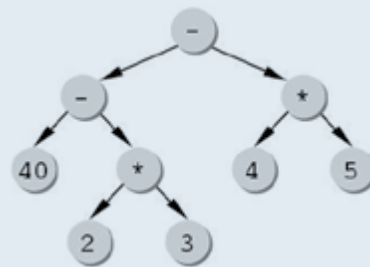


Рис. 17

Эта процедура рекурсивная, ее можно записать в виде псевдокода:

```

найти последнюю выполняемую операцию
если операций нет то
  создать узел-лист
  выход
все
поместить найденную операцию в корень
дерева
построить левое поддерево
построить правое поддерево
    
```

Рекурсия заканчивается, когда в оставшейся части строки нет ни одной операции, значит, там находится число (это лист дерева).

Теперь вычислить выражение по дереву. Если в корне находится знак операции, ее нужно применить к результатам вычисления поддеревьев:

```
n1 := значение левого поддерева
n2 := значение правого поддерева
результат := операция(n1, n2)
```

Снова получился рекурсивный алгоритм.

Возможен особый случай (на нем заканчивается рекурсия), когда корень дерева содержит число (то есть это лист). Это число и будет результатом вычисления выражения.

### Использование связанных структур

Поскольку двоичное дерево — это нелинейная структура данных, использовать динамический массив для размещения элементов не очень удобно (хотя возможно). Вместо этого будем использовать связанные узлы. Каждый такой узел — это структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). У листьев нет “сыновей”, в этом случае в указатели будем записывать значение `nil` (нулевой указатель). Дерево, состоящее из трех таких узлов, показано на рисунке:

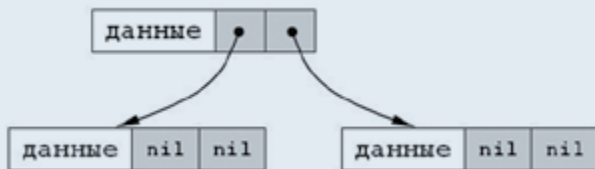


Рис. 18

В данном случае область данных узла будет содержать одно поле — символьную строку, в которую записывается знак операции или число в символьном виде.

Введем два новых типа: `TNode` — узел дерева, и `PNode` — указатель (ссылку) на такой узел:

```
type
  PNode = ^TNode;
  TNode = record
    data: string[20];
    left, right: PNode;
  end;
```

Самый важный момент — выделение памяти под новую структуру. Предположим, что `p` — это переменная-указатель типа `PNode`. Для того чтобы выделить память под новую структуру и записать адрес выделенного блока в `p`, используется процедура `New` (англ. *новый*):

```
New(p);
```

Как программа определяет, сколько памяти нужно выделить? Чтобы ответить на этот вопрос, вспомним, что указатель `p` указывает на структуру типа `TNode`, размер которой и определяет размер выделяемого блока памяти.

Для освобождения памяти служит процедура `Dispose` (англ. *ликвидировать*):

```
Dispose(p);
```

В основной программе объявим одну переменную `PNode` — это будет ссылка на корень дерева:

```
var T: PNode;
```

Основная часть программы будет состоять из двух операторов:

```
T := Tree(s);
writeln('Результат: ', Calc(T));
```

Здесь предполагается, что арифметическое выражение записано в символьной строке `s`, функция `Tree` строит в памяти дерево по этой строке, а функция `Calc` — вычисляет значение выражения по готовому дереву.

При построении дерева нужно выделить в памяти новый узел и искать последнюю выполняемую операцию — это будет делать функция `LastOp`. Она вернет 0, если ни одной операции не обнаружено, в этом случае создается лист — узел без потомков. Если операция найдена, ее обозначение записывается в поле `data`, а в указатели — адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
function Tree(s: string): PNode;
var k: integer;
begin
  New(Tree); { выделить память }
  k := LastOp(s);
  if k = 0 then begin { создать лист }
    Tree^.data := s;
    Tree^.left := nil;
    Tree^.right := nil;
  end
  else begin { создать узел-операцию }
    Tree^.data := s[k];
    Tree^.left := Tree(Copy(s, 1, k-1));
    Tree^.right := Tree(Copy(s, k+1,
      Length(s)-k));
  end;
end;
```

Функция `Calc` тоже будет рекурсивной:

```
function Calc(Tree: PNode): integer;
var n1, n2, res: integer;
begin
  if Tree^.left = nil then
    Val(Tree^.data, Calc, res)
  else begin
    n1 := Calc(Tree^.left);
    n2 := Calc(Tree^.right);
    case Tree^.data[1] of
      '+': Calc := n1 + n2;
      '-': Calc := n1 - n2;
      '*': Calc := n1 * n2;
      '/': Calc := n1 div n2;
    else Calc := MaxInt
    end
  end;
end;
```

Если ссылка, переданная функции, указывает на лист (нет левого поддерева), то значение выражения — это результат преобразования числа из символьной формы в числовую (с помощью процедуры `Val`). В противном случае вычисляются значения для левого и правого поддеревьев, и к ним применяется операция, указанная в корне дерева. В случае ошибки (неизвестной операции) функция воз-

вращает значение **MaxInt** — максимальное целое число.

Осталось написать функцию **LastOp**. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Для этого составим функцию, возвращающую приоритет операции (переданного ей символа):

```
function Priority(op: char): integer;
begin
  case op of
    '+', '-': Priority:=1;
    '*', '/': Priority:=2;
    else Priority:=100;
  end;
end;
```

Сложение и вычитание имеют приоритет 1, умножение и деление — приоритет 2, а все остальные символы (не операции) — приоритет 100 (условное значение).

Функция **LastOp** может выглядеть так:

```
function LastOp(s: string): integer;
var i, minPrt: integer;
begin
  minPrt:= 50;
  LastOp:= 0;
  for i:=1 to Length(s) do
    if Priority(s[i]) <= minPrt then begin
      minPrt:= Priority(s[i]);
      LastOp:= i;
    end;
  end;
```

Обратите внимание, что в условном операторе указано нестрогое неравенство, чтобы найти именно *последнюю* операцию с наименьшим приоритетом. Начальное значение переменной **minPrt** можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом не-операций (100). Тогда, если найдена любая операция, условный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной **LastOp** остается начальное значение 0.

**Хранение двоичного дерева в массиве**

Двоичные деревья удобно хранить в (динамическом) массиве, почти так же, как и списки. Вопрос о том, как сохранить структуру (взаимосвязь узлов), решается достаточно просто. Если нумерация элементов массива **A** начинается с 1, то “сыновья” элемента **A[i]** — это **A[2\*i]** и **A[2\*i+1]**. На рисунке показан порядок расположения элементов в массиве для дерева, соответствующего выражению

40 - 2 \* 3 - 4 \* 5

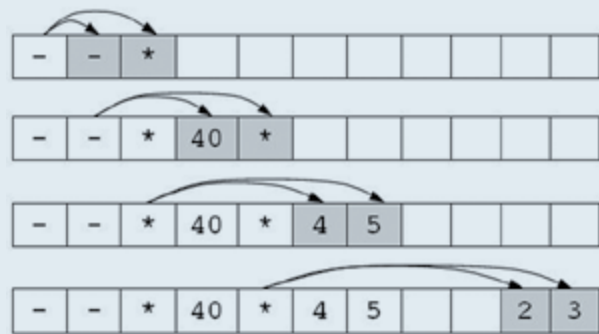
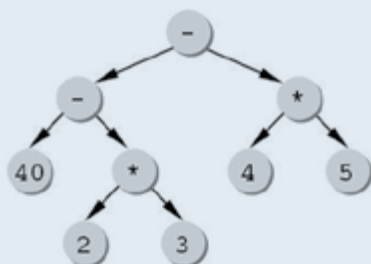


Рис. 19

Алгоритм вычисления выражения остается прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустые, это значит, что их “родитель” — лист дерева.

**Контрольные вопросы**

1. Дайте определение понятий “дерево”, “корень”, “лист”, “родитель”, “сын”, “потомок”, “предок”, “высота дерева”.
2. Где используются структуры типа “дерево” в информатике и в других областях?
3. Объясните рекурсивное определение дерева.
4. Можно ли считать, что связный список — это частный случай дерева?
5. Какими свойствами обладает дерево поиска?
6. Подумайте, как можно построить дерево поиска из массива данных.
7. Какие преимущества имеет поиск с помощью дерева?
8. Что такое “обход” дерева?
9. Какие способы обхода дерева вы знаете? Назовите их. Можно ли придумать другие способы обхода?
10. Как строится дерево для вычисления арифметического выражения?
11. Как представляется дерево в программе на Паскале?
12. Как указать, что узел дерева не имеет левого (правого) “сына”?
13. Как выделяется память под новый узел?
14. Как вы думаете, почему рекурсивные алгоритмы работы с деревьями получаются проще, чем не рекурсивные?
15. Как хранить двоичное дерево в массиве? Можно ли использовать такой прием для хранения деревьев, в которых узлы могут иметь больше двух “сыновей”?

**Задачи**

1. Напишите программу, которая вводит и вычисляет арифметическое выражение без скобок. Все операции с деревом вынесите в отдельный модуль.
2. Добавьте в предыдущую программу процедуры обхода построенного дерева так, чтобы получить префиксную и постфиксную записи введенного выражения.



3. \*Добавьте в предыдущую программу процедуру обхода дерева в ширину.

4. \*Усовершенствуйте программу (см. задачу 1), чтобы она могла вычислять выражения со скобками.

5. \*Включите в вашу программу обработку некоторых ошибок (например, два знака операций подряд). Поработайте в парах: обменяйтесь программами с соседом и попробуйте найти выражение, при котором его программа завершится аварийно (не выдаст сообщение об ошибке).

6. \*Напишите программу вычисления арифметического выражения, которая хранит дерево в виде массива. Все операции с деревом вынесите в отдельный модуль.

## Графы

### Что такое граф?

Как вы знаете из курса 10-го класса, граф — это набор узлов (вершин) и связей между ними (ребер). Информации об узлах и связях графа обычно хранят в виде таблицы специального вида — *матрицы смежности*:

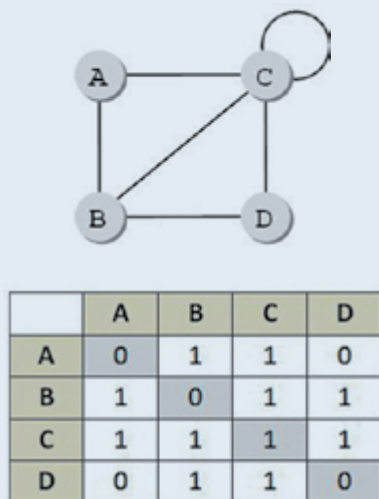


Рис. 20

Единица на пересечении строки A и столбца B означает, что между узлами A и B есть связь. Ноль указывает на то, что связи нет. Матрица смежности симметрична относительно главной диагонали (серые клетки в таблице). Единица на главной диагонали обозначает *петлю* — ребро, которое начинается и заканчивается в одной и той же вершине (в данном случае — в вершине C). Матрица смежности не дает никакой информации о том, как именно расположены узлы друг относительно друга. Для таблицы, приведенной выше, возможны, например, такие варианты, как на рис. 21.

В рассмотренном примере все узлы связаны, то есть между любой парой узлов существует *путь* — последовательность ребер, по которым можно перейти из одного узла в другой. Такой граф называется *связным*.

Вспоминая материал предыдущего пункта, можно сделать вывод, что дерево — это частный случай

связного графа, в котором нет замкнутых путей — циклов.

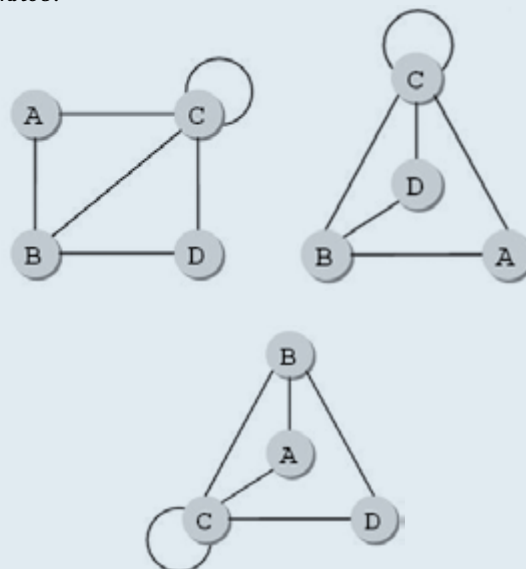


Рис. 21

Если для каждого ребра указано направление, граф называют *ориентированным* (или *орграфом*). Ребра орграфа называют *дугами*. Его матрица смежности не всегда симметричная. Единица, стоящая на пересечении строки A и столбца B, говорит о том, что существует дуга из вершины A в вершину B:

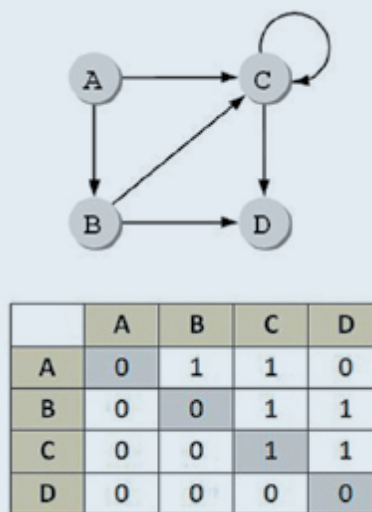
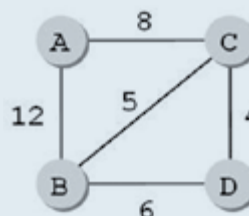


Рис. 22

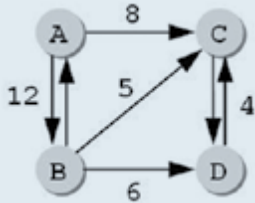
Часто с каждым ребром связывают некоторое число — *вес ребра*. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется *взвешенным*. Информация о таком графе хранится в виде *весовой матрицы*, содержащей веса ребер:



	A	B	C	D
A		12	8	0
B	12		5	6
C	8	5		4
D	0	6	4	

Рис. 23

У взвешенного орграфа весовая матрица не всегда симметрична относительно главной диагонали:



	A	B	C	D
A		12	8	
B	12		5	6
C				4
D			4	

Рис. 24

Если связи между двумя узлами нет, на бумаге можно оставить ячейку таблицы пустой, а при хранении в памяти компьютера записывать в нее условный код, например, 0, -1 или очень большое число ( $\infty$ ), в зависимости от задачи.

“Жадные” алгоритмы

**Задача 8.** Известна схема дорог между несколькими городами. Числа на схеме обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться):

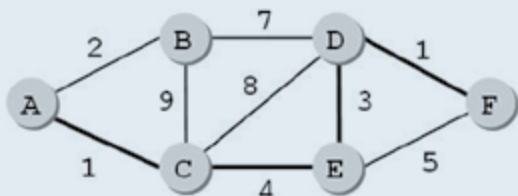


Рис. 25

Нужно найти кратчайший маршрут из города А в город F.

Первая мысль, которая приходит в голову, — на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы еще не были. Для данной схемы на первом этапе едем в город С (длина 1), далее — в E (длина 4), затем в D (длина 3) и, наконец, в F (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется “жадным”. Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы “жадный” алгоритм на самом деле дает оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой:

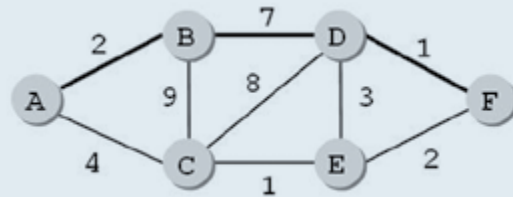


Рис. 26

“Жадный” алгоритм даст маршрут А-В-D-Ф длиной 10, хотя существует более короткий маршрут А-С-Е-Ф длиной 7.

“Жадный” алгоритм не всегда позволяет получить оптимальное решение.

Однако есть задачи, в которых “жадный” алгоритм всегда приводит к правильному решению. Одна из таких задач (ее называют задачей Прима — Крускала в честь Р.Прима и Д.Крускала, которые независимо предложили ее в середине XX века) формулируется так:

**Задача 9.** В стране Лимонии есть n городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?

В теории графов эта задача называется задачей построения минимального остовного дерева (то есть дерева, связывающего все вершины). Остовное дерево для связного графа с n вершинами имеет n-1 ребро.

“Жадный” алгоритм для этой задачи выглядит так:

- 1) начальное дерево — пустое;
- 2) на каждом шаге к дереву добавляется ребро минимального веса, которое еще не входит в дерево и не приводит к появлению цикла в дереве.

На рис. 27 показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии):

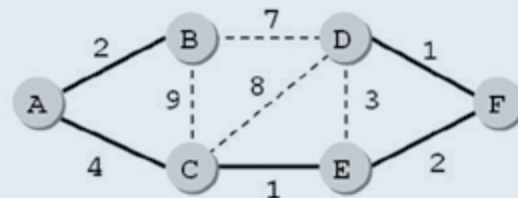


Рис. 27

Здесь возможна такая последовательность добавления ребер: CE, DF, AB, EF, AC. Обратите внимание, что после добавления ребра EF следующее “свободное” ребро минимального веса — это DE (длина 3), но оно образует цикл с ребрами DF и EF, и поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро еще не включено в дерево и не образует цикла в нем? Существует очень красивое решение этой проблемы, основанное на раскраске вершин.

Сначала все вершины раскрашиваются в разные цвета (то есть каждой из них присваиваются разные числовые коды):

```
for i:=1 to N do col[i]:=i;
```

Здесь  $N$  — количество вершин, а  $col$  — целочисленный массив с индексами от 1 до  $N$ .

Затем в цикле  $N-1$  раз (именно столько ребер нужно включить в дерево) выполняем следующие операции:

1) ищем ребро минимальной длины среди всех ребер, концы которых окрашены в разные цвета;

2) найденное ребро  $(iMin, jMin)$  добавляется в список выбранных, и все вершины, имеющие цвет  $col[jMin]$ , перекрашиваются в цвет  $col[iMin]$ .

Приведем полностью основной цикл программы:

```
for k:=1 to N-1 do begin
  { поиск ребра с минимальным весом }
  min:=MaxInt;
  for i:=1 to N do
    for j:=1 to N do
      if (col[i] <> col[j]) and
        (W[i,j] < min) then begin
        iMin:=i; jMin:=j; min:=W[i,j];
      end;
  { добавление ребра в список выбранных }
  ostov[k,1]:=iMin;
  ostov[k,2]:=jMin;
  { перекрашивание вершин }
  for i:=1 to N do
    if col[i] = col[jMin] then
      col[i]:=col[iMin];
end;
```

Здесь  $W$  — целочисленная матрица размера  $N$  на  $N$  (индексы строк и столбцов начинаются с 1);  $ostov$  — целочисленный массив из  $N-1$  строк и двух столбцов для хранения выбранных ребер (для каждого ребра хранятся две вершины, которые оно соединяет).

После окончания цикла остается вывести результат — ребра из массива  $ostov$ :

```
for i:=1 to N-1 do
  writeln('(', ostov[i,1], ', ',
    ostov[i,2], ')');
```

### Кратчайшие маршруты

В предыдущем пункте мы познакомились с задачей выбора кратчайшего маршрута и увидели, что в ней “жадный” алгоритм не всегда дает правильное решение. В 1960 году Э.Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех ребер (расстояния между вершинами) положительные.

Рассмотрим уже знакомую схему, в которой не сработал “жадный” алгоритм (см. рис. 28).

Алгоритм Дейкстры использует дополнительные массивы: в одном (назовем его  $R$ ) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором

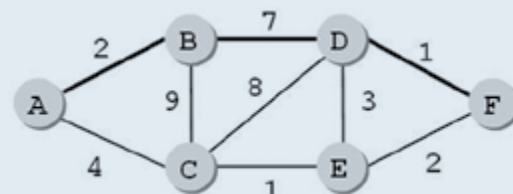


Рис. 28

(массив  $P$ ) — вершина, из которой нужно приехать в данную вершину.

Сначала записываем в массив  $R$  расстояния от исходной вершины  $A$  до всех вершин, а в соответствующие элементы массива  $P$  — вершину  $A$ :

	A	B	C	D	E	F
R	0	2	4	∞	∞	∞
P	x	A	A			

Рис. 29

Знак “∞” обозначает, что прямого пути из вершины  $A$  в данную вершину нет (в программе вместо “∞” можно использовать очень большое число). Таким образом, вершина  $A$  уже рассмотрена и выделена серым фоном. В первый элемент массива  $P$  записан символ “x”, обозначающий начальную точку маршрута (в программе можно использовать несуществующий номер вершины, например, 0).

Из оставшихся вершин находим вершину с минимальным значением в массиве  $R$ : это вершина  $B$ . Теперь проверяем пути, проходящие через эту вершину: не позволят ли они сократить маршрут к другим, которые мы еще не посещали. Идея состоит в следующем: если сумма весов  $W[x,z]+W[z,y]$  меньше, чем вес  $W[x,y]$ , то из вершины  $x$  лучше ехать в вершину  $y$  не напрямую, а через вершину  $z$ :

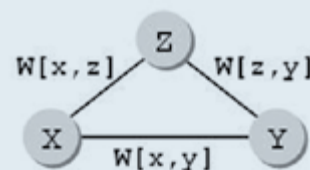


Рис. 30

Проверяем наш граф: ехать из  $A$  в  $C$  через  $B$  невыгодно (получается путь длиной 11 вместо 4), а вот в вершину  $D$  можно проехать (путь длиной 4), поэтому запоминаем это значение вместо ∞ в массиве  $R$  и записываем вершину  $B$  на соответствующее место в массиве  $P$  (“в  $D$  приезжаем из  $B$ ”):

	A	B	C	D	E	F
R	0	2	4	9	∞	∞
P	x	A	A	B		

Рис. 31

Вершины  $E$  и  $F$  по-прежнему недоступны.

Следующей рассматриваем вершину  $C$  (для нее значение в массиве  $R$  минимально). Оказывает-

ся, что через нее можно добраться до **Е** (длина пути 5):

	A	B	C	D	E	F
R	0	2	4	9	5	∞
P	x	A	A	B	C	

Рис. 32

Затем посещаем вершину **Е**, которая позволяет достигнуть вершины **Ф** и улучшить минимальную длину пути до вершины **Д**:

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	x	A	A	E	C	E

Рис. 33

После рассмотрения вершин **Ф** и **Д** таблица не меняется. Итак, мы получили, что кратчайший маршрут из **А** в **Ф** имеет длину 7, причем он приходит в вершину **Ф** из **Е**. Как же получить весь маршрут? Нужно просто посмотреть в массиве **P**, откуда лучше всего ехать в **Е**, — выясняется, что из вершины **С**, а в вершину **С** — напрямую из начальной точки **А**:

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	x	A	A	E	C	E

Рис. 34

Поэтому кратчайший маршрут **А-С-Е-Ф**. Обратите внимание, что этот маршрут “раскручивается” в обратную сторону, от конечной вершины к начальной. Заметим, что полученная таблица содержит *все* кратчайшие маршруты из вершины **А** во все остальные, а не только из **А** в **Ф**.

Алгоритм Дейкстры можно рассматривать как своеобразный “жадный” алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается вершина **X**, расстояние до которой от вершины **А** минимально. Однако можно доказать, что это расстояние — действительно минимальная длина пути от **А** до **X**. Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом **X** — это ближайшая не выбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в **X**, проходящие через еще не выбранные вершины, будут длиннее, поскольку все ребра имеют положительную длину. Таким образом, найденная длина пути из **А** в **X** — минимальная. После завершения алгоритма, когда все вершины выбраны, в массиве **R** находятся длины кратчайших маршрутов.

В программе объявим константу и переменные:

```
const N = 6;
var W: array[1..N,1..N] of integer;
    active: array [1..N] of boolean;
    R, P: array [1..N] of integer;
    i, j, min, kMin: integer;
```

Массив **W** — это весовая матрица, ее удобно вводить из файла. Логический массив **active** хранит

состояние вершин (просмотрена или не просмотрена); если значение **active[i]** истинно, то вершина активна (еще не просматривалась).

В начале программы присваиваем начальные значения (объяснение см. выше), сразу помечаем, что вершина 1 просмотрена (не активна), с нее начинается маршрут.

```
for i:=1 to N do begin
    active[i] := True;
    R[i] := W[1,i];
    P[i] := 1;
end;
active[1] := False;
P[1] := 0;
```

В основном цикле, который выполняется **N-1** раз (так, чтобы все вершины были просмотрены), среди активных вершин ищем вершину с минимальным соответствующим значением в массиве **R** и проверяем, не лучше ли ехать через нее:

```
for i:=1 to N-1 do begin
    { поиск новой рабочей вершины
      R[i] -> min }
    { максимальное целое число }
    min := MaxInt;
    for j:=1 to N do
        if active[j] and (R[j] < min) then begin
            min := R[kMin];
            kMin := j;
        end;
    active[kMin] := False;
    { проверка маршрутов через
      вершину kMin }
    for j:=1 to N do
        if R[kMin] + W[kMin,j] < R[j] then begin
            R[j] := R[kMin] + W[kMin,j];
            P[j] := kMin;
        end;
    end;
```

В конце программы выводим оптимальный маршрут в обратном порядке:

```
i := N;
{ для начальной вершины P[i]=0 }
while i <> 0 do begin
    write(i:5);
    i := P[i]; { переход к следующей вершине }
end;
```

Алгоритм Дейкстры, как мы видели, находит кратчайшие пути *из одной заданной вершины* во все остальные. Найти все кратчайшие пути (из любой вершины в любую другую) можно с помощью *алгоритма Флойда — Уоршелла*, основанного на той же самой идее сокращения маршрута (иногда бывает короче ехать через промежуточные вершины, чем напрямую):

```
for k:=1 to N
    for i:=1 to N
        for j:=1 to N
            if W[i,k] + W[k,j] < W[i,j] then
                W[i,j] := W[i,k] + W[k,j];
```

В результате исходная весовая матрица графа **W** размером **N** на **N** превращается в матрицу, хранящую длины оптимальных маршрутов. Для того чтобы найти сами маршруты, нужно использовать



еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив **P** в алгоритме Дейкстры.

### Некоторые задачи

С графами связаны некоторые классические задачи. Самая известная из них — задача коммивояжера (бродячего торговца).

**Задача 10.** Бродячий торговец должен посетить  $N$  городов по одному разу и вернуться в город, откуда он начал путешествие. Известны расстояния между городами (или стоимость переезда из одного города в другой). В каком порядке нужно посещать города, чтобы суммарная длина пути (или стоимость) оказалась наименьшей?

Эта задача оказалась одной из самых сложных задач оптимизации. По сей день известно только одно надежное решение — полный перебор вариантов, число которых равно факториалу от  $N-1$ . Это число с увеличением  $N$  растет очень быстро, быстрее, чем любая степень  $N$ . Уже для  $N = 20$  такое решение требует огромного времени вычислений: компьютер, проверяющий 1000 вариантов в секунду, будет решать задачу около четырех миллионов лет. Поэтому математики прилагали большие усилия для того, чтобы сократить перебор — не рассматривать те варианты, которые заведомо не дают лучших результатов, чем уже полученные. В реальных ситуациях нередко оказываются полезны приближенные решения, которые не гарантируют точного оптимума, но позволяют получить приемлемый вариант.

Приведем формулировки еще некоторых задач, которые решаются с помощью теории графов. Алгоритмы их решения достаточно сложны, при необходимости вы можете найти их в литературе или в Интернете.

**Задача 11 (о максимальном потоке).** Есть система труб, которые имеют соединения в  $N$  узлах. Один узел  $s$  является источником, еще один — стоком  $t$ . Известны пропускные способности каждой трубы. Надо найти наибольший поток от источника к стоку.

**Задача 12.** Имеется  $N$  населенных пунктов, в каждом из которых живет  $p_i$  школьников ( $i = 1, \dots, N$ ). Надо разместить школу в одном из них так, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным. В каком пункте нужно разместить школу?

**Задача 13 (о наибольшем паросочетании).** Есть  $M$  мужчин и  $N$  женщин. Каждый мужчина указывает несколько (от 0 до  $N$ ) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до  $M$ ), за которых она согласна выйти замуж. Требуется заключить наибольшее количество браков.

### Контрольные вопросы

1. Что такое граф?
2. Как обычно задаются связи узлов в графах?
3. Что такое матрица смежности?

4. Что такое петля? Как “увидеть” ее в матрице смежности?
5. Что такое путь?
6. Какой граф называется связным?
7. Что такое орграф?
8. Как по матрице смежности отличить орграф от неориентированного графа?
9. Что такое взвешенный граф? Как хранится в памяти информация о нем?
10. Что такое “жадный” алгоритм? Всегда ли он позволяет найти лучшее решение?

### Задачи

1. Напишите программу, которая вводит из файла весовую матрицу графа и строит для него минимальное остовное дерево.
2. Оцените асимптотическую сложность алгоритма построения минимального остовного дерева.
3. Напишите программу, которая вводит из файла весовую матрицу графа, затем вводит с клавиатуры номера начальной и конечной вершин, и затем определяет оптимальный маршрут.
4. Напишите программу, которая вводит из файла весовую матрицу графа и определяет длины всех оптимальных маршрутов с помощью алгоритма Флойда — Уоршелла.
5. Оцените асимптотическую сложность алгоритмов Дейкстры и Флойда — Уоршелла.
6. Напишите программу, которая решает задачу коммивояжера для пяти городов методом полного перебора. Можно ли использовать ее для 50 городов?

## Динамическое программирование

### Что такое динамическое программирование?

Мы уже сталкивались с последовательностью чисел Фибоначчи (см. учебник 10-го класса):

$$F_1 = F_2 = 1; F_n = F_{n-1} + F_{n-2} \text{ при } n > 2.$$

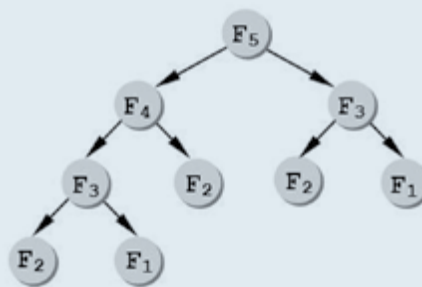


Рис. 35

Для их вычисления можно использовать рекурсивную функцию:

```
function Fib(N: integer): integer;  
begin  
  if N < 3 then  
    Fib := 1  
  else Fib := Fib(N-1) + Fib(N-2);  
end;
```

Каждое из этих чисел связано с предыдущими, вычисление  $F_5$  приводит к рекурсивным вызовам, которые показаны на рис. 35. Таким образом, мы два раза вычислили  $F_3$ , три раза  $F_2$  и два раза  $F_1$ . Рекурсивное решение очень простое, но оно неоптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные значения.

Какой же выход? Напрашивается такое решение — хранить все предыдущие числа Фибоначчи в массиве. Пусть этот массив называется  $F$ :

```
const N = 10;
var F: array[1..N] of integer;
```

Тогда для вычисления всех чисел Фибоначчи от  $F_1$  до  $F_N$  можно использовать цикл:

```
F[1] := 1; F[2] := 1;
for i := 3 to N do
    F[i] := F[i-1] + F[i-2];
```

**Динамическое программирование** — это способ решения сложных задач путем сведения их к более простым задачам того же типа.

Такой подход впервые систематически применил американский математик Р.Беллман при решении сложных многошаговых задач оптимизации. Его идея состояла в том, что оптимальная последовательность шагов оптимальна на любом участке. Например, пусть нужно перейти из пункта **A** в пункт **E** через один из пунктов **B**, **C** или **D** (числами обозначена “стоимость” маршрута):

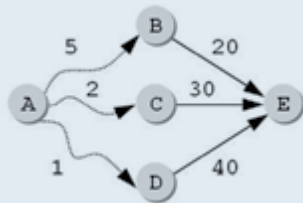


Рис. 36

Пусть уже известны оптимальные маршруты из пунктов **B**, **C** и **D** в пункт **E** (они обозначены сплошными линиями) и их “стоимость”. Тогда для нахождения оптимального маршрута из **A** в **E** нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут **A–B–E**, стоимость которого равна 25. Как видим, такие задачи решаются “с конца”, то есть решение начинается от конечного пункта.

В информатике динамическое программирование часто сводится к тому, что мы храним в памяти решения всех задач меньшей размерности. За счет этого удается ускорить выполнение программы. Например, на одном и том же компьютере вычисление  $F_{45}$  с помощью рекурсивной функции требует около 8 секунд, а с использованием массива — менее 0,01 с.

Заметим, что в данной простейшей задаче можно обойтись вообще без массива:

```
f2 := 1; f1 := 1;
for i := 3 to N do begin
    FN := f1 + f2;
    f2 := f1;
    f1 := FN;
end;
```

**Задача 14.** Найти количество  $K_N$  цепочек, состоящих из  $N$  нулей и единиц, в которых нет двух стоящих подряд единиц.

При больших  $N$  решение задачи методом перебора потребует огромного времени вычисления. Для того чтобы использовать метод динамического программирования, нужно

- 1) выразить  $K_N$  через предыдущие значения последовательности  $K_1, K_2, \dots, K_{N-1}, K_N$ ;
- 2) выделить массив для хранения всех предыдущих значений  $K_i (i = 1, \dots, N - 1)$ .

Самое главное — вывести рекуррентную формулу, выражающую  $K_N$  через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из  $N$  бит, первый элемент которой — 0.

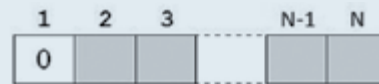


Рис. 37

Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длиной  $N$  с нулем в начале существует столько, сколько подходящих последовательностей длины  $N-1$ , то есть  $K_{N-1}$ . Если же первый символ — 1, то вторым обязательно должен быть 0, а остальная цепочка из  $N-2$  битов должна быть любой “правильной”. Поэтому подходящих последовательностей длиной  $N$  с единицей в начале существует столько, сколько подходящих последовательностей длины  $N-2$ , то есть  $K_{N-2}$ .

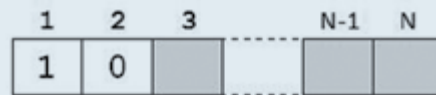


Рис. 38

В результате получаем  $K_N = K_{N-1} + K_{N-2}$ . Значит, для вычисления очередного числа нам нужно знать два предыдущих.

Теперь рассмотрим простые случаи. Очевидно, что есть две последовательности длиной 1 (0 и 1), то есть  $K_1 = 2$ . Далее, есть три подходящих последовательности длины 1 (00, 01 и 10), поэтому  $K_2 = 3$ . Легко понять, что решение нашей задачи — число Фибоначчи:  $K_N = F_{N+2}$ .

**Поиск оптимального решения**

**Задача 15.** В цистерне  $N$  литров молока. Есть бидоны объемом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все бидоны были заполнены и количество используемых бидонов было минимальным.

Для заданного  $N$  человек скорее всего будет решать задачу перебором вариантов. Наша задача осложняется тем, что требуется написать программу, которая решает задачу для любого введенного числа  $N$ .

Самый простой подход — заполнять сначала бидоны самого большого размера (6 л), затем — меньшие и т.д. Это так называемый “жадный” алгоритм. Как вы знаете, он не всегда приводит к оптимальному решению. Например, для  $N = 10$  “жадный”

алгоритм дает решение  $6 + 1 + 1 + 1 + 1$  — всего 5 бидонов, в то время как можно обойтись двумя (5 + 5).

Как и в любом решении, использующем динамическое программирование, главная проблема — составить рекуррентную формулу. Сначала определим оптимальное число бидонов  $K$ , а потом подумаем, как определить, какие именно бидоны нужно использовать.

Представим себе, что мы выбираем бидоны постепенно. Тогда последний выбранный бидон может иметь, например, объем 1 л, в этом случае  $K_N = 1 + K_{N-1}$ . Если последний бидон имеет объем 5 л, то  $K_N = 1 + K_{N-5}$ , а если 6 л —  $K_N = 1 + K_{N-6}$ . Так как нам нужно выбрать минимальное значение, то

$$K_N = 1 + \min(K_{N-1}, K_{N-5}, K_{N-6}).$$

Вариант, выбранный при поиске минимума, определяет последний добавленный бидон, его нужно сохранить в отдельном массиве  $P$ . Этот массив будет использован для определения количества выбранных бидонов каждого типа. В качестве начального значения берем  $K_0 = 0$ .

Полученная формула применима при  $N \geq 6$ . Для меньших  $N$  используются только те данные, которые есть в таблице. Например,

$$K_3 = 1 + K_2 = 3, \\ K_5 = 1 + \min(K_4, K_0) = 1.$$

На рис. 39 показаны массивы для  $N = 10$ .

N	0	1	2	3	4	5	6	7	8	9	10
K	0	1	2	3	4	1	1	2	3	4	2
P	0	1	1	1	1	5	6	1	1	1	5

Рис. 39

Как по массиву  $P$  определить оптимальный состав бидонов? Пусть, для примера,  $N = 10$ . Из массива  $P$  находим, что последний добавленный бидон имеет объем 5 л. Остается  $10 - 5 = 5$  л, в элементе  $P[5]$  тоже записано значение 5, поэтому второй бидон тоже имеет объем 5 л. Остаток 0 л означает, что мы полностью определили набор бидонов.

Можно заметить, что такая процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда — Уоршелла, по сути, используется метод динамического программирования.

**Задача 16 (Задача о куче).** Из камней весом  $p_i (i = 1, \dots, N)$  набрать кучу весом ровно  $W$  или, если это невозможно, максимально близкую к  $W$  (меньшую, чем  $W$ ).

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как 1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из  $N$  бит. При этом количество вариантов равно  $2^N$ , и при больших  $N$  полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее. Идея состоит в том, чтобы сохранять в массиве решения

всех более простых задач этого типа (при меньшем количестве камней и меньшем весе  $W$ ).

Построим матрицу  $T$ , где элемент  $T[i, w]$  — это оптимальный вес, полученный при попытке собрать кучу весом  $w$  из  $i$  первых по счету камней. Очевидно, что первый столбец заполнен нулями (при заданном нулевом весе никаких камней не берем).

Рассмотрим первую строку (есть только один камень). В начале этой строки будут стоять нули, а дальше, начиная со столбца  $p_1$ , — значения  $p_1$  (взяли единственный камень). Это простые варианты задачи, решения для которых легко подсчитать вручную. Рассмотрим пример, когда требуется набрать вес 8 из камней весом 2, 4, 5 и 7 единиц:

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0								
5	0								
7	0								

Рис. 40

Теперь предположим, что строки с 1-й по  $(i-1)$ -ю уже заполнены. Перейдем к  $i$ -й строке, то есть добавим в набор  $i$ -й камень. Он может быть взят или не взят в кучу. Если мы не добавляем его в кучу, то  $T[i, w] = T[i-1, w]$ , то есть решение не меняется от добавления в набор нового камня. Если камень с весом  $p_i$  добавлен в кучу, то остается “добрать” остаток  $w - p_i$  оптимальным образом (используя только предыдущие камни), то есть  $T[i, w] = T[i-1, w - p_i] + p_i$ .

Как же выбрать, “брать или не брать”? Проверить, в каком случае полученное решение будет больше (ближе к  $w$ ). Таким образом, получается рекуррентная формула для заполнения таблицы:

$$\text{при } w < p_i: \quad T[i, w] = T[i-1, w] \\ \text{при } w \geq p_i: \quad T[i, w] = \max(T[i-1, w], \\ T[i-1, w - p_i] + p_i)$$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке — слева направо:

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0	0	2	2	4	4	6	6	6
5	0	0	2	2	4	5	6	7	7
7	0	0	2	2	4	5	6	7	7

Рис. 41

Видим, что сумму 8 набрать невозможно, ближайшее значение — 7 (правый нижний угол таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом  $p_i$  не включен в набор, то  $T[i, w] = T[i-1, w]$ , то есть число в таблице не меняется при переходе на строку вверх. Начинаем с левого нижнего угла таблицы, идем вверх, пока значения в столбце равны 7. Последнее такое значение — для камня с весом 5, поэтому он и выбран. Вычитая его вес из суммы,

получаем  $7 - 5 = 2$ , переходим во второй столбец на одну строку вверх и снова идем вверх по столбцу, пока значение не меняется (равно 2). Так как мы успешно дошли до самого верха таблицы, взяли первый камень с весом 2.

Как мы уже отмечали, количество вариантов в задаче для  $N$  камней равно  $2^N$ , то есть алгоритм полного перебора имеет асимптотическую сложность  $O(2^N)$ . В данном алгоритме количество операций равно числу элементов таблицы, то есть сложность нашего алгоритма —  $O(N \cdot W)$ . Однако нельзя сказать, что он имеет линейную сложность, так как есть еще сильная зависимость от заданного веса  $W$ . Такие алгоритмы называют *псевдополиномиальными*, то есть “как бы полиномиальными”. В них ускорение вычислений достигается за счет использования дополнительной памяти для хранения промежуточных результатов.

**Количество решений**

**Задача 17.** У исполнителя Утроитель две команды, которым присвоены номера:

1. прибавь 1
2. умножь на 3

Первая из них увеличивает число на экране на 1, вторая — утраивает его. Программа для Утроителя — это последовательность команд. Сколько есть программ, которые число 1 преобразуют в число 20?

Прежде всего отметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнем с простых случаев, с которых будем начинать вычисления. Понятно, что для числа 1 существует только одна программа — пустая, не содержащая ни одной команды. Для числа 2 есть тоже только одна программа, состоящая из команды сложения. Если через  $K_N$  обозначить количество разных программ для получения числа  $N$  из 1, то  $K_1 = K_2 = 1$ .

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую  $K_N$  с предыдущими элементами последовательности  $K_1, K_2, \dots, K_N$ , то есть с решениями таких же задач для меньших  $N$ .

Если число  $N$  не делится на 3, то оно могло быть получено только последней операцией сложения, поэтому  $K_N = K_{N-1}$ . Если  $N$  делится на 3, то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить  $K_{N-1}$  (количество программ с последней командой сложения) и  $K_{N/3}$  (количество программ с последней командой умножения). В итоге получаем:

- если  $N$  не делится на 3:  $K_N = K_{N-1}$ ;
- если  $N$  делится на 3:  $K_N = K_{N-1} + K_{N/3}$ .

Остается заполнить таблицу для всех значений от 1 до  $N$ :

$N$	1	2	3	4	5	6	7	8	9	10	11
$K_N$	1	1	2	2	2	3	3	3	5	5	5
$N$	12	13	14	15	16	17	18	19	20		
$K_N$	7	7	7	9	9	9	12	12	12		

Заметим, что количество вариантов меняется только в тех столбцах, где  $N$  делится на 3, поэтому из всей таблицы можно оставить только эти столбцы:

$N$	1	3	6	9	12	15	18	21
$K_N$	1	2	3	5	7	9	12	15

Заданное число 20 попадает в последний интервал (от 18 до 21), поэтому ответ в данной задаче — 12. Для небольших значений  $N$  эту задачу легко решить вручную.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив  $K$ , индексы которого изменяются от 1 до  $N$ , и заполнить его по приведенным выше формулам:

```

K[1] := 1;
for i := 2 to N do begin
  K[i] := K[i-1];
  if i mod 3 = 0 then
    K[i] := K[i] + K[i div 3];
end;
    
```

Ответом будет значение  $K[N]$ .

**Задача 18 (Размен монет).** Сколькими различными способами можно выдать сдачу размером  $W$  рублей, если есть монеты достоинством  $p_i$  ( $i = 1, \dots, N$ )? Для того чтобы сдачу всегда можно было выдать, будем предполагать, что в наборе есть монета достоинством 1 рубль ( $p_1 = 1$ ).

Эта задача, так же как и задача о куче, решается только полным перебором вариантов, число которых при больших  $N$  очень велико. Будем использовать динамическое программирование, сохраняя в массиве решения всех задач меньшей размерности (для меньших  $N$  и меньшего числа монет). В матрице  $T$  значение  $T[i, w]$  будет обозначать количество вариантов сдачи размером  $w$  рублей ( $w$  изменяется от 0 до  $W$ ) при использовании первых  $i$  монет из набора. Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), также и при наличии только одного типа монет (напомним, что  $p_1 = 1$ ) есть тоже только один вариант. Поэтому нулевой столбец и первую строку таблицы можно заполнить сразу единицами. Для примера мы будем рассматривать задачу для  $w = 10$  и набора монет достоинством 1, 2, 5 и 10 рублей:

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2		1									
5			1								
10				1							

Рис. 42

Таким образом, мы определили простые базовые случаи, от которых “отталкивается” рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам, слева направо. Для вычисления  $T[i, w]$  предположим, что мы добавляем в набор монету достоинством  $p_i$ . Если сумма  $w$  меньше, чем  $p_i$ , то количество вариантов не увеличивается, и  $T[i, w] = T[i-1, w]$ . Если сумма



больше  $p_i$ , то к этому значению нужно добавить количество вариантов с “участием” новой монеты. Если монета достоинством  $p_i$  использована, то нужно учесть все варианты “разложения” остатка  $w - p_i$  на все доступные монеты, то есть  $T[i, w] = T[i-1, w] + T[i, w - p_i]$ . В итоге получается рекуррентная формула

$$\text{при } w < p_i: T[i, w] = T[i-1, w]$$

$$\text{при } w \geq p_i: T[i, w] = T[i-1, w] + T[i, w - p_i]$$

которая используется для заполнения таблицы:

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

Рис. 43

Ответ к задаче находится в правом нижнем углу таблицы.

Вы могли заметить, что решение этой задачи очень похоже на решение задачи о куче камней. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы. Использование методов динамического программирования позволяет ускорить решение за счет хранения промежуточных результатов, однако требует дополнительного расхода памяти.

### Контрольные вопросы

1. Что такое динамическое программирование?
2. Какой смысл имеет выражение “динамическое программирование” в теории многошаговой оптимизации?
3. Какие шаги нужно выполнить, чтобы применить динамическое программирование к решению какой-либо задачи?
4. За счет чего удастся ускорить решение сложных задач методом динамического программирования?
5. Какие ограничения есть у метода динамического программирования?

### Задачи

1. Напишите программу, которая определяет оптимальный набор бидонов в задаче с молоком. С клавиатуры или из файла вводится объем цистерны, количество типов бидонов и их размеры.
2. Напишите программу, которая решает задачу о куче камней заданного веса, рассмотренную в тексте параграфа.
3. Напишите программу, которая решает “задачу о ранце”: есть  $N$  предметов, для каждого из которых известны вес  $p_i$  ( $i = 1, \dots, N$ ) и стоимость  $c_i$  ( $i = 1, \dots, N$ ). В ранец можно взять предметы общим весом не более  $W$ . Напишите программу, которая определяет самый дорогой набор предметов, который можно унести в ранце.
4. У исполнителя Калькулятор две команды, которым присвоены номера:
  1. прибавь 1
  3. умножь на 4

Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число  $N$ , введенное с клавиатуры. Используйте сокращенную таблицу.

5. У исполнителя Калькулятор три команды, которым присвоены номера:

1. прибавь 1
2. умножь на 3
3. умножь на 4

Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число 1 в число  $N$ , введенное с клавиатуры.

### Самое важное в главе 6:

- Структура — это сложный тип данных, который позволяет объединить данные разных типов. Элементы структуры называют полями. При обращении к полям структуры используется точечная нотация: `<имя структуры>.<имя поля>`.
- Указатель — это переменная, в которой можно хранить адрес другой переменной заданного типа. В указателе можно запомнить адрес новой переменной, выделенной в памяти во время работы программы.
- Динамические массивы — это массивы, память для которых выделяется во время работы программы. Динамический массив в программе на языке Паскаль — это указатель, в который записывается адрес выделенного блока памяти. При записи такой переменной в файл сохранится только значение указателя, а значения элементов массива будут потеряны.
- Список — это упорядоченное множество, состоящее из переменного числа элементов, в котором введены операции вставки (включения) и удаления (исключения).
- Стек — это линейный список, в котором добавление и удаление элементов разрешается только с одного конца. Системный стек применяется для хранения адресов возврата из подпрограмм и хранения локальных переменных.
- Дерево — это структура данных, которая моделирует иерархию — многоуровневую структуру. Дерево — рекурсивная структура, поэтому для его обработки удобно использовать рекурсивные алгоритмы. Деревья используются в задачах поиска, сортировки, вычисления арифметических выражений.
- Граф — это набор узлов и связывающих их ребер. Информация о графе чаще всего хранится в виде матрицы смежности или весовой матрицы. Наиболее известные задачи, которые решаются с помощью теории графов, — поиск оптимальных маршрутов.
- Динамическое программирование — это метод, позволяющий ускорить решение задачи за счет хранения решений более простых задач того же типа. Для его использования нужно вывести рекуррентную формулу, связывающую решение задачи с решением подобных задач меньшей размерности, и определить простые базовые случаи (условие окончания рекурсии).



## Методика обучения информатике в разноуровневых группах (10–11-е классы)

И.Н. Фалина,  
Е.В. Андреева,  
СУНЦ МГУ,  
Москва

► Действующему учителю информатики или преподавателю вуза часто приходится сталкиваться с ситуацией, когда во вновь сформированных классах или группах оказываются учащиеся с разным уровнем знаний и умений по информатике. Такая же ситуация возникает в группах дополнительного образования, на факультативных занятиях.

Преподаватели информатики СУНЦ МГУ им. М.В. Ломоносова (школа-интернат им. А.Н. Колмогорова) сталкиваются с подобной ситуацией каждый год: набор в нашу школу проводится в 10-й и 11-й классы на основе экзаменов по математике и физике (в физико-математические классы) или математике и химии/биологии (классы химического или биологического профиля). И в одном классе оказываются школьники с одинаковым достаточно высоким уровнем знаний по физике и математике и разным уровнем подготовки по информатике.

Уточним, что мы понимаем под “разным уровнем подготовки по информатике”. Программа изучения информатики в нашей школе построена таким образом, что практически все теоретические темы поддерживаются программированием. Поэтому при оценке входного уровня подготовки по информатике мы учитываем:

- 1) теоретические знания;
- 2) умение программировать на одном из алгоритмических языков;
- 3) уровень знакомства с основными “школьными” алгоритмами;
- 4) способность к алгоритмическому мышлению и его сформированность;
- 5) умение работать “с компьютером” в целом.

В ситуации “разноуровневых” учащихся учителя часто используют дифференцированный метод обучения. Дифференциация по уровню умственного развития, навыков или умений не получает в современной педагогике однозначной оценки; в ней, наряду с положительными, имеются и отрицательные аспекты. Мы в своей практике стараемся не использовать эту методику напрямую. Основная причина, по которой мы отказались от нее, — снижение развития потенциальных возможностей школьников как в “сильных”, так и в “слабых” группах, возникновение морально-психологического напряжения между школьниками из разных групп.

Для эффективного обучения каждого ученика мы разработали *методику выравнивающего и развивающего обучения информатике*. Под эффективным обучением мы понимаем успешное освоение учащимися программы по информатике, поддержание у них интереса к изучаемому предмету, развитие творческих способностей учащихся. Данная методика позволяет выровнять уровень знаний школьников по базовому курсу (выравнивающая функция), при этом поддерживается высокий уровень интереса к изучаемому материалу и приобретение новых знаний и умений или, как теперь принято говорить, компетенций (развивающая функция). Развивающая функция является ведущей по отношению к выравнивающей.

В основу нашей методики положены такие *принципы*, как:

- (1) принцип обучения на высоком уровне трудности;
- (2) принцип ведущей роли фундаментальных теоретических знаний;
- (3) принцип дидактической спирали;
- (4) использование специально разработанной для каждой темы системы задач концентрической структуры;
- (5) развивающая функция системы контроля;
- (6) использование программного средства автоматической проверки программ.

На схеме ниже показано, с учетом каких принципов организуется каждая из четырех основных частей учебного процесса (лекции, семинарские занятия, самостоятельная работа и система контроля).

Покажем на примерах, как мы реализуем каждый из этих принципов.

### 1. Принцип обучения на высоком уровне сложности

В нашей школе учатся ученики только 10-х и 11-х классов. На информатику отводится три часа в неделю: один час — лекция, два часа — семинарские занятия.

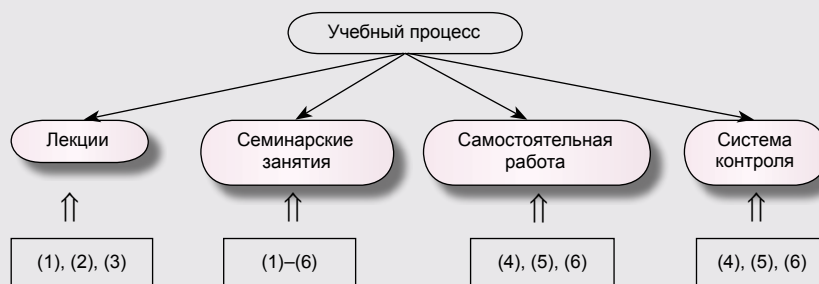
Теоретический материал на лекциях и практические задания на семинарских занятиях подбираются так, чтобы школьнику любого уровня знаний было интересно. Достигается это, в частности, изложением материала на высоком уровне сложности: ниже — не интересно, выше — непонятно. Уловить границу “сложности” весьма непросто. С одной стороны, надо изложить материал по программе (для тех, кто изучает эту тему первый раз), с другой стороны, избежать явного повторения материала (для тех, кто это уже изучал в средней школе). Одним из способов решения этой задачи является использование на лекциях **метода сравнительного анализа**.

**Пример 1.** В 10-м классе при изучении темы “Кодирование информации” мы вновь касаемся систем счисления. Для того чтобы удержать интерес класса к этой теме, можно излагать материал с использованием метода сравнительного анализа.

Известно, что все позиционные системы устройства практически одинаково [1]. Поэтому достаточно ввести понятие базиса позиционной системы (это понятие оказывается новым для многих учащихся) и показать, что запись чисел в различных позиционных системах строится по одним и тем же правилам. Следовательно, нет смысла изучать отдельно различные позиционные системы счисления. А для более глубокого усвоения этой темы целесообразно останавливаться на отличиях одной системы от другой, при этом рассказывать, как и где можно использовать (и используются) эти особенности. Как правило, мы рассматриваем и сравниваем три позиционных системы счисления (см. табл. ниже).

Далее мы показываем, что не существует и отдельных алгоритмов перевода из какой-либо позиционной системы счисления в десятичную — это просто разложение по базису. Причем мы получаем единый способ перевода и для целых чисел, и для конечных дробей. Для подтверждения этого факта можно сравнить любые два из приведенных ниже алгоритмов.

Позиционные системы счисления		
Традиционные, или $P$ -ичные	Нетрадиционные системы счисления	
$P$ -ичные с основанием $P = 2, 8, 16$ ( $P \geq 2$ )	Смешанные, или уравновешенные, системы счисления	Фибоначчиева система счисления
Базис — геометрическая прогрессия со знаменателем $P$ . Алфавит содержит $P$ цифр от 0 до $[P - 1]$	На примере троичной уравновешенной системы: Базис — геометрическая прогрессия со знаменателем 3. Алфавит — символы $\bar{1}, 0, 1$ , где $\bar{1} = -1$	Базис — числа Фибоначчи (начиная с $F_1 = 1$ ): 1, 2, 3, 5, 8, 13, 21 и т.д. Алфавит содержит две цифры: 0 и 1



**Алгоритм перевода чисел из P-ичной системы счисления в десятичную**

1) каждая цифра P-ичного числа переводится в десятичную систему;

2) полученные числа-цифры целой части нумеруются справа налево, начиная с нуля; цифры дробной части нумеруются слева направо индексами  $-1, -2, \dots$ ;

3) десятичное число, соответствующее каждой P-ичной цифре, умножается на  $P^k$ , где  $k$  — номер этого числа-цифры (п. 2), и результаты складываются, причем все арифметические действия проводятся в десятичной системе.

**Алгоритм перевода целых чисел из уравновешенной системы счисления в десятичную**

1) цифры “уравновешенного” числа нумеруются справа налево, начиная с нуля;

2) десятичное число, соответствующее каждой P-ичной цифре, умножается на  $3^k$ , где  $k$  — номер этого числа (п. 1), и результаты складываются, причем все арифметические действия проводятся в десятичной системе.

**Алгоритм перевода целых чисел из фибоначчьева системы счисления в десятичную**

1) напишем над каждой цифрой в фибоначчьева записи числа вес этого разряда (соответствующее число Фибоначчи), начиная с младшей цифры числа;

2) сложим все числа, стоящие над единицами. Полученное число будет десятичным эквивалентом фибоначчьева числа.

Теоретический материал необходимо подкрепить соответствующими практическими заданиями (см. табл. ниже).

А далее можно обратить внимание на такие интересные и важные факты.

**Факт первый.** В уравновешенных системах счисления отрицательные числа записываются без знака минус, и, следовательно, не надо вводить дополнительный код для выполнения арифметических операций с отрицательными числами.

Для изменения знака у представляемого числа нужно изменить знаки у всех его цифр. Например,  
 $-100_{10} = \bar{1} \bar{1}10 \bar{1}_{зур} = -3^4 - 3^3 + 1 \cdot 3^2 - 1 \cdot 3^0 = -81 - 27 + 9 - 1 = -100.$

При работе с этой системой счисления отсутствует проблема округления. Эта система более емкая по сравнению с двоичной системой. Например, в ячейку из восьми разрядов при использовании двоичной системы можно записать  $2^8 = 256$  различных целых чисел, а при использовании уравновешенной троичной системы —  $3^8 = 6561$  число, то есть в  $3^8 / 2^8 = 25,62$  раза больше (хотя физически “троичные” разряды изготовить сложнее).

Уравновешенная троичная система счисления использовалась в ЭВМ “Сетунь”.

**Факт второй.** Фибоначчьева система счисления обладает естественной избыточностью (множественное представление одного и того же числа). Это свойство было использовано при построении экспериментального Фибоначчи-процессора, который сам определял, произошел ли сбой при работе процессора [5].

**Замечание к примеру 1.** Проведение сравнительного анализа объектов (в нашем случае — позиционных систем счисления) требует от учителя достаточно высокой эрудиции и хорошего владения предметом. Это с одной стороны. С другой стороны, и от учащихся требуется напряженная работа на уроке. Наша цель — заинтересовать ученика, но интерес проявляется только при понимании “материала”, пусть даже не полном. Как говорят, на протяжении всего урока ученик “должен быть в теме”. Именно здесь и определяется тот порог трудности, выше которого ученики теряют связь в излагаемом материале, а ниже которого им не интересно.

Проводя последовательное сравнение нескольких типов позиционных систем счисления, достигаются следующие учебные цели:

1. **Формирование у школьников таких умственных действий, как понимание и применение материала.** При сравнительном анализе сопоставляются объекты, процессы или явления, обладающие однотипными характеристиками. Цель такого сравнения — выявить особенности каждого объекта. Делая акцент на особенностях алгоритмов перевода, выполнения арифметических операций в сравниваемых позиционных системах счисления, мы формируем у школьника “реперные сигналы”, которые помогут при необходимости вспомнить и применить нужные алгоритмы.

Двоичная система счисления	$1100100_2 = 100_{10}$	$2^6 + 2^5 + 2^2 = 64 + 32 + 4 = 100$
Восьмеричная система счисления	$144_8 = 100_{10}$	$1 \cdot 8^2 + 4 \cdot 8^1 + 4 \cdot 8^0 = 64 + 32 + 4 = 100$
Шестнадцатеричная система счисления	$64_{16} = 100_{10}$	$6 \cdot 16^1 + 4 \cdot 16^0 = 94 + 4 = 100$
Троичная уравновешенная система счисления	$11 \bar{1}01_{зур} = 100_{10}$	$1 \cdot 3^4 + 1 \cdot 3^3 - 1 \cdot 3^2 + 1 \cdot 3^0 = 81 + 27 - 9 + 1 = 100$
Фибоначчьева система счисления	$1000010011_{fib} = 100_{10}$ $1000010100_{fib} = 100_{10}$ $110010011_{fib} = 100_{10}$ $110010100_{fib} = 100_{10}$ $101110011_{fib} = 100_{10}$ $101110100_{fib} = 100_{10}$	Следует обратить внимание на различные представления числа 100 в фибоначчьева системе счисления. $89 \ 55 \ 34 \ 21 \ 13 \ 8 \ 5 \ 3 \ 2 \ 1$ $1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 = 89 + 8 + 2 + 1 = 100$



2. *Расширение границ подаваемого материала.* Излагая материал с использованием сравнительного анализа, учитель может делать “развивающие” отступления. Например, рассказать о троичной машине “Сетунь”, связать данную тему с архитектурой компьютера. Такие отступления наполняют практическим содержанием формальный курс информатики, работают на повышение интереса к предмету, мотивируют школьников осваивать конкретную тему.

## 2. Принцип ведущей роли фундаментальных теоретических знаний

Принцип обучения на высоком уровне трудности, принцип ведущей роли фундаментальных теоретических знаний и принцип дидактической спирали взаимосвязанны. Зачастую невозможно реализовать один без использования другого.

**Пример 2.** При изучении циклических конструкций мы даем школьникам такое задание:

*Сколько раз проработает цикл?*

```
h := 0.1;
while h <> 1 do
  h := h + 0.1;
```

Это задание идет на грани “уровня” трудности. Мы помогаем достичь эту грань ученику предварительным изложением фундаментальных знаний. В данном случае реализация этого тезиса состоит в том, что мы рассказываем о математических (теоретических) основах представления информации в компьютере и числовой информации в частности.

Описанный цикл будет работать бесконечно (вернее, до переполнения значения  $h$ ), так как переменная целого типа  $h$  никогда не станет равной 1. Связано это с тем, что “хорошая” десятичная дробь 0,1 не представима в компьютере точно (если речь идет о принятых в настоящее время способах представления вещественных чисел и двоичных компьютерах с конечным числом разрядов). Вернее, она представима или с избытком, или с недостатком, в зависимости от количества разрядов, отводимых на мантиссу в том или ином типе данных. Причина такого неточного представления кроется в том, что в двоичной системе счисления десятичная дробь 0,1 представима периодической дробью  $0,0(0011)_2$ .

Следовательно, чтобы правильно ответить на этот вопрос, школьник должен

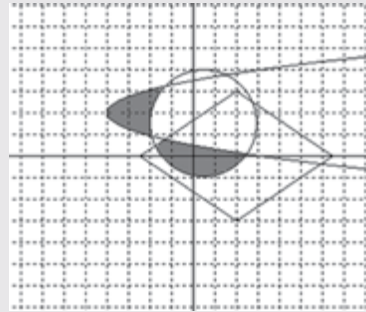
- 1) знать общие принципы представления вещественных чисел в компьютере;
- 2) понимать, какие вещественные числа представимы в компьютере точно, а какие — нет;
- 3) уметь переводить дробные числа из десятичной системы счисления в двоичную.

## 3. Принцип дидактической спирали

При изучении каждой темы мы стараемся излагать только ту часть материала, которая для учащихся будет новой по сравнению с тем, что они уже изучали. Для этого изложение ведется в обобщен-

ном виде и активно опирается на ранее полученные знания, в том числе и по другим предметам.

**Пример 3.** В практикум “Булева алгебра логики” включена задача: “Определить, принадлежит ли точка с координатами  $(x, y)$  заштрихованной фигуре”. Для этой задачи каждому школьнику дается индивидуальный рисунок.



А в практикум “Численные методы” включена задача: «Вычислить площадь заштрихованной фигуры, используя, например, метод Монте-Карло. Рисунок с заштрихованной фигурой взять из практикума “Булева алгебра логики”». При таком методическом подходе есть возможность сконцентрировать внимание на изучении сущности метода Монте-Карло и его реализации, не обращая внимание на способ определения попадания точки в требуемую область. При выполнении этого задания используются и различные знания, полученные в курсе математики.

**Пример 4.** При изучении основ программирования мы всегда проверяем, как учащиеся умеют применять знания, полученные на “предыдущем витке”. В качестве примера проанализируем задание из самостоятельной работы “Условный оператор” [4].

### Вариант 1

Запишите оператор присваивания, эквивалентный условному оператору

```
if a then x := b else x := c,
```

где все переменные — логического типа.

### Вариант 2

Запишите условный оператор, который эквивалентен оператору присваивания

```
x := a or b and c
```

и в котором не используются логические операции (все переменные — логические).

Для решения этих заданий школьникам необходимо соотнести знания таблиц истинности логических операций, приоритета выполнения логических операций с результатами выполнения условного оператора. Выполнив это задание, школьник выйдет на новый “виток знаний”, опираясь на ранее изученные факты.

Возможные варианты ответов.

**Вариант 1.** Ответом может являться любое выражение с переменными  $a$ ,  $b$  и  $c$ , которое равно  $b$ , если  $a$  равно **true**, и  $c$  иначе. Например:

```
x := a and b or not a and c.
```

**Вариант 2.** Приведенное выражение:  $x := a$  or  $b$  and  $c$  эквивалентно выражению  $x := a$  or

( $b$  and  $c$ ). В силу свойств логических операций, если переменная  $a$  имеет значение **true**, то и все выражение также равно **true**. Если же переменная  $a$  равна **false**, и хотя бы одна из переменных  $b$  или  $c$  равна **false**, то все выражение равно **false**.

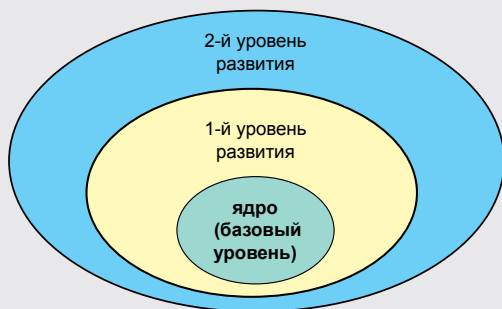
Требуемый оператор можно записать, например, так:

```
if a then x := true
else begin
  if b then if c then x := true
            else x := false
  else x := false
```

#### 4. Использование специально разработанной для каждой темы системы задач концентрической структуры

Основой успешного применения методики выравнивающего и развивающего обучения является специально разработанная система заданий для каждой темы (мы их называем практикумами). Система заданий (задач) строится на следующих принципах:

- 1) каждая задача должна быть методически значимой (не должно быть “случайных” задач);
- 2) каждая задача должна иметь концентрическую структуру;



3) концентрическая структура задачи проявляется на системе специально разрабатываемых тестов и критериев проверки;

4) в систему задач по каждой теме должны входить задачи разного уровня сложности;

5) задачи подбираются таким образом, чтобы ранее решенные задачи использовались при изучении последующих тем (дидактическая спираль). Это позволяет концентрировать внимание на разбираемой теме. Для задач, требующих решения на компьютере, это позволяет сокращать время написания и отладки программы (особенно для начинающих);

6) при проверке задач используется система автоматизированной проверки;

7) в начальных практикумах должны быть задания, решение которых не требует хороших навыков работы с компилятором, текстовым редактором. Эти задачи должны быть рассчитаны на понимание теоретических основ курса, для их решения требуются в основном время, ручка, бумага. Текст программы на языке программирования занимает всего несколько строк. Но сами задачи при этом

не должны быть банальными, “проходными”. При успешном решении такой задачи у начинающего ученика появляется дополнительная мотивация: “Я не самый плохой, я могу”, — так как ученик думает, что он решил “ту же задачу”, что и “продвинутый в информатике” товарищ по классу, а “продвинутому” тоже должно быть над чем подумать.

Обучение на высоком уровне трудности должно сопровождаться соблюдением меры трудности (у каждого школьника своя граница трудности, которая может перемещаться в обе стороны). Реализация меры трудности (установление границы трудности) достигается нами использованием задач концентрированной структуры (в основном это задачи по программированию).

Мы рассматриваем задачи концентрической структуры трех типов:

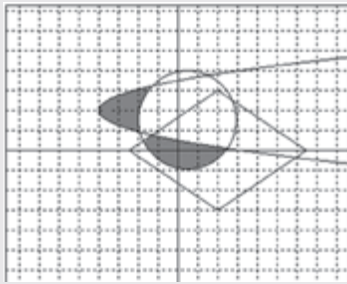
- 1) По одному и тому же условию задачи можно требовать ее выполнения на трех разных уровнях: базовом, уровень 1 и уровень 2 (см. примеры 5 и 6).
- 2) В задаче “незначительно”, с точки зрения ученика, меняется условие, но реально задача переходит в разряд более сложных (см. пример 7).
- 3) Задачу надо решать разными методами (для базового — решение разобранным на занятиях простейшим школьным алгоритмом, для 1-го и 2-го уровней — специально указанными алгоритмами) (см. пример 8).

**Пример 5.** Вернемся к задаче “Попадание точки в заштрихованную область” из примера 3. Для этой задачи каждому школьнику дается индивидуальный рисунок. В практикуме по изучению логических выражений, в том числе в языке программирования, условие задачи формулируется следующим образом:

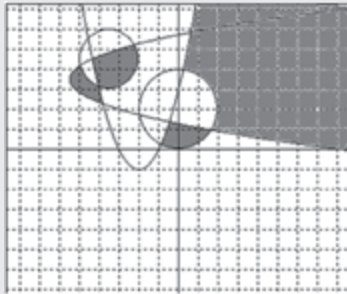
“На координатной плоскости изображены линии, описываемые уравнениями окружности, прямой, параболы, ромба, прямоугольника. Несколько областей, ограниченных этими линиями, заштрихованы. Определить, принадлежит ли точка с координатами  $(x, y)$  заштрихованной фигуре. Точки, лежащие на границе, не учитывать (для простоты решения их можно считать одновременно как принадлежащими заштрихованной области, так и не принадлежащими ей). Значения  $x$  и  $y$  вводятся с клавиатуры.

Ответ выдать в следующем виде: *True*, если точка с координатами  $(x, y)$  принадлежит заштрихованной фигуре, *False* — в противном случае.

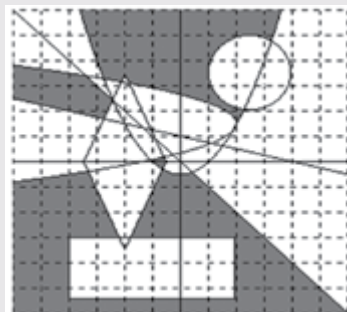
Обязательное требование: 1) операторы *If* и *Case* использовать нельзя; 2) для описания каждой области ввести свою булевскую переменную (это упрощает процесс отладки и поиска ошибок); основное логическое выражение конструировать только из таких переменных; 3) имена переменных должны быть mnemonic, например, *In\_Circle* (внутри окружности), *Upper\_Line* (над прямой)”.



Базовый уровень



1-й уровень сложности



2-й уровень сложности

При одной постановке задачи с помощью разных рисунков можно получить задания разных уровней сложности — причем не только варьируя количество и виды линий, ограничивающих искомую область. В ряде заданий “естественное”, с точки зрения школьника, описание требуемой области приводит к тому, что попутно оказываются описанными еще несколько областей, возможно, выходящих за пределы картинки. Это демонстрируется учащимся на этапе проверки (его функция визуализируется и масштабируется). Это задание также служит примером интеграции с курсом математики (тема “графики функций и уравнения линий”).

**Пример 6.** В начале 10-го класса в практикуме “Целочисленная арифметика” [3] есть задача “Для введенных с клавиатуры целых чисел  $m$  и  $n$  (указанного типа данных) вывести на экран результаты всех возможных операций над этими числами”.

**Базовый уровень:** при правильно выполненном задании начинающий школьник должен объяснить такой результат сложения чисел  $m = 100$ ,  $n = 40$  ( $m, n$  относятся к типу *shortint*, программа написана на Borland Pascal, установлены директивы компилятора \$R-,Q-):  $rez = m + n = -116$  тем, что математический результат выходит за пределы

диапазона представления чисел в выбранном типе данных.

На первом уровне сложности от ученика желательно услышать, что целые числа  $m$  и  $n$  в данном случае в компьютере занимают один байт и имеют следующее машинное представление:

$m = 100$	0	1	1	0	0	1	0	0
$n = 40$	0	0	1	0	1	0	0	0
Результат	1	0	0	0	1	1	0	0

При сложении этих чисел в старшем (знаковом) разряде получается единица, что соответствует отрицательному числу.

**2-й уровень:** сильного ученика можно дополнительно попросить попытаться объяснить машинное представление полученного отрицательного числа  $-116$  (до объяснения понятия “дополнительный код”).

После такого практического занятия теоретический материал по теме “кодирование целых, в том числе отрицательных, чисел в компьютере; дополнительный код” воспринимается гораздо лучше и вызывает интерес у учащихся.

**Пример 7.** В этом же начальном практикуме “Целочисленная арифметика” сильным и слабым школьникам предлагаются одни и те же задачи с “незначительными” изменениями. На все задачи практикума распространяется ограничение: в программах можно использовать только целочисленные переменные, операции “+”, “-”, “\*”, **div** и **mod** и оператор присваивания, в некоторых заданиях допустимо использование модуля (функция **abs**). Тем самым нельзя использовать условный оператор или операторы цикла.

**Базовый уровень:** Ввести с клавиатуры два целых числа (по модулю не превосходящих 1 000 000):  $m$  и  $n > 0$ . Если  $m$  делится на  $n$  или  $n$  делится на  $m$ , то вывести 1, в противном случае — любое другое число.

**1-й и 2-й уровни:** Ввести с клавиатуры два целых числа (по модулю не превосходящих 1 000 000):  $m$  и  $n$ . Если  $m$  делится на  $n$  или  $n$  делится на  $m$ , то вывести 1, в противном случае — любое другое число.

В условии задания для повышенного уровня сложности сняли ограничение  $m, n > 0$ . Задача стала принципиально сложнее. Если значение только одного числа равно 0, то результат — 1 (0 делится на любое другое число). Если оба введенных числа равны 0, то результатом должно быть любое число, отличное от 0 (0 на 0 не делится). Кроме того, в программе следует избегать деления на 0. В усложненном варианте правильно написанная программа должна проходить на следующих тестах:

Примеры входных данных	Примеры выходных данных
2 8	1
0 0	100
5 0	1
0 4	1

Ниже приведены тексты программ на Pascal для обеих формулировок данной задачи.



<p><b>Базовый уровень</b></p> <pre>var m, n, rez: integer; begin   readln(m, n);   rez := (n mod m)*(m mod n) + 1;   writeln(rez); end.</pre>
<p><b>1-й и 2-й уровни</b></p> <pre>var m, n, rez, m0, n0: integer; begin   readln(m, n);   m0 := 1 div (m*m + 1);   n0 := 1 div (n*n + 1);   rez := (n mod (m + m0))*(m mod (n + n0))     + n0 + m0;   writeln (rez) end.</pre>

При этом выражение вида  $1 \text{ div } (m*m + 1)$ , которое равно нулю везде, кроме  $m = 0$ , может быть обсуждено заранее на уроке.

Таким образом, начинающему ученику ставится цель — решить задачу правильно в целом, от “продвинутого” ученика требуется, чтобы задача прошла на всех тестах, в последнем случае срабатывает “развивающая” составляющая системы тестов. Обратим еще раз внимание, что для решения задач из такого практикума совсем не обязательно хорошо уметь программировать. Главное — составить правильный алгоритм, а текст программы из пяти строк набрать и отладить не составит труда. В результате ребята с разным уровнем подготовки сталкиваются при выполнении этого задания с практически одинаковыми проблемами — построение корректного арифметического выражения (что соответствует работе исполнителя с ограниченным набором команд).

**Пример 8.** Задача “Проверить число на простоту”. Решая эту задачу на базовом уровне, школьник должен научиться делать это перебором делителей до корня из  $N$  (ограничения по времени при автоматической проверке программы не позволят перебирать все делители или делители до  $N/2$ ). На первом уровне школьник должен познакомиться с методом “решето Эратосфена” и запрограммировать его, в идеале сравнив производительность с методом перебора делителей. На втором уровне школьник должен придумать (или найти самостоятельно) малоизвестный способ заполнения решета Эратосфена, при котором каждое составное число вычеркивается ровно один раз (так называемое “линейное решето Эратосфена”). Последний уровень сложности предлагается ученикам, активно участвующим в олимпиадах по информатике.

Приведем программу, соответствующую первому уровню выполнения задания и печатающую все простые числа, не превосходящие 10 000:

```
const maxn = 10000;
var a: array[2..maxn] of boolean;
    i, j, k: integer;
begin
  for i := 2 to maxn do
```

```
    a[i] := true;
  for i := 2 to maxn do
    if a[i] then
      begin
        writeln(i); {найдено очередное
                     простое число}
        j := i*i;
        {числа вида 2*i, 3*i, меньше j,
         уже вычеркнуты!!!}
        while j <= maxn do
          begin
            a[j] := false;
            j := j + i
          end
        end
      end
    end.
```

**Пример 9.** Задача сортировки массива. Уровень сложности реализуется проверкой на различных тестах и дополнительных требованиях. Например:

**Базовый уровень:** программа, написанная учащимся, должна правильно сортировать любую входную последовательность небольшого размера (до 1000 элементов).

**1-й и 2-й уровни:** к требованиям базового уровня добавляется требование подсчета сложности используемого алгоритма и сравнения полученного результата с теоретической сложностью.

Возможен такой вариант проверки задания по уровням сложности:

**Базовый уровень:** программа, написанная учащимся, должна правильно работать на любых входных данных при  $N$  от 1 (вырожденный случай) до 1000.

**1-й уровень:** школьник должен уметь писать эффективные алгоритмы сортировки (слиянием, “быструю”), работающие за одну секунду с последовательностями, содержащими до  $10^8$  элементов.

**2-й уровень:** школьник должен уметь доказывать сложность соответствующего алгоритма и строить примеры входных данных, при которых время работы алгоритма, в частности “быстрой” сортировки, будет наилучшим.

*Замечание к примерам 5–9.* При использовании задач концентрической структуры достигаются различные педагогические цели:

1) начинающему ученику показывают, что он решает примерно ту же задачу, что и все остальные ученики, это дает толчок к повышению мотивации обучения, ученик не чувствует себя “отстающим”. Для решения задачи не требуются хорошие навыки программирования;

2) сильным ученикам дается существенно более сложная задача, которая для них не будет очевидной;

3) использование задач концентрической структуры обеспечивает определенную *комфортность* учителю на уроке, так как в рамках одной и той же темы можно дать всего лишь 4–5 различных задач и при этом учесть все уровни подготовки учеников за счет проверки задач на различных тестах или с различными требованиями к программе.



## 5. Развивающая функция системы контроля

Этот принцип-требование можно сформулировать и по-другому: “*проверяя, развивай и не подавляй*”.

Как уже было сказано, обучение на высоком уровне трудности сопровождается соблюдением меры трудности, которая выражена в контроле качества усвоения. Главное в этом контроле — не оценка знаний и навыков посредством отметок, а дифференцированное и, возможно, более точное определение качества усвоения, его особенностей у разных учеников данного класса.

Наша система контроля основана на принципе развивающего обучения “в изучении программного материала идти вперед быстрым темпом”. Быстрый темп изучения — это отказ от топтания на месте, от однообразного повторения пройденного. Практическая реализация принципа изучения в быстром темпе подразумевает постоянный контроль за знаниями и умениями учащихся, так как без убежденности в полном усвоении материала всеми учениками нет смысла двигаться вперед. В таком случае этот принцип вырождается в спешку.

Для каждой темы выбирается свой вид контроля знаний. Но при любом способе проверки необходимо выполнять условия, которые позволяют в системе контроля акцентировать развивающую функцию для учащихся. Для этого необходимо выполнение следующих условий:

- ни одно задание не должно быть оставлено без проверки и оценивания со стороны преподавателя;
- незамедлительное сообщение результатов проверки;
- максимальное участие школьника в процессе проверки выполненного им задания.

Мы используем следующие виды контроля:

- *Тестирование на компьютерах.* Тестирование относится к виду контроля, при котором скорее выявляется, какие темы ученик не знает, чем глубина знаний по тем вопросам, на которые он ответил правильно. Составление вопросов для тестирования и, главное, вариантов возможных ответов требует тщательного продумывания. При всех недостатках тестирования этот вид контроля возможно и необходимо использовать, так как выполнение теста из 5–10 вопросов занимает 15–20 минут, и преподаватель, и ученик получают оценку моментально после окончания тестирования, кроме того, соблюдается объективность оценки.

- В контрольные работы необходимо включать “развивающие” вопросы и задачи, т.е. такие, которые непосредственно не разбирались на лекциях и семинарах, но для решения которых изложенных сведений достаточно. Заметим, что оценка “развивающего” вопроса не выносится за рамки оценки всей контрольной работы. Примером таких вопросов на контрольной работе по системам счисления могут быть следующие:

1) В каких системах счисления число  $17_p$  четное? (Решение: так как  $17_p = P + 7$ , то  $P$  должно быть нечетным. Кроме того, так как в записи числа при-

сутствует цифра 7,  $P$  должно быть больше 7.) Решая эту задачу, школьники должны также осознать, что четность — это свойство числа, а не системы счисления, и четность последней цифры — это всего лишь признак четности числа в десятичной системе счисления. Этот признак может оказаться несправедливым в других  $P$ -ичных системах счисления.

2) Выпишите базис 2–10-й системы счисления (или, например, смешанной системы счисления, принятой для записи времени по принципу “дни, часы, минуты, секунды, десятые доли секунды и т.д.”). На лекциях школьникам вводилось понятие базиса позиционной системы счисления, а также рассказывалось о  $P$ - $Q$ -ичных системах, из которых более подробно разбирались системы вида  $P^m = Q$ , например, 2–16-ричная. На контрольной работе школьнику фактически предлагается обобщить эти знания и выписать базис понятной, но не рассматриваемой ранее системы. Для 2–10-й системы, в которой каждая цифра десятичной системы записывается в виде двоичного четырехзначного числа, это последовательность 1, 2, 4, 8, 10, 20, 40, 80, 100, ...

Результатом такого подхода к подбору заданий для самостоятельных и контрольных работ является то, что на ЕГЭ по информатике у учащихся практически не возникает сложностей с выполнением заданий нового типа, предварительно не показанных в демоверсии.

- При проверке домашних заданий целесообразно придерживаться правила “одна неделя задержки — минус один балл”. Выполнение каждого задания-практикума рассчитано на определенное время, как правило, на 1–2 недели. Если ученик сдает задание в срок, то он получает ту отметку, которую заслуживает. Если ученик по каким-либо причинам (кроме болезни) не сдает свою работу в срок, то отрицательная отметка не выставляется. Но ученик знает, что если через неделю он сдает работу на “отлично”, то получает только “хорошо”, через две-три недели отметку выше тройки он не получит. Такой подход позволяет при необходимости увеличить время выполнения домашнего задания, но предупреждает накопление несданных работ и приучает к систематической работе.

- В течение учебного года мы дважды проводим семестровые *коллоквиумы* с привлечением выпускников, на которые выносятся основные теоретические вопросы лекций и семинаров. Основная цель такого мероприятия заключается в том, чтобы школьники перечитали и осмыслили весь пройденный материал. Формат коллоквиума близок к формату устного экзамена, опыт которого в последнее время, к сожалению, постепенно утрачивается, в том числе в вузах. В результате школьники, даже владея материалом, зачастую не могут ясно излагать свои мысли.

- В конце года все ученики сдают экзамен по информатике. Экзамен состоит из двух частей: компьютерный тест и решение задач по программированию. В тест включаются от 20 до 30 вопросов, затрагиваю-

щих все темы курса (как теоретический лекционный материал, так и практические темы семинарских занятий, в том числе по программированию). Как уже говорилось выше, тест в первую очередь выявляет, что учащийся не знает. Поэтому в случае неудовлетворительного выполнения теста (менее 50% правильных ответов) ученик отправляется на пересдачу. Причем мы не рассматриваем это как наказание, а как указание на необходимость повторить весь материал еще раз. И ученики в большей части случаев успешно справляются с тестом (с уже новыми вопросами) при повторном его прохождении. За основу теста мы взяли вопросы американского курса Computer Science-I, дополнив за годы работы его своими вопросами.

### 6. Использование программного средства автоматической проверки программ

В профильных классах тема “Алгоритмизация и программирование” на протяжении всех лет существования школьной информатики оставалась в учебном плане: менялись изучаемые языки программирования, менялось количество часов, однако методика изучения этой темы существенно почти не менялась.

Парадоксально, но факт: при обучении “программированию” сам компьютер используется в недостаточной степени, а методы обучения и используемые программные средства зачастую остаются на уровне 90-х годов прошлого столетия. В значительной степени снять данную проблему может использование специальных программных средств автоматической проверки. Мы называем такое ПС “Электронный задачник по программированию”.

Тема “Алгоритмизация и программирование” обычно изучается серьезно в старших классах (9–11-е классы). По многим причинам именно методика преподавания этой темы имеет исключительно большое значение в приобретении школьниками новых знаний, в выработке умений и навыков составлять алгоритмы решения задачи, а затем и записывать их в виде программ на каком-либо алгоритмическом языке. Кроме того, 30–40% заданий ЕГЭ относятся к теме “Алгоритмизация и программирование”, и не учитывать этого нельзя.

Мы включили в методику обучения программированию использование “Электронного задачника” (ЭЗ) по следующим причинам:

1. Повышается мотивация учащихся (у школьников появляется азарт, им очень хочется “сдать” свою задачу ЭЗ, “победить” ЭЗ).
2. Из банка задач в идеале можно составлять индивидуальное задание для каждого ученика и, что не менее важно, ЭЗ оперативно проверит выполненное задание.
3. В определенном смысле преодолевается проблема “копирования” решения задачи.
4. Осуществляется качественная проверка выполненного задания.
5. На уроке поддерживается высокий темп работы каждого ученика.

6. ЭЗ используется при самостоятельной работе учащихся (выполнение домашнего или дополнительного задания, подготовка к олимпиадам).

### Результаты применения методики выравнивающего и развивающего обучения информатике

- 1) Достаточно высокий результат усвоения в пределах требований к обязательным результатам обучения;
- 2) За пределами этих обязательных требований конечно же мы имеем различие в учебных результатах и успехи в олимпиадах по информатике и программированию;
- 3) При освоении обязательного базового курса у всех учеников постоянно поддерживается интерес к изучаемому материалу;
- 4) Корректируется самооценка учащихся;
- 5) У учащихся формируются навыки самостоятельной работы;
- 6) Преподаватель имеет возможность работать в режиме “индивидуальной работы” практически на одном и том же методическом материале;
- 7) Индивидуальная работа в условиях классно-урочной системы накладывает на учителя очень жесткие требования к его реакции, памяти, собранности; разработка и использование системы задач с концентрической структурой позволяет уменьшить нагрузку на учителя во время урока;
- 8) При разработке описанной выше системы задач повышается методическая грамотность учителя;
- 9) Ученики нашей школы сдают ЕГЭ по информатике с результатом существенно выше среднего по стране (около 90 баллов, при более чем 100 сдающих).

Наконец, отвлечемся от сухого изложения и приведем цитату нашего ученика, недавно появившуюся на форуме школы: “Информатика — это предмет, который преподается в школе замечательно, причем во всех классах”. Наверное, такую оценку учащихся и можно считать основным результатом нашей работы.

### Литература

1. Андреева Е.В., Босова Л.Л., Фалина И.Н. Математические основы информатики. Учебное пособие. М.: БИНОМ. Лаборатория знаний, 2009.
2. Андреева Е.В., Босова Л.Л., Фалина И.Н. Математические основы информатики. Методическое пособие. М.: БИНОМ. Лаборатория знаний, 2007.
3. Фалина И.Н., Богомолова Т.С., Большакова Е.А., Гуцин И.С., Шухардина В.А. Алгоритмизация и программирование. Сборник контрольных работ с решениями (9–11-е классы). М.: КУДИЦ-ПРЕСС, 2007.
4. Андреева Е.В. Программирование — это так просто, программирование — это так сложно. Современный учебник программирования. М.: МЦНМО, 2009.
5. Фалина И.Н. Кодирование информации: фибonacci система и компьютер. Информатика № 15/2011.

## MimioClassroom — комплексное решение для оборудования интерактивного класса

Представьте себе набор простых в использовании инструментов, объединенных в единую систему, которая содействует созидательному процессу обучения и позволяет ученикам активно и увлеченно учиться. Инструменты, которые значительно расширяют возможности учителя и упрощают его работу. **MimioClassroom** помогает сосредоточиться на преподавании, не беспокоясь о технологиях. А процесс обучения становится более наглядным и привлекательным для учеников.

Интерактивные инструменты **MimioClassroom** приносят простоту в обучение с помощью современных технологий. Они снабжены обширной библиотекой готовых к использованию уроков и мероприятий по всем предметам. Кроме того, можно использовать материалы, созданные для интерактивных инструментов других производителей. Установка и освоение инструментов **MimioClassroom** не вызывает трудностей и не отнимает лишнего времени.



**MimioTeach.** Почему бы не сделать обычную школьную доску интерактивной, вместо того чтобы покупать интерактивную доску? **MimioTeach** позволяет обучать в интерактивном режиме, используя уже имеющуюся в классе меловую или маркерную доску. Функциональность, аналогичная интерактивной доске, при существенно более низкой цене и отсутствии затрат на установку. Работает с уже установленным в классе оборудованием.



**MimioVote** обеспечивает проведение тестирования учащихся, упрощает оценивание, выставление отметок и ведение статистики. Поддерживаются различные режимы проведения тестирования. Вы можете задавать различный темп и условия тестирования, а программное обеспечение системы сохранит журнал ответов и оценок как по ученикам, так и в целом по классам. Использование готовых тестовых наборов, удобные инструменты создания и импортирования тестов облегчают подготовку к уроку и экономят ваше время.



**MimioView** позволяет использовать изображения и видео, полученные документ-камерой в интерактивных уроках. Демонстрируйте ученикам уже имеющиеся у вас документы, рисунки, репродукции и трехмерные объекты. Подключите **MimioView** к микроскопу и познакомьте детей с тайнами микромира. Эта документ-камера прекрасно работает также с движущимися объектами.



**MimioCapture** считывает с доски заметки и рисунки, сделанные легкочищаемыми маркерами. Упростите учебный процесс, используя возможность сохранять все, что было нарисовано и написано на доске. **MimioCapture** сохранит в памяти все рисунки и надписи, что позволяет неоднократно возвращаться к этому в дальнейшем. У вас появляется возможность раздать ученикам конспект урока, а сохраненные изображения сформируют уникальную библиотеку контента, совместимую с другими интерактивными устройствами **Mimio**.



**MimioPad.** Позволяет преподавателю свободно передвигаться по классу в процессе работы с доской. Обеспечивает все возможности управления интерактивными устройствами **Mimio** из любой точки класса.

Гарантийный срок на все оборудование **Mimio** составляет 5 лет с момента покупки.

Пользователям бесплатно предоставляются разнообразные возможности обучения и повышения квалификации, очные и заочные.

Узнайте больше об интерактивных технологиях **Mimio**, посетите наш сайт:

[www.mimioclass.ru](http://www.mimioclass.ru)

или позвоните:

**8 (800) 5555-33-0**

*Звонок по России бесплатный*





## Вопросы по информатике для конкурсов “Что? Где? Когда?” и “Брейн-ринг”

Д.М. Златопольский,  
Москва

44

декабрь 2011 / ИНФОРМАТИКА

1. Первая механическая вычислительная машина “Mark I”, построенная в 1944 году, официально называлась “ASCC”, а первая электронная вычислительная машина, изготовленная в Европе в 1949 году, называлась “EDSAC”. Буква “А” в этих двух аббревиатурах соответствовала слову “Automatic” (автоматическая). А с каким словом были связаны буквы “С” (си) в конце аббревиатур ASCC и EDSAC?

**Примечание.** Аббревиатуры ASCC и EDSAC ведущий показывает участникам конкурса изображенными на табличках.

2. В программировании есть понятие “конкатенация”. Так называют операцию соединения (сцепления) нескольких величин строкового типа в одну (от англ. *concatenate* — “сцеплять”). А как называется обратная операция — разбиение строки на несколько частей?

3. Вы, конечно, знаете, что такое “тотализатор”. А теперь послушайте отрывок из научно-фантастического рассказа, написанного знаменитым писателем-фантастом Жюлем Верном в конце XIX века: “Когда тотализатор закончил подсчет, прибыль выразилась в двухстах пятидесяти тысячах долларов за истекший день — на пятьдесят тысяч больше, чем накануне”<sup>1</sup>.

Что за “тотализатор” имел в виду писатель?

4. Как известно, двумерный массив называют “матрица”. А как называют двумерный массив, в котором число строк равно числу столбцов?

5. Одни из первых советских электронных вычислительных машин назывались “МЭСМ” и “БЭСМ”. Что означали вторая буква “Э”, третья буква “С” и четвертая буква “М” в этих двух аббревиатурах, вы, конечно, уже поняли. А что означали первые буквы в них — “М” и “Б”?

6. Ведущий (обращаясь к одному из участников конкурса): “Вы знаете, что такое *бескозырка моряка*?”

(После ответа.) А какое отношение имеет бескозырка к информатике?”

### Вариант вопроса

Что общего между двумя предметами, изображенными на картинке:

<sup>1</sup> Отрывок приведен в книге В.В. Шилова “Удивительная история информатики и автоматике”. М.: ЭНАС, 2011.





#### Ответы

1. Со словом “Calculator” (калькулятор). Слово “компьютер” стало употребляться позднее.

2. Декатенация (от англ. *decatenate* — “разъединять”).

**Примечание.** Учащиеся должны вспомнить такие понятия, как “демонтаж”, “декодирование”, “дешифрование”, “декомпозиция” и т.п.

3. В отрывке “тотализатор” — это счетное устройство, а точнее, суммирующее устройство. Ответы “человек, проводивший расчеты” и т.п. — неверные, так как в условии фигурирует слово “Что”, а не “Кто”.

4. Квадратная матрица.

5. Первая буква “М” в аббревиатуре МЭСМ означала “малая” (МЭСМ — “малая электронная счетная машина”. Первая буква “Б” в аббревиатуре БЭСМ была связана со словом “большая” (предполагается также, что она могла быть связана со словом “быстродействующая”).

6. Бескозырки использовались (и используются) при передаче информации на флоте на основе русской семафорной азбуки. При такой передаче каждой букве соответствует определенное положение рук с флажками. При отсутствии флажков применяются бескозырки.

*Ответ на вариант вопроса.* Пара одного из этих предметов используется при передаче информации на флоте на основе русской семафорной азбуки (см. выше).

## Информатика и школьно-письменные принадлежности

Задание может предъявляться учащимся в одном из пяти уровней сложности.

### Уровень 1

Назовите термины, связанные с компьютерами или информатикой, которые являются также названиями школьно-письменных принадлежностей. Дайте определения этих терминов в контексте информатики.

*Возможные ответы:* блокнот, калькулятор, карандаш, карта, книга, ластик, линейка, лист, папка, портфель, файл.

### Комментарии к терминам

“Блокнот” — встроенная в операционную систему Microsoft Windows программа.

“Калькулятор” — встроенная в операционную систему Microsoft Windows программа.

“Карандаш” — инструмент графического редактора Microsoft Paint.

Карта (карта памяти — таблица, указывающая распределение частей программы в памяти).

Книга — документ, созданный с помощью электронной таблицы Microsoft Excel или т.п.

“Ластик” — инструмент графического редактора Microsoft Paint.

Линейка — часть окна текстового редактора, используемая для установки полей, отступов и т.п.

Лист — часть книги в электронной таблице Microsoft Excel или т.п.

Папка — поименованная группа файлов, объединенных по какому-то признаку.

“Портфель” — изображение на кнопке **Вставить** в текстовом редакторе Microsoft Word.

Файл — совокупность данных на носителе, снабженная именем.

### Уровень 2

Следующие определения относятся к школьно-письменным принадлежностям, но соответствующие термины используются и в информатике. Назовите эти термины и дайте им определения в контексте информатики.

1. Тетрадь или книжечка для записей, состоящая из отрывных листков.

2. Небольшой прибор для вычислений.

3. Письменная принадлежность — деревянная палочка со стержнем.

4. Чертеж поверхности Земли, небесного тела или звездного неба.

5. Произведение печати в виде переплетенных листов с каким-нибудь текстом.

6. Кусочек специально обработанной резины для стирания написанного, нарисованного.

7. Планка для вычерчивания прямых линий, для измерений.

8. Тонкий плоский кусок бумаги.

9. Загибающаяся с краев обложка, в которую вкладываются бумаги, рисунки.

10. Жесткая прямоугольная сумка с закидывающейся крышкой и запором для ношения бумаг, книг.

11. Пакетик из тонкой прозрачной пленки для вкладывания бумаг.

*Ответы.* 1. Блокнот. 2. Калькулятор. 3. Карандаш. 4. Карта. 5. Книга. 6. Ластик. 7. Линейка. 8. Лист. 9. Папка. 10. Портфель. 11. Файл.

Определения терминов в контексте информатики приведены в задании уровня 1.

### Уровень 3

Перед вами — таблица с двумя колонками, в которых представлены определения некоторых терминов. В первой колонке приведены определения, касающиеся информатики, во второй — школьно-письменных принадлежностей. Необходимо найти пары определений (по одному из каждой колонки), относящиеся к терминам, являющимся омонимами.

1. Встроенная в операционную систему Microsoft Windows программа для подготовки текстов	1. Тонкий плоский кусок бумаги
2. Инструмент графического редактора Microsoft Paint	2. Произведение печати в виде переплетенных листов с каким-нибудь текстом
3. Таблица, указывающая распределение частей программы в памяти	3. Небольшой прибор для вычислений
4. Документ, созданный с помощью электронной таблицы Microsoft Excel или т.п.	4. Жесткая прямоугольная сумка с закидывающейся крышкой и запором для ношения бумаг, книг
5. Инструмент графического редактора Microsoft Paint	5. Чертеж поверхности Земли, небесного тела или звездного неба
6. Часть окна текстового редактора, используемая для установки полей, отступов и т.п.	6. Пакетик из тонкой прозрачной пленки для вкладывания бумаг
7. Часть книги с данными в электронной таблице Microsoft Excel или т.п.	7. Тетрадь или книжечка для записей, состоящая из отрывных листков
8. Встроенная в операционную систему Microsoft Windows программа для ручных вычислений	8. Планка для вычерчивания прямых линий, для измерений
9. Поименованная группа файлов, объединенных по какому-то признаку	9. Кусочек специально обработанной резины для стирания написанного, нарисованного
10. Изображение на кнопке <b>Вставить</b> в текстовом редакторе Microsoft Word	10. Письменная принадлежность — деревянная палочка со стержнем
11. Совокупность данных на носителе, снабженная именем	11. Загибающаяся с краев обложка, в которую вкладываются бумаги, рисунки

*Ответы:* 1 — 7 (“Блокнот” — блокнот). 2 — 10 (“Карандаш” — карандаш). 3 — 5 (карта). 4 — 2 (книга). 5 — 9 (“Ластик” — ластик). 6 — 8 (линейка). 7 — 1 (лист). 8 — 3 (“Калькулятор” — калькулятор). 9 — 11 (папка). 10 — 4 (“Портфель” — портфель). 11 — 6 (файл).

Первыми указаны номера определений в левой колонке.

#### Уровень 4

Термины, соответствующие следующим определениям, связаны также со школьно-письменными принадлежностями. Назовите эти термины.

1. Встроенная в операционную систему Microsoft Windows программа для подготовки текстов.
2. Встроенная в операционную систему Microsoft Windows программа для ручных вычислений.
3. Инструмент графического редактора Microsoft Paint.
4. Таблица, указывающая распределение частей программы в памяти.

5. Документ, созданный с помощью электронной таблицы Microsoft Excel или т.п.
  6. Инструмент графического редактора Microsoft Paint.
  7. Часть окна текстового редактора, используемая для установки полей, отступов и т.п.
  8. Часть книги с данными в электронной таблице Microsoft Excel или т.п.
  9. Поименованная группа файлов, объединенных по какому-то признаку.
  10. Изображение на кнопке **Вставить** в текстовом редакторе Microsoft Word.
  11. Совокупность данных на носителе, снабженная именем.
- Ответы.* 1. “Блокнот”. 2. “Калькулятор”. 3, 6. “Карандаш”, “Ластик”. 4. Карта (памяти). 5. Книга. 7. Линейка. 8. Лист. 9. Папка. 10. Портфель. 11. Файл.

#### Уровень 5

Ниже приведены определения терминов, используемых в информатике и одновременно относящихся к школьно-письменным принадлежностям. Назовите эти термины.

1. Встроенная в операционную систему Microsoft Windows программа для подготовки текстов. — Тетрадь или книжечка для записей, состоящая из отрывных листков.
2. Инструмент графического редактора Microsoft Paint. — Письменная принадлежность, деревянная палочка со стержнем.
3. Таблица, указывающая распределение частей программы в памяти. — Чертеж поверхности Земли, небесного тела или звездного неба.
4. Документ, созданный с помощью электронной таблицы Microsoft Excel или т.п. — Произведение печати в виде переплетенных листов с каким-нибудь текстом.
5. Инструмент графического редактора Microsoft Paint. — Кусочек специально обработанной резины для стирания написанного, нарисованного.
6. Часть окна текстового редактора, используемая для установки полей, отступов и т.п. — Планка для вычерчивания прямых линий, для измерений.
7. Часть книги с данными в электронной таблице Microsoft Excel или т.п. — Тонкий плоский кусок бумаги.
8. Встроенная в операционную систему Microsoft Windows программа для ручных вычислений. — Небольшой вычислительный прибор.
9. Поименованная группа файлов, объединенных по какому-то признаку. — Загибающаяся с краев обложка, в которую вкладываются бумаги, рисунки.
10. Изображение на кнопке **Вставить** в текстовом редакторе Microsoft Word. — Жесткая прямоугольная сумка с закидывающейся крышкой и запором для ношения бумаг, книг.
11. Совокупность данных на носителе, снабженная именем. — Пакетик из тонкой прозрачной пленки для вкладывания бумаг.

- Ответы.* 1. “Блокнот” — блокнот. 2. “Карандаш” — карандаш. 3. Карта. 4. Книга. 5. “Ластик” — ластик. 6. Линейка. 7. Лист. 8. “Калькулятор” — калькулятор. 9. Папка. 10 “Портфель” — портфель. 11. Файл.

# Годовая подшивка газеты «ИНФОРМАТИКА» на компакт-диске

## ПОЛНАЯ ПОДБОРКА МАТЕРИАЛОВ ЗА 2010 ГОД

А ТАКЖЕ ТЕМАТИЧЕСКИЕ СБОРНИКИ  
И ПОДШИВКИ ДРУГИХ ГАЗЕТ ИД «ПЕРВОЕ СЕНТЯБРЯ»



Удобная система навигации и поиска: материалы можно выбрать по тематике, рубрике или по номеру газеты.

Для пользователей любого уровня: включая и работой — не требуются установка и место на винчестере.

Компакт-диск пригоден для работы на компьютерах даже устаревшей конфигурации (Windows-95 и выше).

Стоимость диска включает доставку. Рассылка производится только на территории РФ.

### КУПОН ✂

ЗАПОЛНЯЕТСЯ ПЕЧАТНЫМИ БУКВАМИ!

ФАМИЛИЯ

ИМЯ

ОТЧЕСТВО

ИНДЕКС  АДРЕС

### ЭТИ ДИСКИ МОЖНО ПРИОБРЕСТИ:

- заполнив купон и отправив его в конверте с пометкой «Книга — почтой» по адресу:  
ИД «Первое сентября», ул. Киевская, д. 24, г. Москва, 121165
- заказав по телефону: (499) 249-47-58
- заказав по электронной почте: [podpiska@1september.ru](mailto:podpiska@1september.ru)
- заказав на сайте: [www.1september.ru](http://www.1september.ru)

### ТЕМАТИЧЕСКИЕ СБОРНИКИ

Цена за один диск с доставкой – 399 руб.

Газета «Начальная школа»  
«50 лет системе Л.В. Занкова»

«1001 ёлка на Новый год»

Газета «Школьный психолог»  
«Тренинг в теории и на практике»

Газета «Школьный психолог»  
«Тест со всех сторон»

Газета «Литература»  
«Консультации по темам экзаменационных сочинений»

Цены действительны  
до 31 декабря 2011 года

	2003 г.	2004 г.	2005 г.	2006 г.	2007 г.	2008 г.	2009 г.	2010 г.
Цена за один диск с доставкой	299 руб.	299 руб.	299 руб.	299 руб.	399 руб.	399 руб.	499 руб.	699 руб.
Английский язык	x	x	x	x	шт.	шт.	шт.	шт.
Библиотека в школе	x	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Биология	шт.	шт.	шт.	шт.	шт.	шт.	шт.	шт.
География	шт.	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Дошкольное образование	x	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Здоровье детей	x	x	шт.	шт.	шт.	шт.	шт.	шт.
Информатика	x	x	x	x	x	x	x	шт.
Искусство	x	x	x	шт.	шт.	шт.	шт.	шт.
История	шт.	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Классное руководство и воспитание школьников	x	x	x	x	x	шт.	шт.	шт.
Литература	шт.	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Математика	x	x	x	x	x	x	шт.	шт.
Начальная школа	x	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Немецкий язык	x	x	x	x	шт.	шт.	шт.	шт.
Русский язык	шт.	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Спорт в школе	x	x	шт.	шт.	шт.	шт.	шт.	шт.
Управление школой	x	x	шт.	шт.	шт.	шт.	шт.	шт.
Химия	шт.	шт.	шт.	шт.	шт.	шт.	шт.	шт.
Физика	x	x	шт.	шт.	шт.	шт.	шт.	шт.
Французский язык	x	x	x	x	шт.	шт.	шт.	шт.
Школьный психолог	шт.	шт.	шт.	шт.	шт.	шт.	шт.	шт.



## ШКОЛА ПРОГРАММИРОВАНИЯ

Еще раз о поразрядных логических  
и сдвиговых операциях

► В статье [1] рассказывалось о логических операциях, которые выполняются над числами в процессоре компьютера (где они представлены в двоичном виде). Среди них — команда **AND** (русский вариант — **И**).

В отличие от арифметических операций над двумя операндами логические команды являются *поразрядными*. Например, при сложении двух двоичных цифр возможен перенос в старший разряд, а при логических операциях все разряды рассматриваются изолированно друг от друга. Разумеется, действия над всеми разрядами выполняются параллельно и одновременно. Правила выполнения логической операции **AND** в каждом разряде представлены в таблице:

X (данные)	Y (маска)	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

**Примечание.** Первый операнд условно назван “данные”, а второй — “маска”.

Например, при  $X = 101011$  и  $Y = 1101$  имеем

$$X \text{ AND } Y = 1001,$$

$$X \text{ OR } Y = 101111,$$

$$X \text{ XOR } Y = 100110.$$

Анализ показывает, что, применив маску, равную 1, к любому двоичному числу, можно выделить (получить) младший разряд этого числа (проверьте!).

В статье [1] описывалась также так называемая “сдвиговая” команда **SHR** (от *Shift Right* — сдвинуть вправо). В результате выполнения этой команды в регистрах процессора происходит поразрядный сдвиг двоичных цифр на заданное количество разрядов вправо. При этом соответствующее число

цифр в младших разрядах теряется, а освободившиеся старшие разряды — заполняются нулями. Например, если в 8 разрядах было представлено двоичное число:

1	1	0	0	1	0	0	1
---	---	---	---	---	---	---	---

то после сдвига вправо на два разряда оно примет вид:

0	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

Несмотря на то, что, как отмечалось, логические и сдвиговые операции выполняются в процессоре компьютера, в языке программирования Паскаль имеется возможность дать команду на их выполнение в программе. Для этого в Паскале существуют операции с соответствующими именами: **and** и **shr**. Их можно применять к данным целого типа, и результат, который они возвращают, также является целым числом.

Так, результатом операции **and** над двумя операндами является десятичное число, которое соответствует двоичному представлению результата поразрядной операции **И** над исходными операндами. Например, в результате выполнения фрагмента программы:

```
x := 11 and 2;
writeln(x);
```

на экран будет выведено число 2. Обоснование:

	Двоичное представление			
11	1	0	1	1
2	0	0	1	0
11 and 2	0	0	1	0
Десятичное представление	2			

Результатом выполнения операции **shr**:

```
x := a shr b
```

будет целое десятичное число  $x$ , которое соответствует результату поразрядного сдвига вправо на  $b$  разрядов двоичного представления операнда  $a$ .

Если проанализировать результат выполнения оператора присваивания  $a := a \text{ shr } 1$  при четном и при нечетном  $a$ , то можно установить, что новое



значение величины  $a$  будет представлять собой целочисленное частное от деления “старого” значения на 2 (убедитесь в этом!). Это обстоятельство, а также отмеченный выше факт о выделении младшего разряда (последней цифры) двоичного числа при применении маски, равной 1, при логической операции `and` позволяют использовать операции `and` и `shr` для решения задачи перевода заданного натурального числа в двоичную систему счисления.

Разбор методики решения будем проводить с использованием школьного алгоритмического языка, для которого условно примем, что и в нем имеются нужные нам операции

1) **и** — аналог поразрядной логической операции `and`;

2) `СдвигВправоНа` — аналог сдвиговой операции `shr`.

Напомним методику перевода целых чисел из десятичной системы счисления в систему двоичную. Необходимо определять остаток от деления заданного числа и всех промежуточных целочисленных частных на 2 и делать это до тех пор, пока частное не станет равно нулю. Последовательность полученных остатков (начиная с последнего) и даст искомое двоичное представление заданного числа.

Рассмотрим несколько вариантов решения задачи.

В первом двоичные цифры запишем в массив с целыми элементами с именем *цифры*. Размер этого массива следует определить с учетом возможной длины двоичной записи числа. Используем также величину *кол\_цифр* — фактическое количество цифр в новой записи числа.

Приводя фрагмент программы решения задачи, обратим внимание на то, что в массив цифры записываются, начиная с последней цифры двоичной записи:

```
...
кол_цифр := 0
нц пока n > 0
  |Увеличиваем значение кол_цифр
  кол_цифр := кол_цифр + 1
  |Определяем последнюю двоичную цифру
  | (используя логическую операцию и)
  |и записываем ее в массив
  цифры[кол_цифр] := n и 1
  |"Отбрасываем" последнюю цифру,
  |определяя целочисленное частное
  |с помощью сдвиговой операции
  n := n СдвигВправоНа 1
кц
  |Выводим ответ
  |(цифры в обратном порядке)
вывод нс, "Запись числа n в двоичной системе счисления: "
нц для i от кол_цифр до 1 шаг -1
  вывод цифры[i]
кц
...
```

— где  $n$  — заданное натуральное десятичное число.

В приведенном варианте массив был нужен для запоминания цифр и их последующего вывода в обратном порядке. Вместо использования массива можно применить прием, называемый “рекурсией”. Создадим рекурсивную (вызывающую саму себя в качестве вспомогательной) процедуру `ВыделениеЦифрыИЕеВывод`, работающую так, как описано в комментариях:

```
алг ВыделениеЦифрыИЕеВывод (арг цел n)
нач цел цифра
  |Выделяем очередную цифру
  цифра := n и 1
  |(но пока не выводим)
  |Отбрасываем ее в числе n
  n := n СдвигВправоНа 1
если n > 0 |Только в этом случае
то
  |Вызываем эту же процедуру
  |с новым значением параметра n
  ВыделениеЦифрыИЕеВывод (n)
все
  |После окончания работы
  |вызванной процедуры
  |выводим цифру, выделенную
  |в текущей процедуре
вывод цифра
кон
```

Основная часть программы оформляется в виде:

```
алг Вывод_двоичного_представления_числа
нач цел n
вывод нс, "Введите натуральное число "
ввод n
вывод нс, "Запись этого числа в двоичной системе счисления: "
  ВыделениеЦифрыИЕеВывод (n)
кон
```

В третьем варианте решения будем считать, что требуется получить строковую (в терминах языка программирования Паскаль) величину  $n2$ , представляющую собой запись заданного числа в двоичной системе. Здесь массив можно не использовать, а формировать значение величины  $n2$ , добавляя очередную найденную цифру (ее имя — *цифра*) в начало уже полученного ранее значения величины  $n2$ :

```
...
n2 := "" |Начальное значение
нц пока n > 0
  |Определяем последнюю двоичную цифру
  цифра := n и 1
  |Добавляем ее строковое представление
  |в начало "старого" значения n2
  |Рассматриваем 2 варианта
если цифра = 1
то
  n2 := "1" + n2
иначе |цифра = 0
  n2 := "0" + n2
все
```

```

| "Отбрасываем" последнюю цифру
n := n СдвигВправоНа 1
кц
| Выводим ответ (значение величины n2)
вывод нс, "Запись числа n в двоичной
системе счисления: ", n2
...

```

**Задания для самостоятельной работы**

1. Разработайте две модификации последнего варианта фрагмента программы
  - 1) в которой условный оператор (команда если) не используется;
  - 2) в которой применяется рекурсия.
2. Разработайте программы решения задач:
  - 1) нахождения количества цифр в двоичной записи заданного натурального числа;

- 2) нахождения количества цифр 1 в двоичной записи заданного натурального числа;
- 3) нахождения количества цифр 0 и количества цифр 1 в двоичной записи заданного натурального числа.

Во всех случаях массив для хранения отдельных цифр не использовать.

Указанные фрагменты и/или программы оформите на языке Паскаль или на школьном алгоритмическом языке (учитывая сделанные в статье допущения). Результаты присылайте в редакцию.

**Литература**

1. Златопольский Д.М. Логические и сдвиговые операции. / "В мир информатики" № 170 ("Информатика" № 16/2011).

**ЗАДАЧНИК**

**Ответы, решения, разъяснения к заданиям, опубликованным ранее**

**1. Задача "Чернявый, Кудрявый и Лысый"**

Напомним, что необходимо было определить, кто из трех участников преступной группировки — Чернявый, Кудрявый и Лысый — совершил преступление, если из двух заявлений каждого:

Чернявый: "Я не делал этого. Это сделал Лысый";

Кудрявый: "Лысый невиновен. Это сделал Чернявый";

Лысый: "Я этого не делал. Кудрявый этого тоже не делал"

— суд установил, что один из них дважды солгал, другой дважды сказал правду, а третий — один раз солгал, а другой раз сказал правду.

*Решение*

Рассмотрим возможные варианты для двух правдивых заявлений.

1. Предположим, что их сделал Лысый. Из его слов следует (что ни он, ни Кудрявый не виноваты), что виноват Чернявый. Но тогда оба утверждения Кудрявого также верны, а этого быть не может. Следовательно, наше предположение неверно.

2. Допустим, что два раза сказал правду Кудрявый. В этом случае оба утверждения Лысого тоже верны, что также противоречит условию.

3. Из предположения, что правдивыми являются заявления Чернявого (то есть преступление совершил Лысый), следует, что Кудрявый дважды солгал, а Лысый в первый раз солгал, а во второй раз сказал правду. То есть такой вариант допустим.

Можно также найти решение, используя метод схем, разработанный О.Б. Богомоловой и описанный в "Информатике" № 8/2011.

*Ответ:* преступление совершил Лысый.

*Правильные ответы прислали:*

— Базылев Юрий, Республика Карелия, поселок Надвоицы, школа № 1, учитель **Богданова Л.М.**;

— Глазкова Екатерина, Республика Коми, г. Сыктывкар, МОУ "Лицей народной дипломатии", учитель **Гранаткина О.М.**;

— Григоренко Василий, Есипова Мария, Круглякова Мария и Яснова Дарья, средняя школа поселка Осиновка, Алтайский край, учитель **Евдокимова А.И.**;

— Михайлов Вячеслав, Свердловская обл., Красноуфимский р-н, Тавринская средняя школа, учитель **Ярцев В.А.**;

— Рябов Илья, Республика Коми, г. Сыктывкар, школа № 18, учитель **Гладких Ю.В.**;

— Яковлева Анна, средняя школа поселка Новопетровский Московской обл., учитель **Артамонова В.В.**

Возможны и другие способы решения, в том числе с использованием схем.

**2. Числовой ребус "Физики и лирики"**

Напомним, что необходимо было решить числовой ребус:

$$\text{ЛИРИК} = \frac{1}{2} \text{ФИЗИКА}$$

*Решение*

Для решения удобнее всего изобразить ребус не в виде, приведенном в условии, и не в виде умножения числа **ЛИРИК** на 2, а в виде примера на сложение:

+	Л	И	Р	И	К
	Л	И	Р	И	К
	Ф	И	З	И	К
				А	

Прежде всего видно, что **Ф** = 1, а **Л** = 5, 6, 7, 8 или 9.

Далее целесообразно исследовать возможные значения цифры Л и соответствующие значения других цифр.

Например, при Л = 5 (И = 0):

$$\begin{array}{rcccccc} & & 5 & 0 & P & 0 & K \\ + & & 5 & 0 & P & 0 & K \\ \hline \Phi & 0 & 3 & 0 & K & A & \end{array}$$

нет подходящих значений букв К и Р.

Соответствующий анализ показывает, что подходит только значение Л = 8, а все решение ребуса такое:

ЛИРИК = 87375, ФИЗИКА = 174750.

Ребус ФИЗИК =  $\frac{1}{2}$  ЛИРИКА решается аналогично.

Правильные ответы представили:

— Базылев Юрий, Республика Карелия, поселок Надвоицы, школа № 1, учитель **Богданова Л.М.**;

— Васинская Екатерина и Сафиуллин Ильдар, Республика Башкортостан, г. Стерлитамак, школа № 17, учитель **Орлова Е.В.**;

— Вахрушев Антон, Москва, гимназия № 1530, учитель **Козырева О.В.**;

— Глазкова Екатерина, Республика Коми, г. Сыктывкар, МОУ “Лицей народной дипломатии”, учитель **Гранаткина О.М.**

### 3. Задача “Сколько градусов было на улице?”

Напомним, что необходимо было определить температуру воздуха, при которой оба термометра, один из которых показывает температуру по шкале Цельсия, а другой — по шкале Фаренгейта, стоят на одинаковой отметке.

Решение

Зависимость между температурой по шкале Цельсия (С) и по шкале Фаренгейта (F) следующая:

$$C = (F - 32)/1,8$$

Так как оба термометра показывали одно и то же значение, то заменим, например, F на C:

$$C = (C - 32)/1,8,$$

откуда C = -40

Ответ: на улице было -40 градусов.

Правильные ответы прислали:

— Базылев Юрий, Республика Карелия, поселок Надвоицы, школа № 1, учитель **Богданова Л.М.**;

— Глазкова Екатерина, Республика Коми, г. Сыктывкар, МОУ “Лицей народной дипломатии”, учитель **Гранаткина О.М.**;

— Гусева Варвара и Казанкова Екатерина, Санкт-Петербург, г. Зеленогорск, лицей № 445, учитель **Зорина Е.М.**;

— Селин Влад, Амурская обл., средняя школа поселка Ерофей Павлович, учитель **Краснёнкова Л.А.**;

— Яковлева Анна, средняя школа поселка Новопетровский Московской обл., учитель **Артамонова В.В.**

Задания для самостоятельной работы, предложенные в статье “Расчеты в отладчике Debug. Новые команды”, выполнили:

— Стручков Илья, Москва, гимназия № 1530, учитель **Шамшев М.В.**;

— Султанов Тимур, Республика Башкортостан, г. Уфа, лицей № 60, учитель **Гильзер Н.В.**, а в статье “Странный муж”:

— Базылев Юрий, Республика Карелия, поселок Надвоицы, школа № 1, учитель **Богданова Л.М.**;

— Красиков Андрей, г. Пенза, школа № 512, учитель **Гаврилова М.И.**

Андрей, Илья, Тимур и Юрий будут награждены дипломами. Поздравляем!

## Задачи на взвешивания

Д.М. Златопольский, Д.Ю. Усенков

### Задача 1

Купец получил за товар 64 золотые монеты. Потом он узнал, что одна из этих монет фальшивая (более легкая). Как на рычажных весах с чашками и без гирь купец сможет определить, какая из монет фальшивая?

Самое простое решение — последовательно делить все количество монет пополам, на весах сравнивать их, затем брать более легкую кучку монет, делить ее пополам и т.д., пока при последнем взвешивании не останется сравнить между собой две монеты. Более легкая из них и есть фальшивая. Сказанное можно оформить в виде таблицы (синим цветом и жирным начертанием выделены кучки монет, содержащие фальшивую монету, а также сама эта монета):

№ взвешивания	Берем монеты	1-я чашка весов	2-я чашка весов	Допустим, что перевесила
1		<b>32</b>	32	2-я чашка
2	С 1-й чашки	16	<b>16</b>	1-я чашка
3	Со 2-й чашки	8	<b>8</b>	1-я чашка
4	Со 2-й чашки	<b>4</b>	4	2-я чашка
5	С 1-й чашки	2	<b>2</b>	1-я чашка
6	Со 2-й чашки	<b>1</b>	1	2-я чашка

Ответ: потребуется 6 взвешиваний.

**Задания для самостоятельной работы**

1. Нарисуйте блок-схему, описывающую все возможные варианты взвешиваний по рассмотренной методике при 16 монетах.

2. Определите, какое минимальное число взвешиваний потребуется для поиска фальшивой монеты среди  $2^n$  монет.

**Задача 2**

Есть 8 монет, из которых 7 настоящих, каждая из которых весит 10 г, и одна фальшивая весом 9 г. Необходимо, используя электронные весы, показывающие вес с точностью 1 г, найти фальшивую монету за минимальное число взвешиваний.

Прежде чем описывать решение, заметим, что фраза “наименьшее число взвешиваний” означает, что должны рассматриваться “худшие” варианты, а не варианты типа “а вдруг повезет”.

Например, если мы взвесим монету и ее вес окажется равным 9 г, это не означает, что задачу можно решить за одно взвешивание.

Решение во многом аналогично предыдущей задаче.

При первом взвешивании определим вес любых четырех монет, например, 1, 2, 3 и 4. Если весы покажут 40 г, то фальшивая монета находится среди оставшихся монет, если 39 г — среди взвешиваемых. Затем нужно узнать вес любых двух монет из соответствующей группы и выявить пару монет, среди которых есть фальшивая. Третье взвешивание покажет, какая монета из этой пары фальшивая.

**Задание для самостоятельной работы**

3. Нарисуйте блок-схему, описывающую все возможные варианты взвешиваний по рассмотренной методике решения задачи 2.

**Примечание.** Интересно, что для решения задачи можно применить... двоичную систему счисления!

Пронумеруем монеты: 1, 2, ..., 8 и представим номера монет в виде трехразрядных двоичных чисел:

Номер монеты	1	2	3	4	5	6	7	8
Условный номер	0	1	2	3	4	5	6	7
Двоичное представление	000	001	010	011	100	101	110	111

В первом взвешивании должны участвовать монеты, содержащие единицу в первом разряде, во втором — монеты с единицей во втором разряде, в третьем взвешивании — с единицей в третьем разряде. Результаты измерений будем записывать так: если общий вес монет равен 39 г, то во взвешиваемой группе содержится фальшивая монета — запишем 1 в соответствующий номеру измерения

разряд, в противном случае запишем 0. Полученное двоичное число будет однозначно определять номер фальшивой монеты.

**Пример.** Пусть фальшивая монета имеет номер 6. Ее условный номер:  $5_{10} = 101_2$ , значит, эта монета будет участвовать в первом и третьем взвешиваниях. При них весы покажут 39 г, а при втором взвешивании — 40 г, т.е. в результате получится двоичное число 101, соответствующее условному номеру фальшивой монеты.

**Обоснование**

Обсудим вопрос: “В каком случае при каком-то взвешивании весы покажут 39 г?”. Ответ — если среди взвешиваемых монет есть фальшивая. А 40 г? — Если ее там нет. Число взвешиваний равно количеству двоичных разрядов в записи номеров. Фальшивая монета будет участвовать в тех взвешиваниях, номера которых соответствуют номеру разряда в ее двоичной записи с единицами. И именно при этих взвешиваниях весы покажут 39 г (а мы запишем в соответствующем разряде 1). В остальных взвешиваниях фальшивая монета участвовать не будет, то есть при них весы покажут 40 г (а мы запишем 0 в соответствующем разряде). Следовательно, двоичное число, составленное по результатам всех взвешиваний, будет таким же, как двоичный номер искомой монеты.

**Задание для самостоятельной работы**

4. Подумайте, будет ли работать описанная методика, когда общее число монет равно:

- 1) 16;
- 2) 14.

**Задача 3**

Купец получил за товар 9 золотых монет. Потом он узнал, что одна из этих монет фальшивая, которая весит больше настоящих. За какое наименьшее число взвешиваний на рычажных весах с чашками и без гирь купец сможет определить, какая из монет фальшивая?

Если решать задачу так же, как задачу 1, то понадобится 3 взвешивания (убедитесь в этом самостоятельно). Но, оказывается, можно решить ее всего за два взвешивания!

Для этого можно разделить все монеты не на две, а на три кучки, после чего сравнить на весах две любые кучки. Если весы в равновесии, то фальшивая монета — в третьей кучке из трех монет, в противном случае — среди тех, которые на весах перевесили. Таким образом, в любом случае мы выделим кучку из трех монет, среди которых есть искомая. Сравнив на весах две любые монеты из этой кучки, мы этим, вторым, взвешиванием установим, какая из монет фальшивая.

**Задание для самостоятельной работы**

5. Решите задачу: “Купец получил за товар 64 золотые монеты. Потом он узнал, что одна из этих



монет фальшивая, которая весит больше настоящих. За какое наименьшее число взвешиваний на рычажных весах с чашками и без гирь купец сможет определить, какая из монет фальшивая?”

**Указания по выполнению.** Разделите монеты, среди которых есть фальшивая, на три части (в том числе неравные, если число монет не кратно 3). Например, первоначальную кучу из 64 монет можно разделить на три кучки из 21, 21 и 22 монет. В любом случае старайтесь, чтобы две из этих трех кучек были равными, а третья может от них отличаться.

Оказывается, что и в решении задачи 3 нам опять поможет система счисления, но не двоичная, а троичная!

Закодируем номера монет в троичной системе счисления:

<b>Номер монеты</b>	1	2	3	4	5	6	7	8	9
<b>Условный номер</b>	0	1	2	3	4	5	6	7	8
<b>Троичное представление</b>	00	01	02	10	11	12	20	21	22

Задачу можно решить за два взвешивания. Для первого взвешивания рассмотрим левый разряд троичных чисел — 1 в нем будет означать, что монету нужно положить на левую чашку весов, 2 — на правую чашку, а 0 — что монета не участвует во взвешивании. Затем аналогично для второго разряда.

Запишем результаты каждого  $i$ -го взвешивания ( $i = 1, 2$ ) в соответствующий разряд по правилу: если перевесила левая чашка — 1, если правая — 2, если весы уравновешены — 0. Полученное троичное число однозначно определяет номер фальшивой монеты.

Проверьте сказанное самостоятельно на примере какой-то монеты. Почему так получается?

### Задание для самостоятельной работы

6. Решите задачу 3 для случая, когда монет — 27.

Ответы на задания и вопросы для самостоятельной работы присылайте в редакцию (можно выполнять не все задания).

### Четыре девушки

Маша, Люда, Женя и Катя умеют играть на различных инструментах (виолончели, рояле, гитаре и скрипке), но каждая только на одном. Они же владеют различными иностранными языками (английским, французским, немецким и испанским), но каждая — только одним. Известно, что:

- 1) девушка, которая играет на гитаре, говорит по-испански;
- 2) Люда не играет ни на скрипке, ни на виолончели и не знает английского языка;

3) Маша не играет ни на скрипке, ни на виолончели и не знает английского языка;

4) Женя знает французский язык, но не играет на скрипке.

Кто на каком инструменте играет и какой иностранный язык знает? Сколько решений имеет задача?

### Потерянный рубль

Когда-то три путешественника забрели на постоялый двор, хорошо поели, заплатили хозяйке 30 рублей и пошли дальше. Через некоторое время после их ухода хозяйка обнаружила, что взяла с путешественников лишнее. Будучи женщиной честной, она оставила себе 25 рублей, а 5 рублей дала мальчику и наказала ему догнать путешественников и отдать им эти деньги. Мальчик бегал быстро и скоро догнал путешественников. Но как им разделить 5 рублей на троих? Они оставили каждый по 1 рублю, а 2 рубля оставили мальчику за его “быстроногость”.

Таким образом, путешественники сначала заплатили за обед по 10 рублей, но по 1 рублю получили обратно, следовательно, они заплатили:  $9 \times 3 = 27$  рублей. Да 2 рубля остались у мальчика:  $27 + 2 = 29$  рублей. Но вначале-то было 30 рублей! Куда делся 1 рубль?

### Охрана бастиона

Вдоль стен бастиона квадратной формы его комендант разместил 16 часовых по 5 человек с каждой стороны — так, как показано на рисунке:

1	3	1
3		3
1	3	1

Через некоторое время пришел полковник, выразил недовольство расстановкой часовых и переставил их так, что с каждой стороны оказалось по 6 человек.

Однако после этого появился генерал. Он также остался недоволен и переставил часовых таким образом, чтобы с каждой стороны их оказалось по 7.

Как расположил часовых полковник?

Как их расставил генерал?

Общее число часовых остается одним и тем же.

### В швейцарской общине

Одна швейцарская община насчитывает 50 членов. Родной язык всех 50 членов общины — немецкий, но 20 из них говорят еще и по-итальянски, 35 из них владеют французским и 10 не знают ни итальянского, ни французского.

Сколько членов общины говорят и по-французски, и по-итальянски?

### Две бочки

Две бочки, по 10 галлонов каждая, снабжены этикетками А и Б. Бочка А содержала больше воды, чем бочка Б.

Сначала из бочки А в бочку Б перелили столько воды, сколько там уже было. После этого из бочки Б в бочку А перелили столько жидкости, сколько в последней осталось. Наконец, из бочки А в бочку Б перелили столько, сколько теперь осталось в бочке Б. После этого в обеих бочках стало по 48 пинт (в галлоне чуть меньше 10 пинт) воды. А сколько ее было в каждой из бочек вначале?

### Петя и Митя

На ступенях дома сидят рядышком два мальчика: Петя и Митя.

— Меня зовут Митя, — сказал мальчик с черными волосами.

— Меня зовут Петя, — сказал мальчик с белыми волосами.

Можно ли определить, как зовут каждого из мальчиков, если по крайней мере один из них говорит неправду?

## ТВОРЧЕСТВО НАШИХ ЧИТАТЕЛЕЙ

### Ребусы, посвященные Году космонавтики. Часть 3

Мы продолжаем публиковать ребусы, которые разработали Анатолий и Ульяна Тимофеевы, ученики Именевской основной школы, Красноармейский р-н Чувашской Республики (учитель **Тимофеева И.А.**).

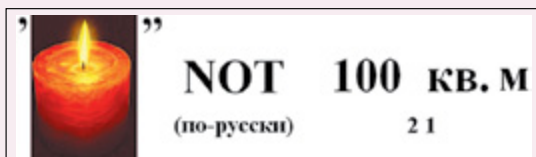
Ребус № 1



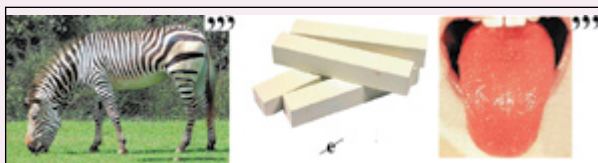
Ребус № 2



Ребус № 3



Ребус № 4



Ребус № 5



Ребус № 6



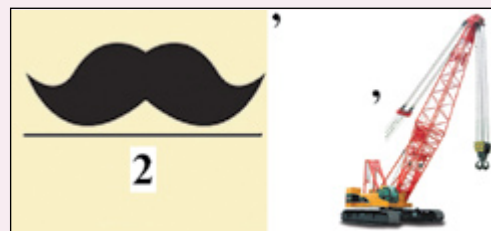
Ребус № 7



Ребус № 8



Ребус № 9



Ответы присылайте в редакцию (можно решать не все ребусы).

## Создаем сами двоичный сумматор

► В статьях [1–2] были описаны так называемые “логические вентили” — технические устройства, с помощью которых можно реализовать логические операции конъюнкции (вентиль “И”) и дизъюнкции (вентиль “ИЛИ”), а также вентили, реализующие сразу две логические операции: “НЕ–ИЛИ” и “НЕ–И”. Для них были введены стандартные обозначения:

1) вентиль “И”:

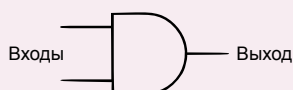


Рис. 1

2) вентиль “ИЛИ”:



Рис. 2

3) вентиль “НЕ–ИЛИ”:

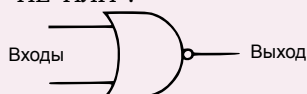


Рис. 3

4) вентиль “НЕ–И”:



Рис. 4

Зависимость выходного сигнала каждого из четырех вентилях от сигналов на входе иллюстрируется следующей таблицей:

<b>И</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	0
<b>1</b>	0	1
<b>ИЛИ</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	1
<b>ИЛИ–НЕ</b>	<b>0</b>	<b>1</b>
<b>0</b>	1	0
<b>1</b>	0	0
<b>И–НЕ</b>	<b>0</b>	<b>1</b>
<b>0</b>	1	1
<b>1</b>	1	0

Используя вентили, можно сконструировать двоичный сумматор — устройство для сложения двух двоичных чисел. Сложение — основное арифметическое действие, и потому, если мы хотим построить компьютер (а в этом и заключается наша цель), надо прежде всего разобраться, как создать сумматор. Когда мы сделаем это, то вы поймете, что сложение — это практически *единственное действие*, которое выполняют компьютеры. Разработав сум-

мирующее устройство, мы окажемся на пути к созданию прибора, в котором сложение используется для вычитания, умножения, деления, расчета зарплаты, вычисления траектории полета на Марс, игры в шахматы и т.д. и т.п.

Сумматор, который мы создадим в этой статье, будет громоздким, неуклюжим, медленным и шумным ☹, по крайней мере в сравнении с современными калькуляторами и компьютерами. Самое интересное, что мы соберем его из простейших электронных устройств, описанных ранее в [1–3]: переключателей, лампочек, проводов, источника питания и реле, объединенных в различные логические вентили. В этом сумматоре не будет ни единой детали, не изобретенной минимум 130 лет назад. Что особенно приятно, мы не будем захламлять всей этой электроникой вашу квартиру — мы построим сумматор на бумаге и в воображении.

Наш сумматор работает только с двоичными числами и лишен некоторых современных прелестей. В нем, например, нельзя применять для ввода складываемых чисел клавиатуру. Вместо нее мы будем использовать вереницу переключателей. Роль экрана монитора для отображения результата будет играть ряд лампочек.

И все же эта машина будет находить сумму двух чисел, причем почти так же, как это происходит в современном компьютере.

Сложение двоичных чисел очень похоже на сложение десятичных. Чтобы сложить два десятичных числа, например 245 и 673, вы разбиваете задачу на несколько простых шагов. На каждом шаге требуется сложить две десятичных цифры. В данном примере суммирование начинается с цифр 5 и 3. Сделать это гораздо проще, если вы на каком-то жизненном этапе выучили таблицу сложения.

Важное различие между десятичными и двоичными числами в том, что таблица сложения для двоичных чисел, как вы, очевидно, знаете, значительно проще аналогичной таблицы для десятичных чисел:

+	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	10

Можно переписать эту таблицу с дополнительными нулями, чтобы каждый результат был двухбитовой величиной:

+	<b>0</b>	<b>1</b>
<b>0</b>	00	01
<b>1</b>	01	10

Результат сложения пары двоичных чисел, представленный подобным образом, содержит 2 бита: *разряд суммы* и *разряд переноса*. Теперь можно разделить таблицу сложения двоичных чисел на две, первая из которых предназначена для разряда суммы:

+ сумма	0	1
0	0	1
1	1	0

а вторая — для разряда переноса:

+ перенос	0	1
0	0	0
1	0	1

Двоичное сложение удобно рассматривать именно с такой точки зрения, так как в нашем сумматоре значения суммы и переноса будут вычисляться раздельно. Построение сумматора состоит в проектировании схемы, которая выполняла бы эти операции. Работа в двоичной системе здорово упрощает нашу задачу, так как все части схемы — переключатели, лампочки и провода — могут символизировать двоичные разряды.

Как и при десятичном сложении, мы будем складывать цифры последовательно столбец за столбцом, начиная с крайнего правого:

+	0	1	1	0	0	1	0	1
	1	0	1	1	0	1	1	0
	1	0	0	0	1	1	0	1

Заметьте, что при сложении цифр в третьем столбце справа 1 переносится в следующую колонку. То же самое происходит в шестом, седьмом и восьмом столбцах справа.

Какого размера двоичные числа мы собираемся складывать? Конечно, воображаемый сумматор в принципе способен складывать числа любой разрядности. Но давайте ограничимся разумной длиной и будем складывать числа не длиннее 8 бит в диапазоне от 00000000 до 11111111, или в десятичном выражении от 0 до 255. Сумма двух 8-битовых чисел может достигать значения 11111110, или 510.

Пульт управления нашим сумматором может выглядеть примерно так.



Рис. 5

На пульте размещены два ряда переключателей по 8 в каждом. Это устройство ввода, которое мы будем использовать для ввода 8-разрядных чисел. В нем нижнее положение переключателя (выключен) соответствует вводу 0, а верхнее (включен) — вводу 1. Устройство вывода в виде ряда из 9 лампочек находится в нижней части пульта. Эти лампочки будут показывать ответ. Горящая лампочка

соответствует 1, выключенная — 0. Нам нужны 9 лампочек, так как сумма двух 8-разрядных чисел может быть 9-разрядным числом.

В остальном сумматор состоит из логических вентиляей, соединенных разными способами. Переключатели будут вызывать срабатывание реле в логических вентиляях, которые подадут питание на нужные лампочки. Например, чтобы сложить 01100101 и 10110110, мы включим переключатели, как показано на рис. 6.



Рис. 6

Горящие лампочки показывают ответ — 100011011 (пока примем его на веру — ведь мы еще ничего не собирали!).

Чуть выше говорилось, что в сумматоре мы будем использовать реле. Создаваемый нами 8-разрядный сумматор будет состоять из 144 реле, по 18 для каждой из 8 пар битов, которые мы будем складывать. Если бы мы привели всю схему соединений всех этих реле, вы бы определенно ничего не поняли. И не только вы. Нет абсолютно никакой надежды, что кто-то сумеет разобраться в хитросплетениях схемы, состоящей из 144 реле. Мы поступим по-другому — применим модульный подход к решению этой проблемы (то есть будем решать ее не всю сразу, а по частям), а помогут нам логические вентили.

Вполне вероятно, что вы уже увидели связь между логическими вентилями, описанными в предыдущей статье, и двоичным сложением. Для этого достаточно взглянуть на таблицу переноса разряда при сложении двух однобитовых чисел (см. выше). Ведь она совершенно идентична таблице с результатами работы вентиля “И” (см. выше).

Итак, вентиль “И” можно использовать для расчета переноса разряда при сложении двух двоичных цифр.

Внимание — мы подбираемся к сути! Очередной шаг состоит, видимо, в проверке, ведут ли себя какие-нибудь реле следующим образом:

+ сумма	0	1
0	0	1
1	1	0

Это вторая часть задачи сложения пары двоичных цифр. Разряд суммы получается не столь прямолинейно, как разряд переноса, но в дальнейшем мы решим и эту проблему.



Для начала сообразим, что вентиль “ИЛИ” дает почти то, что нужно, не считая ячейки в правом нижнем углу таблицы:

<b>ИЛИ</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	1

Результат действия вентиля “И-НЕ” тоже близок к нашим потребностям, кроме верхней левой ячейки таблицы:

<b>И-НЕ</b>	<b>0</b>	<b>1</b>
<b>0</b>	1	1
<b>1</b>	1	0

Подключим к одним и тем же входам вентиль “ИЛИ” и вентиль “И-НЕ”:

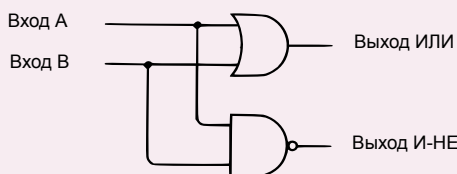


Рис. 7

В следующей таблице возможные сигналы на выходах соединенных вентилях “ИЛИ” и “И-НЕ” сравниваются с тем, что нужно для сумматора:

Вход А	Вход В	Выход ИЛИ	Выход И-НЕ	Что нужно
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

Обратите внимание: единица на выходе нам нужна, только если единице равны выход вентиля “ИЛИ” и выход вентиля “И-НЕ”. Это наводит на мысль, что два этих выхода могут быть входами вентиля “И”:

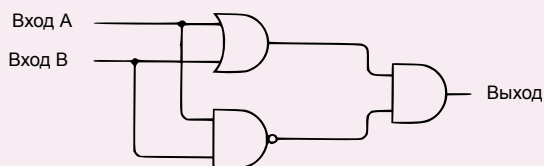


Рис. 8

Отметим, что во всей схеме по-прежнему всего два входа и один выход. Два входа принадлежат как вентилю “ИЛИ”, так и вентилю “И-НЕ”. Два выхода от вентиля “ИЛИ” и от вентиля “И-НЕ” идут на вход вентиля “И”, который и выдает необходимый результат:

Вход А	Вход В	Выход ИЛИ	Выход И-НЕ	Выход И
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

У созданной нами логической схемы есть собственное имя — *исключающее ИЛИ* (Искл-ИЛИ, по

англ. — Exclusive OR, XOR). Исключающей схема названа потому, что ее выход равен 1, если единичный сигнал есть на входе А или на входе В, но не на обоих. Чтобы не рисовать каждый раз вентили “ИЛИ”, “И-НЕ” и “И”, мы будем использовать для схемы “Искл-ИЛИ” такое обозначение:



Рис. 9

Оно очень похоже на символ вентиля “ИЛИ”, за исключением того, что на нем со стороны входа нарисована еще одна кривая линия. Поведение вентиля “Искл-ИЛИ” проиллюстрировано в таблице:

<b>Искл-ИЛИ</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	0

Вентиль “Искл-ИЛИ” — последний, который понадобится для создания сумматора.

Подведем итог. Сложение двух двоичных цифр приводит к появлению бита суммы и бита переноса:

+ сумма	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	0

+ перенос	<b>0</b>	<b>1</b>
<b>0</b>	0	0
<b>1</b>	0	1

Разряд суммы двух двоичных чисел задается выходом вентиля “Искл-ИЛИ”, а разряд переноса — выходом вентиля “И”.

Для сложения двух двоичных цифр А и В мы можем объединить два этих вентиля (см. рис. 10).

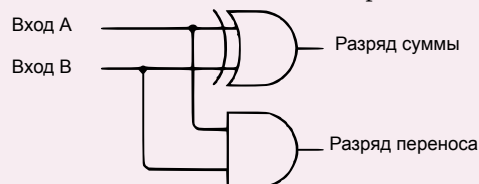


Рис. 10

Чтобы не рисовать многократно вентили “И” и “Искл-ИЛИ”, заменим эту схему обозначением

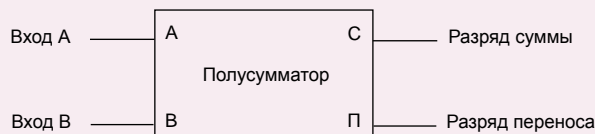


Рис. 11

*Полусумматором* эта схема названа неспроста. Конечно, он складывает две двоичные цифры и выдает разряд суммы и разряд переноса. Но подавляющее большинство двоичных чисел записывается несколькими битами. Наш полусумматор не делает одной важной вещи: не прибавляет к сумме возможный разряд переноса от предыдущего суммирования. Допустим, мы складываем два двоичных числа, как показано ниже:

+	1	1	1	1
	1	1	1	1
	1	1	1	0

Полусумматор можно использовать только для сложения крайнего правого столбца: 1 плюс 1 равно 0, и 1 идет в перенос. Для второго столбца справа из-за переноса нам на самом деле нужно сложить *три* двоичных цифры. То же относится и к остальным столбцам. Каждое последующее сложение двух двоичных цифр может включать перенос из предыдущего столбца.

Чтобы сложить три двоичных цифры, нам нужны два полусумматора и вентиль “ИЛИ”:

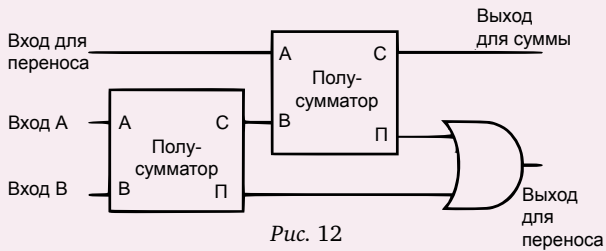


Рис. 12

Чтобы разобраться в работе этой схемы, начнем со входов А и В в первом полусумматоре (том, что слева). Результатом его работы являются сумма и перенос. Эту сумму нужно прибавить к переносу из предыдущего столбца, поэтому оба числа подаются на вход второго полусумматора. Сумма из второго полусумматора будет окончательной суммой. Два переноса из полусумматоров попадают на входы вентилia “ИЛИ”. В этом месте, конечно, можно использовать еще один полусумматор, который выполнит необходимые действия. Но, проанализировав возможные варианты, вы убедитесь, что выходы для переноса из двух полусумматоров никогда не будут одновременно равны 1. Для их сложения вполне достаточно вентилia “ИЛИ”, который действует аналогично вентилю “Искл-ИЛИ”, если его входы не равны 1 одновременно.

Теперь назовем эту схему *полным сумматором* и введем для нее обозначение:

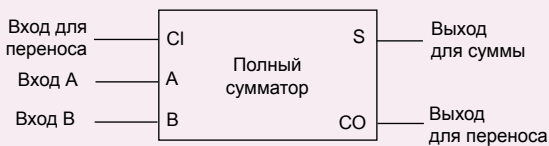


Рис. 13

В следующей таблице приводятся все возможные комбинации входных и выходных сигналов полного сумматора:

Вход А	Вход В	Вход для переноса	Выход для суммы	Выход для переноса
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Чуть раньше уже говорилось, что для всего сумматора нам понадобится 144 реле. Теперь можно пояснить это число. В каждом вентиле “И”, “ИЛИ” и “И-НЕ” по два реле. Таким образом, вентиль “Искл-ИЛИ” построен из шести реле. Полусумматор состоит из вентилia “Искл-ИЛИ” и вентилia “И”, то есть из восьми реле. Каждый полный сумматор состоит из двух полусумматоров и одного вентилia “ИЛИ”, итого — 18 реле. Для нашего сумматора нам нужны восемь полных сумматоров. Всего получается 144 реле.

Прежде чем идти дальше, обязательно вспомните пульт управления с переключателями и лампочками (рис. 5).

Теперь можем приступить к соединению полных сумматоров с переключателями и лампочками.

Начнем с того, что соединим два крайних правых переключателя, крайнюю правую лампочку и полный сумматор:

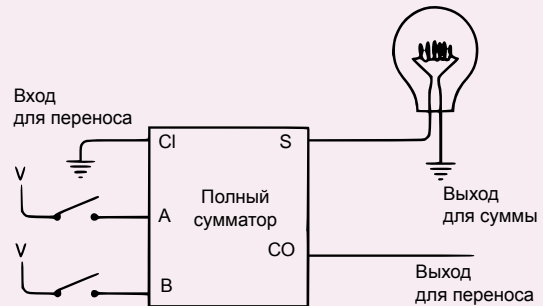


Рис. 14

При сложении двух двоичных чисел первый столбец цифр, которые мы складываем, не похож на другие тем, что в отличие от последующих столбцов в нем не может быть разряда переноса из предыдущего столбца. Поэтому вход для переноса у первого полного сумматора соединяется с землей. Это значит, что его значение всегда 0. Конечно, разряд переноса может появиться *в результате* сложения первой пары двоичных цифр.

Со следующей парой переключателей и следующей лампочкой полный сумматор соединяется так:

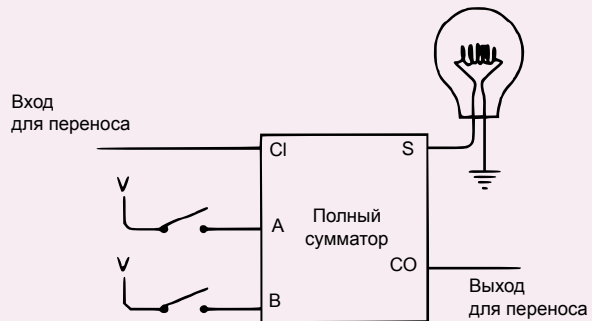


Рис. 15

Выход для переноса из первого полного сумматора становится входом для переноса во второй полный сумматор. Схемы для сложения всех последующих столбцов аналогичны. Разряд переноса из одного столбца подается на вход для переноса следующего столбца.

И, наконец, восьмая, и последняя, пара переключателей соединяется с последним полным сумматором так:

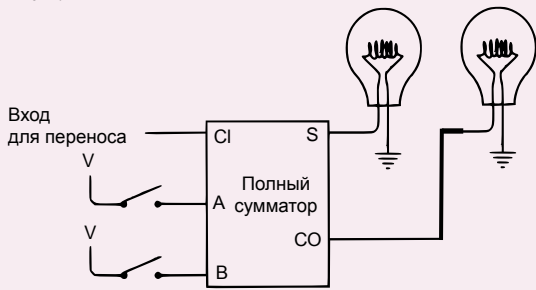


Рис. 16

Последний выход для переноса соединяется с девятой лампочкой.

Готово!

Теперь изобразим все восемь полных сумматоров сразу, подключив все выходы для переноса (CO) к последующим входам для переноса (CI) и обозначив сумму буквой “S” (см. рис. 17).

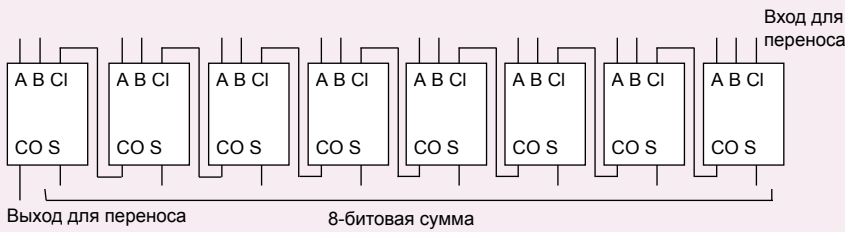


Рис. 17

Наконец, введем единое обозначение для достаточно громоздкого 8-битового сумматора, показанного на рис. 17. На рис. 18 его входы обозначены буквами от  $A_0$  до  $A_7$  и от  $B_0$  до  $B_7$ , а выходы — буквами от  $S_0$  до  $S_7$ .

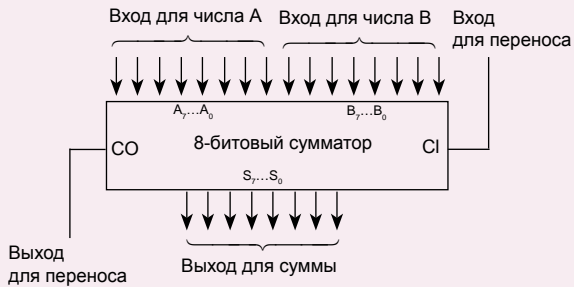


Рис. 18

Отдельные биты многобитового числа также будем обозначать буквами с нижними индексами. Биты  $A_0$ ,  $B_0$  и  $S_0$  называют младшими, а  $A_7$ ,  $B_7$  и  $S_7$  — старшими. Пример соответствия буквенных обозначений разрядам двоичного числа 01101001:

$A_7$	$A_6$	$A_5$	$A_4$	$A_3$	$A_2$	$A_1$	$A_0$
0	1	1	0	1	0	0	1

8-битовый сумматор можно изобразить и так, как показано на рис. 19.

Восьмерки внутри стрелок означают, что каждая из них соответствует группе из восьми отдельных сигналов. Восьмиразрядность числа подчеркивается также нумерацией символов от  $A_7$  до  $A_0$ , от  $B_7$  до  $B_0$  и от  $S_7$  до  $S_0$ .

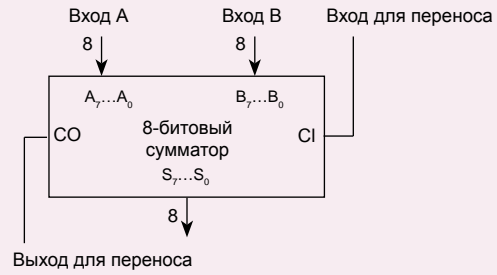


Рис. 19

Создав один 8-битовый сумматор, вы легко построите и второй. Их можно расположить каскадом для сложения 16-битовых чисел (рис. 20).

Выход для переноса сумматора справа соединяется с входом для переноса сумматора слева. Сумматор слева получает на входе 8 старших цифр двух складываемых чисел и выдает на выходе 8 старших разрядов результата.

Итак, мы создали прибор, на котором можно складывать любые двоичные числа (задавая их с помощью переключателей) и получать результат в виде двоичных сигналов (лампочек).

Теперь вы можете спросить: “Неужели в компьютерах сложение действительно осуществляется именно так?”

В общем, да, но не совсем.

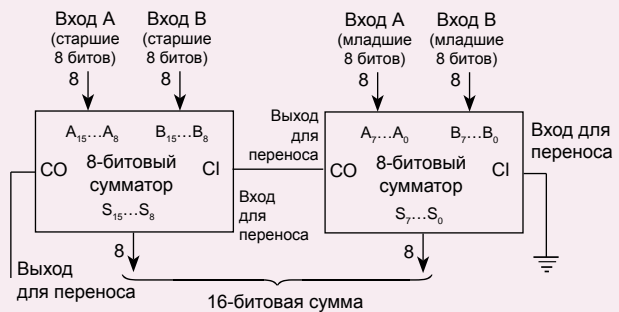


Рис. 20

Во-первых, сумматор может работать быстрее, чем тот, с которым мы разбирались. Взгляните, как действует цепь: выход переноса от младшей пары цифр требуется для сложения со следующей парой цифр, выход переноса второй пары цифр требуется для третьей пары цифр и т.д. Скорость суммирующей схемы равна произведению числа битов на скорость действия одного полного сумматора. Такой перенос называется *сквозным*. В быстродействующих сумматорах с помощью дополнительной цепи реализован *ускоренный перенос*.

Во-вторых (и это более важно), в компьютерах больше не используют реле! Лишь первые цифровые компьютеры, созданные в начале 1930-х годов, были основаны на реле и чуть позже на вакуумных лампах. Сегодняшние компьютеры собирают из транзисторов. В компьютере транзисторы выполняют те же функции, что и реле. Разница в том, что

они гораздо быстрее, компактнее, экономичнее и дешевле. Для создания 8-битового сумматора по-прежнему потребуется 144 транзистора (или больше, если заменить сквозной перенос ускоренным), но он будет иметь микроскопические размеры...

### Вопросы для самостоятельного анализа

В представленной на рис. 17 схеме один из восьми полных сумматоров можно заменить на полусумматор. Какой именно? Можно ли это сделать на 16-битовой схеме (рис. 20)?

За участие в конкурсе № 85 (его задание было связано с использованием средств электронной таблицы Microsoft Excel для моделирования решения задачи “Кто выше”, опубликованной в прошлом учебном году) дипломами будут награждены:  
 — Базылев Юрий, Республика Карелия, поселок Надвоицы, школа № 1, учитель **Богданова Л.М.**;  
 — Любимов Антон, г. Пенза, школа № 512, учитель **Гаврилова М.И.**;  
 — Яновский Виталий, Москва, гимназия № 1530, учитель **Козырева О.В.**

## “ЛОМАЕМ” ГОЛОВУ

### Числовой ребус на космическую тему

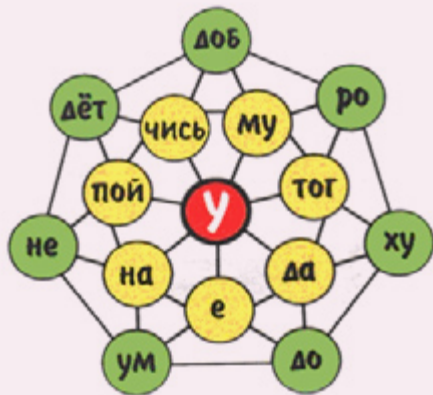
Продолжая космическую тему (см. рубрику “Творчество наших читателей”), предлагаем решить следующий числовой ребус:

$$\begin{array}{r}
 \phantom{+} \quad S \quad A \quad T \quad U \quad R \quad N \\
 + \quad U \quad R \quad A \quad N \quad U \quad S \\
 \hline
 P \quad L \quad A \quad N \quad E \quad T \quad S
 \end{array}$$

Как обычно, одинаковыми буквами зашифрованы одинаковые цифры, разными буквами — разные цифры.

### Старая русская пословица

Обойдя кружочки в определенном порядке и только по одному разу, прочитайте старинную русскую пословицу.



Алгоритм решения задачи оформите в виде:

1. НЕ — ПОЙ.
2. ПОЙ — У.
3. ...

### Занимательные числа

Назовите два числа, у которых количество цифр равно количеству букв, составляющих название этого числа.

### Литература

1. Выбрать кошку помогают... реле. / “В мир информатики” № 169 (“Информатика” № 15/2011).
2. Еще два вентиля. / “В мир информатики” № 170 (“Информатика” № 16/2011).
3. От кошек — через переключатели — к компьютерам. / “В мир информатики” № 168 (“Информатика” № 14/2011).
4. Петцольд Ч. Код. М.: Издательско-торговый дом “Русская редакция”, 2001.

### Числовой ребус без букв и цифр

В данной головоломке не известно ни одной цифры, однако обратите внимание на запятую в частном. Благодаря тому, что после запятой стоят четыре цифры, ребус решается неожиданно легко.

$$\begin{array}{r}
 * * * * * * \quad * * * \\
 * * * \quad * * * * *, * * * * \\
 * * * \\
 * * * \\
 * * * \\
 * * * \\
 * * * \\
 * * * \\
 * * * \\
 * * * * \\
 * * * * \\
 0
 \end{array}$$

### Продолжить последовательность

Определите, как должна выглядеть последняя табличка, чтобы закон изменения положения символов во всех табличках сохранился:

	○	x
□		

x		
	○	
□		

		x
	○	
□		

x		
	○	
□		

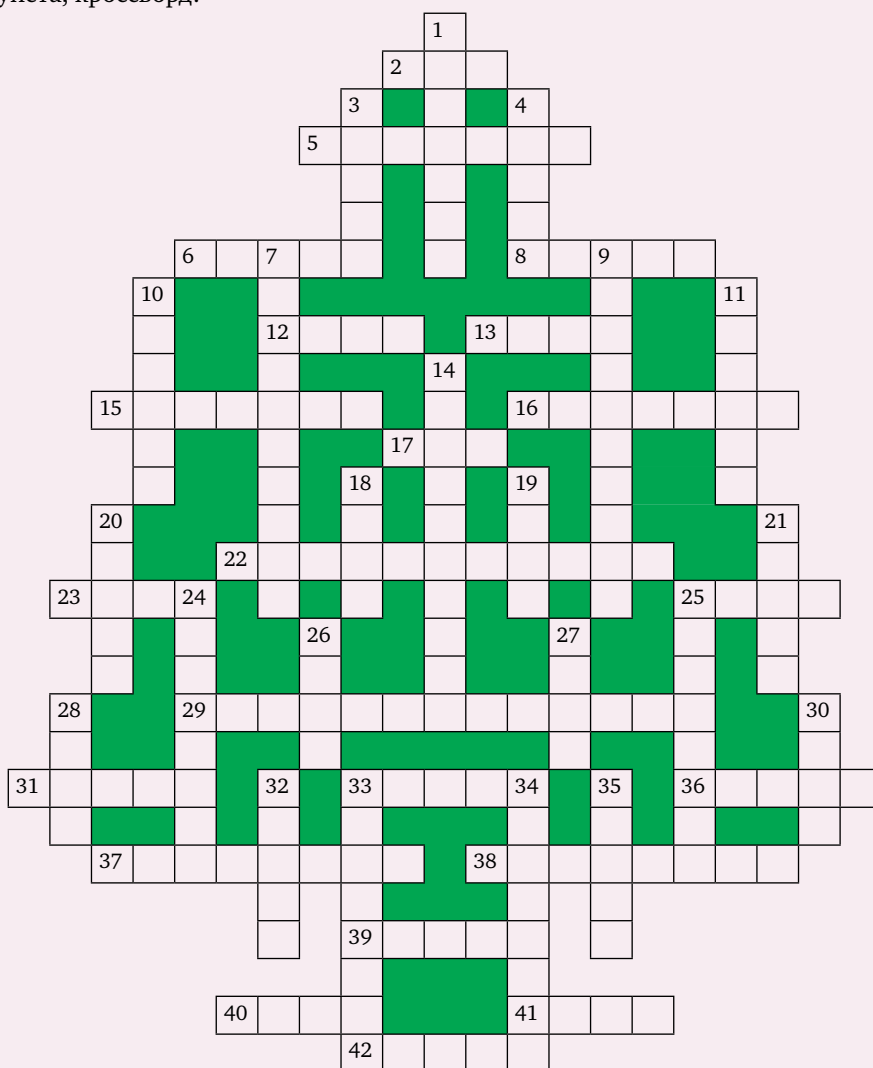
		x
	○	
□		

□		



## Новогодний кроссворд

Решите, пожалуйста, кроссворд:



По горизонтали

2. Единица измерения информации.
5. Устройство для вывода информации в компьютере.
6. Программа, обладающая способностью к самовоспроизведению.
8. Порядковый номер байта оперативной памяти.
12. Символ в формуле.
13. Знак арифметической операции.
15. Пользователь телефонной линии.
16. Электронная схема, применяемая в регистрах компьютера для запоминания одного бита информации.
17. Величина изменения значения переменной цикла.
22. Программное изделие в виде, поставляемом изготовителем.
23. Стадия решения задачи, элемент алгоритма.
25. Система знаков для секретного письма.
29. Транслятор, который “переводит” каждый оператор программы в машинные команды и немедленно исполняет их.
31. Один из первых языков программирования высокого уровня.
33. Металл, используемый при пайке во время ремонта компьютера.
36. Последовательность символов, предназначенная для чтения человеком.
37. Глобальная компьютерная сеть.
38. Вид связи.
39. Буква греческого алфавита.
40. Часть экрана, занимаемая приложением или документом Windows.
41. Популярный вид компьютерных программ.
42. Прибор для радиосвязи.

*По вертикали*

1. Распечатка текста программы.
3. Одно из важнейших понятий объектно-ориентированного программирования, а также группа учеников в школе.
4. В языках программирования — идентификатор, позволяющий именовать некоторый оператор программы.
7. Одна из характеристик работы монитора.
9. Часть имени файла.
10. Название клавиши.
11. Один из двух режимов ввода символов в текстовых редакторах.
14. Электронный прибор на полупроводниковом кристалле — элемент микросхемы.
18. Цифра десятичной системы счисления.
19. Одна из составляющих понятия *мультимедиа*.
20. Непрерывная последовательность данных.
21. Знак, обозначающий число.
24. Человек, знающий несколько языков (не обязательно языков программирования).
25. Устройство для кодирования информации.
26. Специальная область памяти персональных компьютеров, работающая по принципу “Последним пришел — первым вышел”.
27. Конечное число точек на плоскости, соединенных отрезками кривых линий.
28. Основоположник математической логики.
30. Разновидность носителя информации.
32. Элемент базы данных, а также внешнее очертание, наружный вид предмета.
33. Элементарная единица программы.
34. Действие, производимое над объектами языка программирования.
35. Буква греческого алфавита.

**Ответы***По горизонтали*

2. Бит. 5. Плоттер. 6. Вирус. 8. Адрес. 12. Знак. 13. Плюс. 15. Абонент. 16. Триггер. 17. Шаг. 22. Дистрибутив.
23. Этап. 25. Шифр. 29. Интерпретатор. 31. Алгол. 33. Олово. 36. Текст. 37. Интернет. 38. Телеграф.
39. Альфа. 40. Окно. 41. Игра. 42. Рация.

*По вертикали*

1. Листинг. 3. Класс. 4. Метка. 7. Разрешение. 9. Расширение. 10. “Пробел”. 11. “Замена”. 14. Транзистор.
18. Пять. 19. Звук. 20. Поток. 21. Цифра. 24. Полиглот. 25. Шифратор. 26. Стекло. 27. Граф. 28. Буль. 30. Диск.
32. Форма. 33. Оператор. 34. Операция. 35. Омега.

**ВНИМАНИЕ! КОНКУРС**

## Конкурс № 91 “Удалить и заменить букву”

В следующем выпуске будут представлены головоломки, в которых требуется заменить, добавить или удалить букву в заданных словах. В качестве задания конкурса № 91 предлагаем решить более сложную задачу — по заданным словам, удалив в них одну букву и заменив другую, получить термин (или фамилию ученого), связанный с информатикой и ИКТ. Причем первая буква искомого термина неизвестна.

- |            |            |
|------------|------------|
| 1. Факел.  | 8. Порка.  |
| 2. Капкан. | 9. Сукно.  |
| 3. Имам.   | 10. Кадр.  |
| 4. Модель. | 11. Абрек. |
| 5. Бойль.  | 12. Зона.  |
| 6. Полис.  | 13. Шифон. |
| 7. Минута. | 14. Ярлык. |

- |                 |               |
|-----------------|---------------|
| 15. Стела.      | 23. Метраж.   |
| 16. Логин.      | 24. Паром.    |
| 17. Ложка.      | 25. Домино.   |
| 18. Тёмность.   | 26. Посев.    |
| 19. Логика.     | 27. Тина.     |
| 20. Константин. | 28. Завод.    |
| 21. Матросы.    | 29. Поставка. |
| 22. Смена.      | 30. Грант.    |

Приведите также комментарии к найденным словам.

Ответы отправьте в редакцию до 10 января по адресу: 121165, Москва, ул. Киевская, д. 24, “Первое сентября”, “Информатика” или по электронной почте: [vmi@1september.ru](mailto:vmi@1september.ru). Пожалуйста, четко укажите в ответе свои фамилию и имя, населенный пункт, номер и адрес школы, фамилию, имя и отчество учителя информатики.


В ответах сохраните нумерацию слов в задании (для найденных слов поставьте прочерк).

Конкурс будет проводиться в два тура.





**ДИСТАНЦИОННЫЕ КУРСЫ ПОВЫШЕНИЯ КВАЛИФИКАЦИИ  
ВНЕ ЗАВИСИМОСТИ ОТ МЕСТА ПРОЖИВАНИЯ  
(обучение с 1 января по 30 сентября 2012 года)**

**КОД  ПРОФИЛЬНЫЕ КУРСЫ**

- 07-001 *И.Г. Семакин. Информационные системы в базовом и профильном курсах информатики*  
07-008 *А.Г. Гейн. Математические основы информатики*  
 07-009 *С.Л. Островский. Основы web-программирования для школьного «сайтостроительства»*  
07-010 *А.Г. Кушниренко, А.Г. Леонов. Методика преподавания основ алгоритмизации на базе системы «Кумир»*

**КОД  ОБЩЕПЕДАГОГИЧЕСКИЕ КУРСЫ**

- 21-001 *С.С. Степанов. Теория и практика педагогического общения*  
21-002 *Н.У. Заиченко. Методы профилактики и разрешения конфликтных ситуаций в образовательной среде*  
21-003 *С.Н. Чистякова, Н.Ф. Родичев. Образовательно-профессиональное самоопределение школьников в предпрофильной подготовке и профильном обучении*  
21-004 *М.Ю. Чибисова. Психолого-педагогическая подготовка школьников к сдаче выпускных экзаменов в традиционной форме и в форме ЕГЭ*  
 21-005 *М.А. Ступницкая. Новые педагогические технологии: организация и содержание проектной деятельности учащихся*  
 21-007 *А.Г. Гейн. Информационно-методическое обеспечение профессиональной деятельности педагога, педагога-психолога, работника школьной библиотеки*  
21-008 *А.Н. Майоров. Основы теории и практики разработки тестов для оценки знаний школьников*

Имеются два варианта учебных материалов дистанционных курсов: брошюры и брошюры+DVD.

Курсы, включающие видеолекции (DVD), помечены значком 

Нормативный срок освоения каждого курса – 72 часа. Дополнительная информация – на сайте <http://edu.1september.ru>.

Окончившие дистанционные курсы получают удостоверение установленного образца.

Базовая стоимость курса (без учета скидок) составляет 1990 руб. для курсов без видеоподдержки и 2190 руб. – для курсов с видеоподдержкой.



**ОЧНЫЕ КУРСЫ ПОВЫШЕНИЯ КВАЛИФИКАЦИИ  
ДЛЯ ЖИТЕЛЕЙ МОСКВЫ И МОСКОВСКОЙ ОБЛАСТИ  
(обучение с 1 февраля по 30 апреля 2012 года)**

*Я.Н. Зайдельман. Алгоритмизация и программирование: от первых шагов до подготовки к ЕГЭ*

Нормативный срок освоения каждого курса – 72 часа.

Дополнительная информация – на сайте <http://edu.1september.ru>

и по телефону (499) 240-02-24 (звонки принимаются с 15.00 до 19.00).

Окончившие очные курсы получают удостоверение государственного образца.

Базовая стоимость курса (без учета скидки) – 5400 руб.



Электронную заявку можно в режиме online подать  
на сайте <http://edu.1september.ru>. Это удобно и просто!

# АКЦИЯ-2012

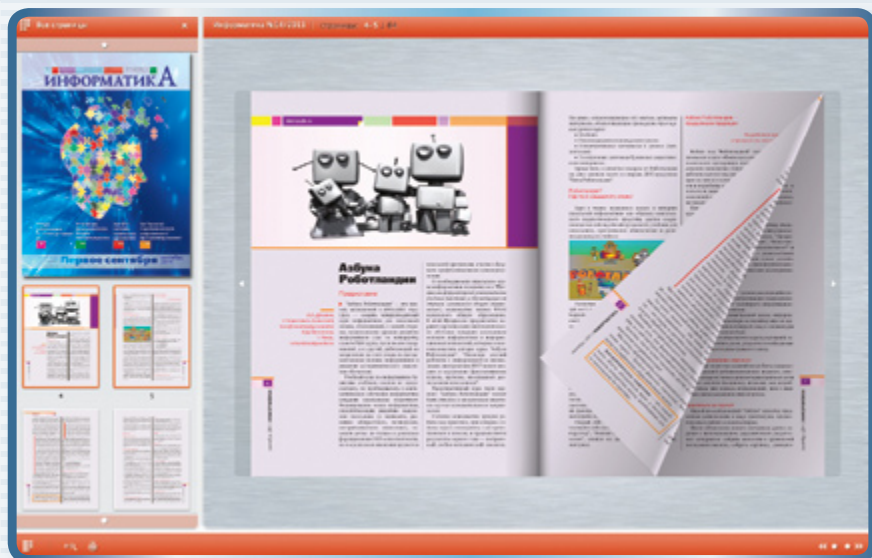
ж у р н а л

## Информатика – Первое сентября

Подписка на электронную версию журнала продолжается!

Стоимость подписки – **200 рублей** за полгода.

Сайт [www.1september.ru](http://www.1september.ru)



Доступный формат  
для всех уровней  
ИКТ-компетентности:

- чтение on-line
- скачивание PDF на компьютер
- распечатывание
- дополнительные материалы к уроку
- журнал и дополнительные материалы доступны с любого компьютера

Каждый подписчик электронной версии журнала уже в феврале получит по почте именной сертификат по ИКТ-компетентности.

**Свежий номер журнала  
в вашем Личном кабинете –  
1 января!**