А. ШЕНЬ

Информатика в IX классе

20 **Глубокоуважаемый коллега-учитель! Предлагаем вашему вниманию методические разработки по курсу «Основы информатики и вычислительной техники». Когда создавалась первая часть пробного учебного пособия и книги для учителей, опыт преподавания основ информатики и вычислительной техники практически отсутствовал. За прошедшее время он появился и в какой-то мере отражен в предлагаемых разработках, но прежде всего мы призываем вас руководствоваться собственным опытом при распределении материала по урокам, выборе разбираемых примеров и т. д.**

Введение

Роль ЭВМ в современном обществе

Цель темы — создать у школьников представление (хотя бы самое приблизительное) о возможностях современных ЭВМ и их приложениях. Помимо приведенных в первой части пробного учебного пособия (далее — УП) материалов рекомендуем воспользоваться сведениями из второй части УП.

Первоначальные сведения об ЭВМ

Если в предыдущей теме объяснялось, что может сделать ЭВМ, то теперь нужно дать хотя бы самое примитивное представление о том, как ЭВМ это делает. Основные моменты:

информация представляется в ЭВМ в виде электрических сигналов;

обработку информации производит *процессор* в соответствии с хранящейся в памяти *программой;* одна и та же ЭВМ может решать совсем разные задачи, в зависимости от того, какая программа в нее заложена;

технические характеристики ЭВМ очень быстро совершенствуются.

Рекомендуем познакомиться с материалами второй части УП, содержащими более подробное изложение сведений об устройстве ЭВМ, истории вычислительной техники и др.; кроме того, советуем не слишком задерживаться на введении и как можно скорее идти дальше: впереди ждет трудный материал, и сэкономленное время очень пригодится.

Упражнения на с. 15 УП предназначены прежде всего для того, чтобы создать представление о технических возможностях современных ЭВМ. Ниже приводятся их решения.

1. На этот подсчет уходит около минуты; всего в УП около 100 страниц, на каждой около 50 строк, на каждую страницу уйдет примерно 5 минут, всего 500 минут, а компьютер тратит 1/6 минуты, т. е. работает в 3000 раз быстрее.

2. 1 рубль = 25 кВт·ч электроэнергии (внимание: киловатты умножаются на часы, а не делятся, как в УП!), т. е. компьютер мощностью в 100 Вт = = 1/10 кВт может работать 250 часов = = 250 · 3600 секунд = 900 000 секунд.

За каждую из них он делает 100 000 операций, так что всего будет 90 000 000 000 (90 млрд.) операций.

3. 24 строки по 80 символов занимают (считая, что символ занимает 1 байт) около 2000 байтов $\approx$2K байта. Поскольку 1M байт равен 1024K байт, то информация занимает около 2/100 000 части диска, т. е. примерно 0,02 %.

4. При проведении такого эксперимента на чтение одного абзаца из 8 строк ушло 15 секунд (около 2 секунд на строчку из примерно 60 символов). Скорость чтения — около 30 символов в секунду. Скорость письма: чтобы написать это предложение (а в нем примерно 100 символов), понадобилось около 30 секунд. Скорость письма — примерно 3 символа в секунду. Чтобы прочесть вслух то же предложение, требуется около 6 секунд, поэтому скорость чтения вслух — примерно 15 символов в секунду.

5. Если в секунду произносится около 15 символов, символ занимает 1 байт, то на заполнение памяти уйдет около 4 тыс. секунд — больше часа.

6. Будем считать, что объем школьного сочинения — три машинописные страницы ($\approx$6000 символов). Тогда на диске уместится около 90 сочинений.

7. На соответствующей странице УП — около 15 строк, в каждой — около 50 символов. На печать одной строки уходит примерно половина секунды, значит, всего потребуется 8 секунд.

8. Для выполнения каждой операции сигнал должен пройти от процессора к памяти и обратно — это 60 см=0,6 м. За секунду ему пришлось бы пройти путь 600 000 000·0,6 м = 360 000 000 м = =360 000 км, т. е. его скорость должна быть не меньше 360 000 км/с, что больше скорости света (300 000 км/с), являющейся предельной.

Раздел 1. Алгоритмы. Алгоритмический язык

§ 1. Алгоритм и его свойства

Мы дадим рекомендации сразу по двум пунктам (понятие алгоритма и формальное исполнение алгоритма), поскольку их материал тесно связан.

Прежде всего одно общее замечание. Основной принцип изложения в УП — обучение на примерах. Поэтому не следует пытаться добиться от школьников знания «определения» алгоритма или его «свойств» или долго обсуждать словесные формулировки, приведенные в УП.

Основным примером является алгоритм Евклида. Хотя он и весьма древний, его понимание требует некоторого знакомства с математикой. Мы приведем его подробное обоснование, которое предназначено прежде всего для вас. Что окажется возможным рассказать на уроках, зависит от математических знаний ваших учеников. В сомнительных случаях стоит воспользоваться советом из методического письма «О преподавании курса «Основы информатики и вычислительной техники» в 1986/87 учебном году» и обсудить алгоритм Евклида и алгоритм игры Баше (следующий пример) после изучения составных команд, если останется время.

Алгоритм Евклида можно сформулировать так.

На доске пишутся два натуральных числа. Затем повторяется одно и то же действие:

пока числа не станут равными, большее число заменяется на разность большего и меньшего чисел.

Пример: 15,39→15,24→15,9→ 6,9→6,3→3,3.

Как видим, алгоритм очень прост: чтобы его выполнить, нужно только уметь вычитать. Замечательно, что такой простой алгоритм позволяет решить довольно сложную задачу — найти наибольший общий делитель двух натуральных чисел. Тут нужны некоторые напоминания:

натуральное число a называется делителем натурального числа b, если b делится на a без остатка, т. е. если b/a — целое число;

общие делители чисел m и n — те числа, которые одновременно являются делителями и m, и n;

наибольшее число, являющееся общим делителем m и n, называется наибольшим общим делителем чисел m и n (обозначение НОД (n, m)).

Так вот, оказывается, что применение к паре чисел (m, n) алгоритма Евклида дает в конце концов пару одинаковых чисел, которые равны НОД (m, n).

Остановимся и удивимся. Ведь в описании алгоритма Евклида нет ни слова о делителях или делении; единственная операция, которая в нем используется,— это вычитание. Почему же в результате его работы получается именно наибольший общий делитель, а не что-нибудь другое?

Отметим для начала, что

при замене большего числа на разность большего числа и меньшего их наибольший общий делитель не меняется: НОД (m, n) = НОД$(m-n, n)$ (при $m>n$).

(В нашем примере было НОД (15,39)=НОД (15,24)=НОД (15,9)= =НОД (6,9)=НОД (6,3)=НОД (3,3)= =3.) Это верно потому, что у пары чисел (n, m) и у пары $(m-n, n)$ одни и те же общие делители, и, следовательно, один и тот же наибольший общий делитель. Чтобы доказать это, нужно использовать такое утверждение:

сумма и разность двух чисел, делящихся на некоторое число d, делится на это число d.

Доказывается оно просто: если m/d — целое число и n/d — целое число, то $(m+n)/d=m/d+n/d$ и $(m-n)/d= =m/d-n/d$ — тоже целые числа. Отсюда следует, что если d — общий делитель чисел m и n, то он является и общим делителем $m-n$ и n. Если же d является общим делителем чисел $m-n$ и n, то и их сумма $(m-n)+n=m$ делится на d, так что d — общий делитель m и n.

Теперь секрет алгоритма ясен: в процессе его работы наибольший общий делитель не меняется, и в конце, когда числа станут равными, они будут наибольшим общим делителем. Ведь общий делитель двух равных чисел — это любое из них.

Выделим еще раз основные этапы этого рассуждения:

если m и n делятся на d, то $m+n$ и $m-n$ делятся на d;

общие делители у чисел $(m-n)$ и n — те же, что у чисел m и n;

НОД$(m-n, n)$=НОД(m, n);

при выполнении алгоритма Евклида наибольший общий делитель пары чисел остается неизменным;

алгоритм Евклида кончает работу, когда остаются равные числа; поскольку НОД$(x, x)=x$, то x равен наибольшему общему делителю исходной пары чисел.

Подчеркнем, что приведенные рассуждения нужны лишь для обоснования алгоритма Евклида; для его исполнения ничего этого не нужно — достаточно уметь проверять, равны ли числа, указывать, какое из чисел больше, и вычитать меньшее из большего.

Еще один вопрос, связанный с алгоритмом Евклида: почему его работа рано или поздно закончится? В УП в этом месте ошибка, повторенная в методическом пособии для учителей. Правильное объяснение состоит в том, что с каждым шагом максимум чисел уменьшается, оставаясь положительным, а это не может происходить бесконечно много раз.

Подробный разбор алгоритма Евклида может занять целый урок, а то и больше. Однако это время не будет потрачено зря.

Другой интересный пример, имеющийся в УП,— алгоритм выигрыша в игре Баше с 15 предметами. Советуем, не сообщая алгоритма школьникам, дать возможность им поиграть друг с другом в эту игру (это удобно сделать, нарисовав на доске или бумаге 15 палочек и стирая или зачеркивая их). При этом некоторые из них сами могут изобрести алгоритм выигрыша; во всяком случае они лучше его оценят.

Почему приведенный в УП алгоритм действительно приводит к выигрышу? После первого хода остается 13 предметов. Затем (после ходов обоих противников) остается 13—4=9, потом 9—4= =5, и наконец остается один предмет. Его вынужден взять последний из играющих.

Интересно также проанализировать игру Баше с другим количеством предметов. Составим таблицу, показывающую, является ли позиция с данным числом предметов выигрышной или проигрышной для того, кто в ней окажется. Позиция с одним предметом — проигрышная по определению. Позиции с 2, 3 и 4 предметами — выигрышные:

находясь в них, можно поставить противника в проигрышное положение, оставив ему один предмет (взяв 1, 2 или 3 предмета). Позиция с 5 предметами является проигрышной: какой ход ни делай, останется 2, 3 или 4 предмета, а эти ситуации выигрышные, и противник сможет выиграть (если будет играть правильно). Раз позиция с 5 предметами — проигрышная, то позиции с 6, 7 и 8 предметами — выигрышные: из них можно, взяв 1, 2 или 3 предмета соответственно, поставить противника в проигрышное положение, оставив ему 5 предметов. Позиция с 9 предметами — снова проигрышная: после любого хода противнику остается выигрышная позиция с 6, 7 или 8 предметами. Позиции с 10, 11 и 12 предметами — снова выигрышные, с 13 — проигрышная, с 14, 15 и 16 — снова выигрышная и т. д.

Предметов	1	2	3	4	5	6	7	8
Позиция	П	В	В	В	П	В	В	В

Предметов	9	10	11	12	13	14	15
Позиция	П	В	В	В	П	В	В

Из этого рассуждения видно, что алгоритм выигрыша состоит в том, чтобы оставлять противника в проигрышной позиции — с 13, 9, 5 и, наконец, с 1 предметом.

Рассмотренные примеры показывают, что смысл составления алгоритма в том, чтобы разбить решение интересующей нас задачи на отдельные простые шаги (команды). Например, алгоритм Евклида разбивает одну команду «найти наибольший делитель» на отдельные шаги, каждый из которых требует лишь вычитания. Исполнитель этого алгоритма должен уметь лишь сравнивать числа и вычитать большее число из меньшего. Составляя алгоритм для заданного исполнителя, нужно учитывать его систему команд — перечень доступных ему действий.

Разбиение решения задачи на отдельные шаги иллюстрируется в УП, пример 1.1. Там составляется алгоритм вычисления выражения $y=(Ax+B)/(Cx-D)$ для исполнителя, который может выполнять арифметические действия с парой чисел и запоминать их результат, но не может производить вычисления по более сложным формулам. Аналогичная задача содержится в упражнении 1 на с. 21. Обратная задача (восстановление формулы по алгоритму) поставлена в упражнении 2. Эти упражнения можно использовать на уроке. Можно также предложить части учеников составить по формуле алгоритм вычисления ее значения, а затем передать этот алгоритм другой части, чтобы те восстановили формулу (не зная ее заранее).

В УП есть и другие примеры, но мы не рекомендуем разбирать их все — лучше хорошо разобрать один, чем пройти мимоходом много. Не следует также задерживаться на вопросах формального, словесного характера (типа приведенных на с. 21 УП).

Решения упражнений на с. 21—22 УП.

1. а)

Умножить x на 2, результат обозначить R1 | R1 : $=2x$
Сложить R1 с 3, результат обозначить R2 | R2 : $=$R1$+3$
Умножить x на 7, результат обозначить R3 | R3 : $=7x$
Вычесть 5 из R3, результат обозначить R4 | R4 : $=$R3-5
Разделить R2 на R4, результат обозначить y | $y:=$R2/R4

Справа приведена сокращенная запись в том виде, в каком она будет встречаться впоследствии.

б) R1 : $=x-3$; R2 : $=$R1$*$R1; R3 : $=2-$R2; R4 : $=$R2$+4$; $y:=$R3/R4.

2. а) При выполнении этого алгоритма R1$=x^2$, R2$=a\cdot$R1$=ax^2$, R3$=$R2$+b=ax^2+b$, $y=$R3$/c=(ax^2+b)/c$.

Ответ: $y=(ax^2+b)/c$.

б) $A_1=x+1$, $A_2=1/A_1=1/(x+1)$, $A_3=A_2+1=1/(x+1)+1$, $A_4=A_2-1=1/(x+1)-1$, $A_5=A_4/A_3=(1/(x+1)-1)/(1/(x+1)+1)$.

Ответ: $y=A_5-1=(1/(x+1)-1)/(1/(x+1)+1)-1$.

3. Решение этой задачи хорошо известно из курса математики, а также приведено в методическом пособии для учителей на с. 38.

4. В соответствии с алгоритмом первый возьмет два предмета. Если после этого второй возьмет 1 предмет, оставив

первому 17, то поставит первого в проигрышное положение.

Упражнения 5 и 6 использовать не рекомендуем.

7. Шаг 3 (вычитание) будет выполняться 10 раз: 100, 18→82,18→64,18→ →46,18 → 28,18 → 10,18 → 10,8 → 2,8 → →2,6→2,4→2,2. (В методическом пособии дан неверный ответ.)

§ 2. Алгоритмический язык

Алгоритмический язык предназначен для записи алгоритмов. Переход от словесных описаний алгоритмов к их записям на алгоритмическом языке играет в информатике ту же роль, что и введение буквенных обозначений в алгебре.

На его изучение отводится 4 урока, но можно занять и большее время. Не следует торопиться: лучше (временно) отстать от программы, чем излишней спешкой сделать дальнейшие уроки бессмысленными.

Общие правила алгоритмического языка

Правила записи алгоритмов на алгоритмическом языке будут постепенно уточняться. Начнем с внешнего оформления алгоритмов. Это оформление (с. 23 УП) можно проиллюстрировать на примере 3.1 или 3.2. После разбора примера следует предложить учащимся самостоятельно оформить какие-либо известные им алгоритмы в виде текстов на алгоритмическом языке.

1.

алг сделать чай сладким
нач
положить сахар
размешать
кон

Что изменится, если мы переставим команды алгоритма «положить сахар» и «размешать»? (О т в е т. Сахар останется неразмешанным.)

Этот пример показывает, что порядок команд в алгоритме очень важен.

2.

алг кипячение
нач
налить воду в чайник
поставить чайник на плиту
зажечь газ
кон

Здесь некоторые команды можно переставить. Можно зажечь газ сначала, а поставить чайник на плиту потом. При этом получится другой, но тоже правильный алгоритм. Но если мы переставим первую и третью команды и будем действовать в таком порядке: зажечь газ — поставить чайник на плиту — налить воду в чайник, то до выполнения третьей команды чайник, стоя на огне без воды, может испортиться.

Предлагая школьникам записывать алгоритмы на алгоритмическом языке, нужно иметь в виду, что в этот момент они могут записать лишь простейшие алгоритмы, сводящиеся к последовательному выполнению команд. Для записи более сложных алгоритмов (например, алгоритма Евклида) на алгоритмическом языке необходимы средства, пока не известные ученикам (составные команды).

Общие правила записи очень просты: по существу, они сводятся к тому, что перед началом алгоритма нужно написать служебное слово **алг,** название и служебное слово **нач,** а в конце — служебное слово **кон.**

Алгоритмы работы с графической информацией

Хотя эти алгоритмы описаны в конце первой части УП, в соответствии с рекомендациями методического письма материал по ним перенесен в начало курса.

Исполнителя алгоритмов работы с графической информацией можно представить себе как маленькую черепашку, движующуюся по плоскости и оставляющую след. Система команд этого исполнителя такова:

вперед (*n*)
назад (*n*)
налево (А)
направо (А)
рисуй
не рисуй

По команде «вперед (*n*)» черепашка движется на расстояние *n* единиц вперед, по команде «назад (*n*)» — назад. По команде «направо (А)» черепашка

поворачивается на угол в А градусов по часовой стрелке, по команде «налево (А)» — против часовой стрелки. По команде «не рисуй» черепашка перестает оставлять след, по команде «рисуй» — вновь начинает.

Упражнения.

1. Выполнены команды

вперед (2)
назад (2)

Где оказалась черепашка? Что нарисовано? (Ответ. Черепашка находится в исходном положении, видя перед собой нарисованный ею отрезок длины 2.)

2. Есть ли разница между действием серии команд

вперед (2)
вперед (3)

и команды «вперед (5)»? (Ответ. Нет; и в том и в другом случае черепашка сдвинется на 5 единиц вперед и нарисует отрезок длины 5.)

3. Есть ли разница между серией команд

вперед (2)
назад (3)

и командой «назад (1)»? (Ответ. Положение черепашки будет одинаковым, однако нарисовано будет разное: в первом случае будет нарисован отрезок длины 3, а во втором — отрезок длины 1.)

4. Можно ли найти такое А, чтобы команды «налево (А)» и «направо (240)» делали бы одно и то же? (Ответ. Можно, например: А=120.)

5. Есть ли разница между серией команд

налево (180)
вперед (1)

и командой «назад (1)»? (Ответ. Есть. Хотя в обоих случаях черепашка сдвинется одинаково, в первом она повернется на 180 градусов, а во втором сохранит первоначальную ориентацию.)

6. Как составить серию команд, которая производила бы то же самое действие, что и команда назад (1), но не содержала бы команд «назад»? (Ответ. Например, годится такая серия:

налево (180)
вперед (1)
налево (180))

Разобравшись с системой команд исполнителя, можно составлять простейшие алгоритмы. Вот несколько примеров.

1)

```
алг правильный треугольник со стороной 4
нач
    вперед (4)
    направо (120)
    вперед (4)
    направо (120)
    вперед (4)
кон
```

2)

```
алг буква М
нач
    вперед (4)
    налево (30)
    назад (2)
    направо (60)
    вперед (2)
    налево (30)
    назад (4)
кон
```

3)

```
алг буква И
нач
    вперед (4)
    назад (4)
    направо (30)
    вперед (4,8)
    налево (30)
    назад (4)
кон
```

4)

```
алг буква Р
нач
    вперед (4)
    направо (90)
    вперед (2)
    направо (90)
    вперед (2)
    направо (90)
    вперед (2)
кон
```

5)

```
алг шаг вправо
нач
    не рисуй
    направо (90)
    вперед (1)
    налево (90)
    рисуй
кон
```

Теперь можно составить алгоритм рисования слова МИР. Чтобы нарисовать это слово, необходимо последовательно выполнить такие действия:

написать букву М;
сделать шаг вправо;
написать букву И;
сделать шаг вправо;
написать букву Р.

Записав последовательно соответствующие команды, получим алгоритм, приведенный в УП (с. 80, пропущены лишь команды «рисуй» и «не рисуй» при написании буквы И — они лишние).

К этому примеру полезно вернуться при изучении темы «Вспомогательные алгоритмы». Ведь, используя построенные ранее алгоритмы как вспомогательные, можно записать алгоритм рисования слова МИР совсем просто:


```
алг написание слова «МИР»
нач
    буква М
    шаг вправо
    буква И
    шаг вправо
    буква Р
кон
```

Раздел «Алгоритмы работы с графической информацией» содержит еще некоторые примеры, которые пока использовать нельзя, так как они предполагают знакомство с составными командами, величинами и другими еще не изученными понятиями.

Составные команды

Команда ветвления. При составлении алгоритмов да и просто в жизни часто встречаются ситуации, когда необходимо сообразовываться с обстоятельствами, выполняя то или иное действие. Мы одеваемся, выходим из дому. Если на улице холодно, то надеваем пальто, а иначе — куртку. Выйдя на улицу, мы смотрим, идет ли автобус. Если он есть, то садимся и едем на автобусе, а иначе идем пешком. На алгоритмическом языке это запишется так:

```
алг одеться
нач
    если на улице холодно
      то
        надеть пальто
      иначе
        надеть куртку
    все
кон
и
алг маршрут
нач
    если подошел автобус
      то
        ехать на автобусе
      иначе
        идти пешком
    все
кон
```

Смысл команды ветвления (в полной форме, с. 25 УП) часто оказывается непонятным. Нужно обратить внимание в первую очередь на следующее.

1. Слова **если, то, иначе, все** не являются командами и не исполняются. Это служебные слова, указывающие порядок выполнения команд. Их роль можно сравнить с ролью знаков препинания в русском языке или скобок в алгебраических формулах.

2. При выполнении команды ветвления выполняется либо серия команд между **то** и **иначе,** либо серия команд между **иначе** и **все.** Одна из этих серий обязательно будет выполнена. Ни в каком случае не будут выполнены обе серии. Завершив выполнение одной из серий, исполнитель тем самым завершает выполнение команды ветвления (после чего переходит к следующей команде, если она есть).

Употребление слов русского языка (**если, то, иначе**) при записи алгоритмов имеет и преимущества, и недостатки. Преимущество состоит в том, что служебные слова напоминают о смысле соответствующих конструкций и позволяют читать алгоритм почти как текст на русском языке. С другой стороны, аналогия с русским языком может служить источником недоразумений. Вот один из многочисленных примеров ошибочного употребления служебных слов (сообщен Ю. Махлиным):

```
если нога поднята
  то
```

```
        опустить ногу
    иначе
        упадешь
все
```

Приведем несколько примеров команд ветвления.

```
если последняя цифра числа равна 0 или 5
  то
    сообщить, что число делится на 5
  иначе
    сообщить, что число не делится на 5
все
если первое число меньше второго
  то
    вычесть первое число из второго
  иначе
    вычесть второе число из первого
все
```

Сокращенная форма команды ветвления. Часто оказывается, что в одном из двух возможных в команде ветвления случаев никаких действий предпринимать не нужно. Тогда применяется сокращенная форма команды ветвления (с. 25 УП). Ее отличие от полной состоит в том, что отсутствуют слово **иначе** и серия идущих за ним команд. Связь сокращенной формы команды ветвления с полной можно пояснить так: команда ветвления в сокращенной форме

```
если условие
  то
    серия
все
```

эквивалентна команде ветвления в полной форме

```
если условие
  то
    серия
  иначе
    ничего не делать
все
```

где «ничего не делать» — команда, не требующая никаких действий от исполнителя.

В качестве примера использования сокращенной формы команды ветвления разберем алгоритм включения электроприбора в сеть 220 В, приведенный на с. 26 УП. При этом можно использовать следующие вопросы.

1. Останется ли алгоритм верным, если электроприбор имеет переключатель на три напряжения — 110, 127 и 220 В? (Ответ. Нет, не останется: если переключатель был установлен на 110 В, то он останется в таком положении и при включении в сеть прибор сгорит.)

2. Как исправить алгоритм, чтобы он оставался верным и для прибора с переключателем на 3 напряжения?

Ответ.

```
алг включение электроприбора
    в сеть 220 В
нач
  если переключатель прибора
      установлен не на 220 В
    то
      установить переключатель
      прибора на 220 В
  все
  включить вилку в розетку
кон
```

Другой пример алгоритма с использованием команд ветвления таков:

```
алг завтрак
нач
  если картошка не очищена
    то
      очистить картошку
      сварить картошку
  все
  съесть картошку
кон
```

Правилен ли этот алгоритм? Нет: если картошка была очищена, то исполнителю придется есть ее сырой. Как его исправить? Например, так:

```
алг завтрак
нач
  если картошка не очищена
    то
      очистить картошку
      сварить картошку
    иначе
      сварить картошку
  все
  съесть картошку
кон
```

Другой вариант исправления, использующий сокращенную форму команды ветвления, таков:

```
алг завтрак
нач
  если картошка не очищена
    то
      очистить картошку
  все
  сварить картошку
  съесть картошку
кон
```

Еще одна задача. Ученик, начав записывать алгоритм определения кислотности раствора (УП, пример 4.3), ошибся в первом условии, написав «если бумажка синяя» вместо «если бумажка красная». Как он должен изменить оставшуюся часть алгоритма, чтобы алгоритм снова стал правильным?

Ответ. Например, так:


```
если бумажка синяя
  то
    ответ: раствор щелочной
  иначе
    если бумажка красная
      то
        ответ: раствор кислотный
      иначе
        ответ: раствор нейтральный
  все
все
```

Команда повторения. Это центральная и, наверно, самая трудная тема курса информатики. К сожалению, программа отводит на нее до смешного мало времени, а в УП мало примеров и почти нет упражнений. Поэтому приведем большое количество примеров, из которых можно выбрать наиболее подходящие для использования на уроках, в качестве домашних заданий и т. п.

Пример 1.

```
пока переключатель каналов не установлен на 10
нц
  повернуть ручку переключателя на одно деление вправо
кц
```

Этот пример иллюстрирует такое свойство команды повторения: после ее выполнения условие не соблюдается, т. е. переключатель установлен на 10. Вопрос: сколько раз будет выполняться команда, входящая в команду повторения, если переключатель установлен на 9? на 1? на 11? на 10? Ответы: 1 раз, 9, 11 (всего каналов 12), 0 раз.

Пример 2. Команда «провести воспитательную работу» может быть описана так:

```
пока нарушитель не раскаялся
нц
  провести воспитательную беседу с нарушителем
кц
```

В результате выполнения этой команды беседа повторяется до тех пор, пока это необходимо. Отметим три важных обстоятельства: если нарушитель раскаялся с самого начала, то беседа не проводится ни разу; команда «провести воспитательную работу» может выполняться бесконечно долго (или, как говорят, зациклиться); но если она кончит работу, то нарушитель окажется раскаявшимся.

Пример 3 (подготовительный к примеру из УП).

```
пока ведро не полно
нц
  долить в ведро литр воды
кц
```

Этот пример иллюстрирует то же свойство: после выполнения команды условие не соблюдается, т. е. ведро полно. Вопрос: сколько раз будет выполняться команда, если в 7-литровом ведре был 1 л? 6 л? 7 л? 3,5 л? Ответы: 6 раз; 1 раз; 0 раз; 4 раза. В последнем случае пол-литра воды проливается. Это происходит потому, что условие в команде повторения проверяется не во время выполнения серии входящих в нее команд, а перед выполнением. Еще вопрос: можно ли придумать такие условия (такую емкость ведра и количество налитой в него воды), чтобы после выполнения этой команды ведро было бы не полным? Ответ: нельзя, после выполнения команды повторения входящее в нее условие не соблюдается.

Пример 4 (4.4 по УП, с. 28) иллюстрирует еще более наглядно, что в середине выполнения серии команд проверка не производится. В качестве упражнения в УП предлагается изменить алгоритм так, чтобы вода не проливалась. Это можно сделать различными способами, один из них таков:

```
пока ведро не полно
нц
  долить 1 л холодной воды
  если ведро не полно
    то
      долить 1 л горячей воды
  все
кц
```

Пример 5.

```
алг завтрак
нач
  пока в тарелке не меньше ложки каши
  нц
      съесть ложку каши
  кц
  выпить чай
кон
```

Сколько каши останется несъеденной в тарелке? Меньше ложки, так как после выполнения команды повторения условие не соблюдается. Что было бы, если бы «не меньше» заменить на «больше»? В этом случае можно было бы утверждать, что в тарелке останется ложка каши или меньше. Что было бы, если мы бы поменяли местами строки «кц» и «выпить чай»? Новый алгоритм требовал бы запивать стаканом чая каждую ложку каши.

Пример 6.

```
пока написанное на доске число меньше 10
нц
  увеличить написанное число на 1
кц
```

Что будет после выполнения команды, если вначале было написано число 7? 10? 11? (Ответ: 10; 10; 11.) Сколько раз в каждом из этих случаев выполнится команда повторения? (Ответ: 3, 0, 0.)

Пример 7.

```
пока написанное на доске число больше 10
нц
  увеличить написанное число на 1
кц
```

Что будет, если на доске до выполнения команды повторения было написано 7? 10? 11? (Ответ. 7; 10; в третьем случае выполнение команды повторения не закончится никогда.)

Два предыдущих примера легко видоизменить и использовать для упражнений.

Пример 8. Объяснить разницу между командами

```
пока в тарелке не меньше ложки каши
нц
  съесть ложку каши
кц
```

и

```
если в тарелке не меньше ложки каши
  то
    съесть ложку каши
все
```

(Ответ. Во втором случае исполнитель съест только одну ложку каши (если она была) и остановится. В первом случае он будет есть кашу, пока это возможно, и остановится, только когда каши останется меньше ложки.)

Пример 9. На доске написано целое число. Выполняется такая команда:

```
пока число больше или равно 10
нц
  уменьшить число на 10
кц
```

Что останется, если вначале было написано 15? 10? — 15? 123456789? (Ответ: 5; 0; — 15; 9.)

Последний ответ объясняется так: после каждого выполнения команды, входящей в команду повторения, последняя цифра числа будет равна 9 (так как вначале она равна 9 и не меняется при вычитании 10 из числа, большего или равного 10), а в конце число должно быть меньшим 10, т. е. однозначным.

Пример 10.

```
пока число на доске не меньше 7
нц
  уменьшить написанное число на 7
кц
```

Что останется после выполнения команды, если вначале на доске было написано натуральное число А? Останется остаток от деления А на 7. В самом деле, при уменьшении числа на 7 этот остаток не меняется. Таким образом, число, которое будет после выполнения команды, дает тот же остаток при делении на 7, что и исходное. С другой стороны, оно мень-

ше 7 (условие в команде повторения не соблюдается после выполнения команды) и неотрицательно, так как мы вычитаем 7 только из чисел, не меньших 7. Упражнение: написать по образцу этой команды команду нахождения остатка при делении на 9. (Решение: заменить в двух местах команды 7 на 9.)

Пример 11.

```
пока число на доске содержит больше
      одной цифры
нц
  заменить число на доске суммой его
  цифр
кц
```

На доске было написано положительное число А. Что останется на доске после выполнения команды? Будет написан остаток от деления числа А на 9 (или 9, если этот остаток равен 0). В самом деле, сумма цифр положительного целого числа дает при делении на 9 тот же остаток, что и само это число (на этом свойстве основан известный признак делимости на 9). Поскольку после выполнения команды должно получиться однозначное число (после выполнения команды условие не соблюдается), то оно будет равно остатку от деления А на 9 (или 9, если остаток равен 0).

Пример 12. На доске написано пятьдесят семь единиц и девяносто одна минус единица. Выполняется такая команда:

```
пока на доске написано больше одного
      числа
нц
  стереть какие-то два числа и заменить
  их на их произведение
кц
```

Что будет написано на доске после выполнения команды? Будет написано единственное число, так как после выполнения команды повторения условие не соблюдается. Оно будет равняться —1, так как произведение всех чисел на доске остается неизменным. Что было бы, если вначале на доске было бы 100 чисел, каждое из которых равно 2? Осталось бы одно число, равное 2 в степени 100.

Пример 13. Запишем, наконец, алгоритм Евклида.

```
алг нахождение НОД двух натуральных
      чисел
нач
  пока числа не равны
  нц
      если первое число больше вто-
            рого
      то
        вычесть второе число из пер-
        вого
      иначе
        вычесть первое число из вто-
         рого
    все
  кц
кон
```

Резюмируем коротко основные свойства команды повторения:

после выполнения команды повторения условие не соблюдается;

серия команд, входящих в команду повторения, выполняется как единое целое, выполнение которого не прерывается, даже если условие перестало соблюдаться;

может случиться, что команда повторения не закончит работу («зациклится»), если после любого числа повторений условие не перестает соблюдаться;

если какая-то величина не меняется при каждом выполнении серии команд из команды повторения, то она не меняется и при выполнении команды повторения целиком, что позволяет предсказать результат выполнения команды повторения.

Не страшно, если изучение команды повторения не уложится в отводимые два урока, а, например, потребует месяца или двух. Гораздо хуже, если эта тема останется не понятой учениками. Поэтому настойчиво рекомендуем вам не двигаться дальше, пока с командой повторения (или с командой ветвления, но она проще) не станет всем все ясно.

Продолжение следует

5/1987

А. ШЕНЬ

Информатика в IX классе

Алгоритмы работы с величинами

Аргументы и результаты. До сих пор рассматривались алгоритмы управления исполнителем, действующим в какой-то обстановке и меняющим ее в ходе исполнения алгоритма (наливающим воду, переключающим электроприбор, рисующим на плоскости и т. д.). Теперь перейдем к алгоритмам, которые не производят каких-либо изменений в окружающем их мире, а лишь перерабатывают поступающую информацию и выдают результаты этой переработки.

Исполнителя такого алгоритма можно сравнить с экспертом, сидящим у себя в кабинете. Ему звонят по телефону и сообщают исходные данные, необходимые для работы (их называют аргументами). После этого он выключает телефон и начинает работать. Кончив работу, он вновь включает телефон и сообщает результаты своей работы.

Понятия аргументов и результатов алгоритма можно пояснить такими примерами.

1. Задача: найти наибольший общий делитель двух натуральных чисел. Аргументы: два натуральных числа. Результаты: натуральное число, являющееся их наибольшим общим делителем.

2. Задача: подсчитать количество букв «а» в слове. Аргументы: слово. Результаты: целое число, равное количеству букв «а» в этом слове.

3. Задача: вычислить квадратный корень. Аргументы: вещественное число. Результаты: вещественное число, являющееся квадратным корнем из исходного.

4. Задача: разделить одно натуральное число на другое с остатком. Аргументы: два натуральных числа (делимое и делитель). Результаты: два целых числа (частное и остаток).

5. Задача: сложить числа в таблице. Аргументы: таблица, заполненная вещественными числами. Результаты: вещественное число, равное сумме всех чисел таблицы.

6. Задача: перевести слово по русско-английскому словарю. Аргументы: русское слово. Результаты: английское слово, являющееся переводом русского.

7. Задача: найти центр окружности, проходящей через данные три точки. Аргументы: три точки плоскости. Результаты: точка плоскости, являющаяся центром искомой окружности.

Типы. В каждой из перечисленных задач аргументы и результаты — данные какого-то определенного типа. Нельзя искать, например, наибольший делитель двух слов, переводить вещественное число по русско-английскому словарю или строить центр окружности, проходящей через данные три числа. При записи алгоритмов мы будем указывать типы исходных данных (аргументов) и результатов, т. е. указывать, какие значения они могут принимать. В школьном курсе информатики рассматриваются четыре типа: целый, вещественный, натуральный и литерный. Данные целого, вещественного и натурального типов — соответственно, целые, вещественные и натуральные числа. Данные литерного типа — произвольные последовательности символов.

Понятие типа и его использование в УП вызывают много вопросов. Обсудим некоторые из них.

Продолжение. Начало см.: Информатика и образование. 1987. № 4.

1. Бывают ли другие типы, кроме перечисленных?

Прежде всего надо иметь в виду, что вопрос поставлен не вполне правильно. Типы, используемые в алгоритмическом языке, определены его разработчиками, так что спрашивать можно так: «Включены ли в алгоритмический язык учебника другие типы?» Да, включены. Это типы, связанные с табличными величинами (однако они плохо продуманы авторами, о чем мы еще будем говорить).

Другое возможное уточнение вопроса: «Возникает ли при решении задач необходимость в других типах, кроме приведенных в УП?» Да. Это видно уже из примера 7, где необходим тип «точка плоскости». Недаром во многих языках программирования имеются другие типы и даже средства, позволяющие программисту (составителю алгоритма) самому создать (определить) новый тип. Например, на Паскале можно написать

```
TYPE ТочкаПлоскости=
  RECORD
    Хкоорд: REAL;
    Укоорд: REAL
  END;
```

При этом определяется новый тип (TYPE), который называется «ТочкаПлоскости»; данное этого типа запись (RECORD) из двух вещественных (REAL) чисел: Хкоорд и Укоорд. После этого слово «ТочкаПлоскости» можно использовать как название типа наряду с типами REAL (**вещ**) и др.

2. В примере 2 (вычисление квадратного корня) аргументом может быть не любое вещественное число, а лишь неотрицательное. Можно ли использовать при составлении этого алгоритма тип **вещ**? Не нужен ли специальный тип **неотрвещ** (неотрицательное вещественное число) или что-то в этом роде?

Тип **вещ** использовать можно. Дело в том, что мы не предполагаем, что допустимы все исходные данные, относящиеся к указанному в алгоритме типу. Если бы это было не так, то алгоритм вычисления, например, значения выражения $1/(x^2-x-1)$ требовал бы специального типа «вещественное число, не равное $(1+\sqrt{5})/2$ и $(1-\sqrt{5})/2$» — ведь эти числа обращают знаменатель в нуль!

3. Если не предполагается, что все значения указанного в алгоритме типа допустимы как исходные данные, то зачем нужен тип **нат**? Ведь натуральные числа — часть целых, и любое данное типа **нат** можно считать данным типа **цел**. И зачем нужен тип **цел**, если целые числа — часть вещественных?

Ответ нужно начать с повторения уже сказанного: необходимость того или иного типа — вопрос удобства. Авторы УП сочли, что полезно ввести специальный тип «натуральные числа», поскольку ситуация, в которой какая-то величина может принимать только натуральные значения, часто встречается. Это решение, однако, оказалось спорным: в школьном курсе математики 0 не считается натуральным числом, поэтому тип **нат** оказывается малоприменимым. Уже в самом первом примере заголовка алгоритма (с. 31 УП) тип **нат** употреблен ошибочно: ведь и частное, и остаток при делении натуральных чисел могут быть равны нулю. Поэтому мы рекомендуем вообще отказаться от употребления типа **нат**.

Иначе обстоит дело с типами **цел** и **вещ**. Дело в том, что вещественные числа представляются в машине лишь приближенно (это неизбежно, так как вещественных чисел слишком много). Всякий, кто пробовал пользоваться калькулятором, знаком с неожиданностями типа $(1/3)\cdot 3=0.99999$, $10000+0.00001=10000$ и т. д.

Подобные «сюрпризы» могут нарушать работу составленных нами алгоритмов. Естественно желание гарантировать точность представления чисел и операций с ними, а в случае целых чисел это возможно. Поэтому, описывая аргумент алгоритма как **цел**, мы не только обязуемся давать исполнителю в качестве аргументов лишь целые числа, но и требуем от него точного выполнения операций над ними.

4. Зачем указывать тип заранее? Почему бы не разрешить одной и той же величине в одном случае принимать числовое значение, в другом — литерное и т. п.?

Такой подход действительно возможен и используется, например в языке программирования Рапира. Он имеет как преимущества, так и недостатки. Решение, принятое в УП (каждая величина имеет фиксированный тип), можно обосновать следующим образом. Алгоритмы не только записывают на алгоритмическом языке, но и читают, а при этом бывает полезно знать, значения какого типа может принимать данная величина. Если типы величин задаются заранее, то узнать это просто, если же типы величин могут меняться, то эта задача сильно усложняется. Кроме того, в случаях, когда важны быстрота выполнения программ и занимаемый ими объем памяти, предварительное указание типов дает возможность хранить их в памяти более экономно и выполнять операции с ними быстрее; при этом величины типа **цел**, как правило, занимают меньше места, чем типа **вещ**, и быстрее обрабатываются.

В завершение следует сказать, что все эти тонкости, скорее всего, не будут замечены школьниками. Это и хорошо — при первом знакомстве с алгоритмическим языком встают более насущные вопросы.

Исполнение алгоритмов

В УП предлагается изображать исполнение алгоритмов с помощью таблиц значений. Мы предложим другой способ, быть может, более удобный при работе в классе с доской и мелом. Разберем исполнение алгоритма примера 7.1 УП (решение квадратного уравнения). В нем есть такие величины:

аргументы a, b, c;

результаты x_1, x_2, y;

промежуточная величина D.

Для каждой из них нарисуем на доске рамку, внутри которой будем писать значение величины, а снаружи — имя. Как выглядит доска, подготовленная для разбора этого примера, изображено на рис. 1.

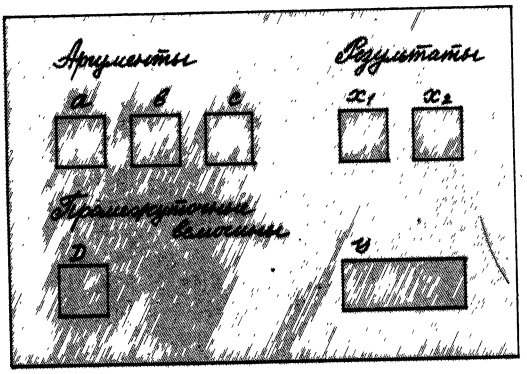
До начала исполнения алгоритма исполнителю сообщаются значения коэффициентов a, b, c. Пусть $a=2$, $b=1$, $c=-6$. Они записываются на доске (рис. 2). Теперь приступаем к выполнению алгоритма. Первая команда ($D:=b^2-4ac$) предписывает записать в рамку с надписью D значение выражения b^2-4ac при данных a, b и c; оно равно 49. Затем надо проверить условие «$D<0$». У нас $D=49$, условие не соблюдается. Следовательно, необходимо выполнить серию команд, идущих в команде ветвления за словом иначе. В рамке «y» пишем «есть решения», а в рамки «x_1» и «x_2» записываем корни уравнения, вычисленные по указанным в алгоритме формулам. На этом исполнение алгоритма заканчивается (рис. 3).

При следующем исполнении алгоритма (с другими аргументами) рамки нужно оставить без изменения, а надписи в них стереть.

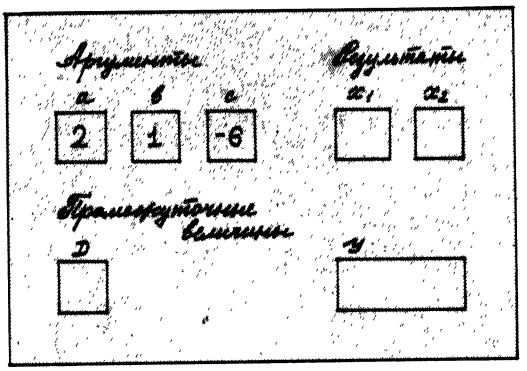
Некоторые рамки могут остаться пустыми. Например, при $D<0$ рамки «x_1» и «x_2» не заполняются. В этом случае говорят, что значения соответствующих величин — неопределенные.

Следующий пример (8.1 из УП) показывает, что значения величин (числа в рамках) могут многократно меняться в ходе исполнения алгоритма.

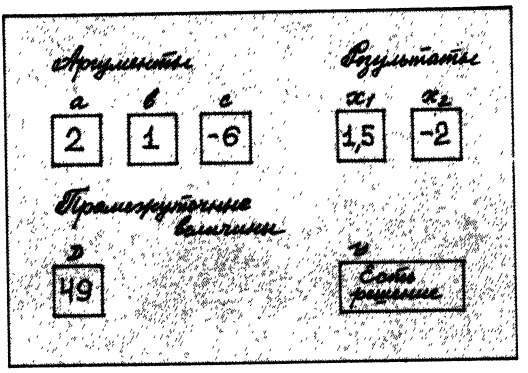
Подготовим доску к его исполнению (рис. 4). Заносим исходные данные ($M=35$, $N=21$), затем исполняем команды $x:=M$; $y:=N$ (числа из рамок «M», «N» переписываем в рамки «x», «y») (рис. 5). Далее выполняется команда повторения



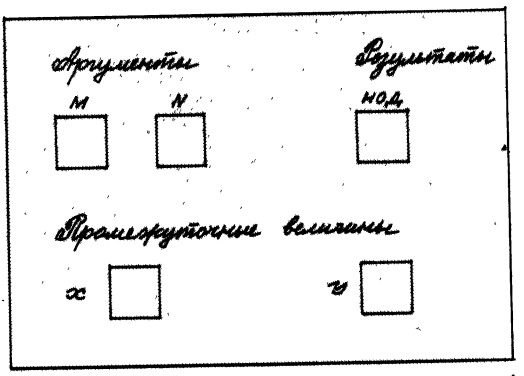
1



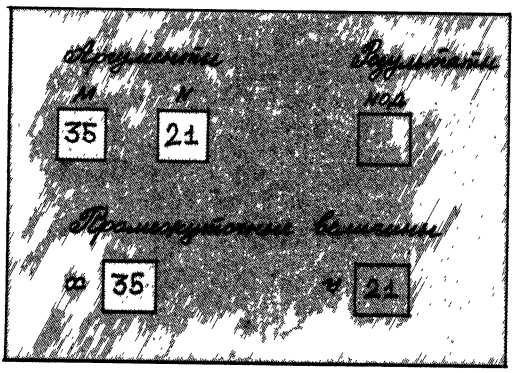
2



3



4



5

```
пока x≠y
нц
    если x>y
      то x:=x—y
      иначе y:=y—x
    все
кц
```

При описании ее исполнения будем рисовать только часть доски, содержащую значения переменных x и y.

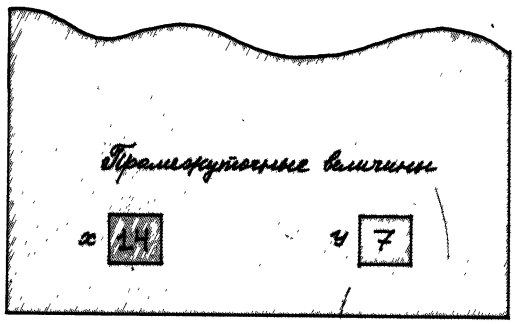
Условие $x \neq y$ соблюдается, выполняется серия команд от **нц** до **кц**, в данном случае одна команда — ветвления. В ней условие $x > y$ соблюдается, поэтому выполняется серия после **то**: $x := x - y$ (рис. 6). И снова $x \neq y$, но теперь $x < y$, и поэтому выполняется присваивание $y := y - x$ (рис. 7). Теперь $x \neq y$, $x > y$; выполняется команда $x := x - y$ (рис. 8), после этого $x = y$. Условие в команде повторения не соблюдается, ее выполнение заканчивается. Остается исполнить последнюю команду алгоритма (НОД$:= x$) и закончить исполнение алгоритма Евклида (рис. 9).

Как мы видели, в ходе исполнения алгоритма значения промежуточных величин могут меняться. А могут ли меняться значения аргументов и результатов? В УП об этом ничего не сказано. Логичнее всего запретить алгоритму менять значения аргументов и разрешить менять значения результатов. Заметим, что в программе Е-86 («Е-практикум») изменение значений аргументов не рассматривается как ошибка в алгоритме, но может повлечь различные неожиданные и странные явления, если этот алгоритм используется как вспомогательный.

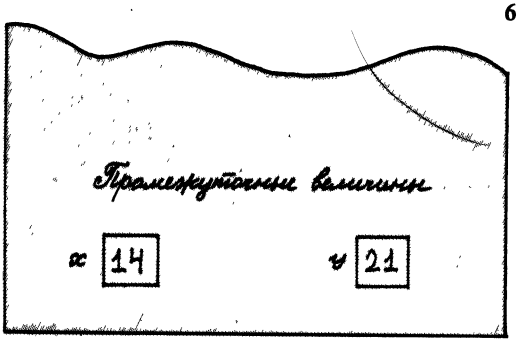
Команда присваивания. Предлагаем несколько упражнений, позволяющих лучше понять механизм ее работы.

1. Значение величины x равнялось 3. Каким оно будет после выполнения команды присваивания $x := 5$?

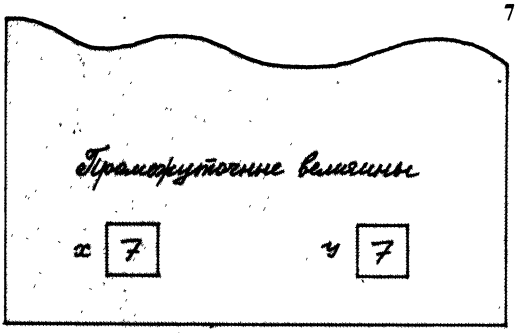
Ответ: 5.



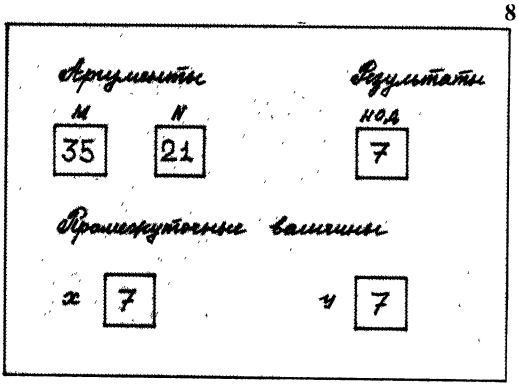
6



7



8



9

2. Значение величины x равнялось 3. Каким оно будет после выполнения команды присваивания $x := x + 5$?

Ответ: 8.

3. Каким было значение величины x, если после выполнения команды присваивания $x:=x+5$ оно стало равным 81?

Ответ: 76.

4. Каким было значение величины x, если после выполнения команды присваивания $x:=6$ оно стало равным 9?

Ответ. Описанная ситуация невозможна. После выполнения команды присваивания $x:=6$ значение величины x может быть равным только 6.

5. Каким могло быть значение величины x, если после выполнения команды присваивания $x:=-x$ оно стало положительным?

Ответ. Значение величины x могло быть равно любому отрицательному числу.

6. Каким могло быть значение величины x, если после выполнения команды $x:=x-2$ оно стало отрицательным?

Ответ. Значение величины x было меньше 2.

7. Какие значения будут иметь величины x и y после выполнения команды $x:=y$, если до этого было $x=4$, $y=6$?

Ответ. $x=6$, $y=6$.

8. Какие значения имели величины x и y до выполнения команды присваивания $x:=y$, если после ее выполнения $x=3$, $y=5$?

Ответ. Описанная ситуация невозможна. После выполнения команды присваивания $x:=y$ значения величин x и y становятся равными.

9. Какие значения имели величины x и y до выполнения команды присваивания $x:=y$, если после ее выполнения $x=3$, $y=3$?

Ответ. Значение величины y было равно 3. Чему было равно значение величины x, установить невозможно, так как в результате выполнения команды присваивания оно утрачено («затерто» новым значением).

10. Какие значения имели величины x и y до выполнения команды присваивания $x:=x+y$, если после ее выполнения $x=2$, $y=3$?

Решение. Величина y не менялась, следовательно, ее значение было равно 3. Значение величины x увеличилось на 3 и стало равно 2. Значит, оно было равным -1.

Ответ: $x=-1$, $y=3$.

11. При составлении алгоритма Васе понадобилось поменять местами значения величин x и y, т. е. сделать так, чтобы новое значение x равнялось старому значению y, а новое значение y равнялось старому значению x. Он написал так:

$x:=y$

$y:=x$

Правильно ли это?

Решение. Нет. После выполнения первой команды старое значение величины x безвозвратно теряется. Например, если до начала исполнения было $x=5$, $y=3$, то после первого присваивания будет $x=3$, $y=3$, и вторая команда ничего не изменит.

12. Как правильно решить предыдущую задачу (разрешается использовать дополнительную переменную величину t).

Решение. Можно написать

$t:=x$

$x:=y$

$y:=t$

После первого присваивания старое значение x будет «запомнено» в t. Затем величина x получит новое значение, равное старому значению величины y. Наконец, величина y получит значение, равное старому значению величины x, сохраненному в t.

Приведем в заключение две более трудные задачи.

13. Решить задачу 12 без использования дополнительной переменной величины.

Ответ. $x:=x+y$; $y:=x-y$; $x:=x-y$.

14. Значения величин x и y равнялись соответственно 7 и 17. Была выполнена последовательность команд присваивания, каждая из которых совпадала с одной из четырех:

$x:=x+y$

$x:=x-y$

$y:=y+x$

$y:=y-x$

Могло ли после этого получиться: а) $x=39$, $y=15$; б) $x=1$, $y=48$?

Указание. Первое невозможно, так как в результате выполнения любой из приведенных команд наибольший общий делитель чисел x и y не меняется. Второе возможно: нужно применить к паре значений (7, 17) последовательность действий алгоритма Евклида, переводящую ее в пару (1, 1), после чего получить пару (1, 48) уже легко.

Примеры алгоритмов работы с величинами

1. Алгоритм сложения дробей.

алг сумма (**цел** числ1, знам1, числ2, знам2, числ, знам)

арг числ1, знам1, числ2, знам2

рез числ, знам

нач

числ:=числ1×знам2+числ2×знам1

знам:=знам1×знам2

кон

После его исполнения (числ/знам) = =(числ1/знам1)+(числ2/знам2).

2. Алгоритм деления с остатком положительных целых чисел.

алг деление (**цел** делимое, делитель, частное, остаток)

```
    арг  делимое, делитель
    рез  частное, остаток
нач
    частное:=0
    остаток:=делимое
    пока остаток⩾делитель
    нц
      остаток:=остаток—делитель
      частное:=частное+1
    кц
кон
```

На каждом шаге этого алгоритма делимое=делитель×частное+остаток, остаток⩾⩾0, а после завершения команды повторения остаток меньше делителя.

3. Алгоритм возведения вещественного числа в положительную целую степень.

```
алг положительная степень (вещ a, цел n,
вещ x)
    арг a, n
    рез x
нач цел k
    k:=1; x:=a
    пока k≠n
    нц
      k:=k+1
      x:=x×a
    кц
кон
```

Здесь поддерживается выполнение соотношения $a^k=x$, для чего выбираются соответствующие начальные значения k и x, а затем при увеличении k соответствующим образом меняется x. В конце выполнения команды повторения входящее в нее условие не соблюдается, т. е. $k=n$ и, следовательно, $x=a^n$. Это и есть результат исполнения алгоритма.

Приведенные примеры (как и примеры в УП) довольно просты. У лучших учеников при их изучении возникает ощущение, что вообще составление алгоритмов — дело нехитрое и не слишком интересное. Чтобы продемонстрировать, что оно может требовать изобретательности, полезно предложить им разобрать пример нетривиального алгоритма, скажем, более хитрого (и более эффективного — требующего меньшего числа действий) алгоритма возведения в целую положительную степень.

```
алг степень (вещ a, цел n, вещ x)
    арг a, n
    рез x
нач вещ t, цел k
    x:=1; t:=a, k:=n
    пока k≠0
    нц
      если k четно
          то
            k:=k/2; t:=t²
          иначе
            k:=k—1; x:=x×t
      все
    кц
кон
```

Ключом к пониманию этого алгоритма является такое замечание: после любого числа циклов выполняется соотношение $a^n=xt^k$. В самом деле, деление k на 2 компенсируется возведением t в квадрат, а уменьшение k на единицу компенсируется умножением x на t. В конце, когда $k=0$, из указанного соотношения вытекает, что $x=a^n$, что и требовалось.

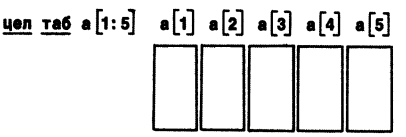
Табличные величины

С помощью команды повторения можно «спрятать» в коротком алгоритме большой объем действий. Табличные величины играют аналогичную роль по отношению к данным: с их помощью можно составить короткий алгоритм, работающий с огромным объемом информации.

Значением табличной величины является не одно число, а целая таблица, заполненная числами. Изображая ее на доске, мы рисуем рамку, разделенную на части (рис. 10). Вообще говоря, в клетках одной таблицы можно хранить совершенно разную по смыслу информацию — нужно только, чтобы типы хранимых величин были одинаковы, но обычно в таблице хранят однородную информацию.

Некоторые вопросы, связанные с табличными величинами, плохо продуманы авторами УП. Прежде всего это относится к таблицам с переменными границами (типа цел таб A[1:n] или вещ таб x[m:n]). Пусть аргументом алгоритма является таблица с переменными границами. Нужно ли перечислять эти границы отдельно в списке аргументов? Могут ли границы быть выражениями или только переменными? Что будет, если сообщенные при исполнении алгоритма значения границ не совпадают с фактическими границами таблицы?

Чтобы не вдаваться в обсуждение этих вопросов, мы предлагаем ограничиться таблицами, границы которых постоянны и заданы явно при их описании. Кроме того,



ограничимся линейными таблицами, нумерация в которых начинается с единицы (нижняя граница равна 1). Тогда слова «*i*-й элемент таблицы» можно будет употреблять, не опасаясь двусмысленности: ведь если нижняя граница таблицы не равна 1, то *i*-й по порядку элемент таблицы и элемент с номером (индексом) *i* — разные элементы!

Поиск заданного элемента в таблице. Эта задача, с которой начинается § 6 УП, довольно сложна. Целесообразно вначале рассмотреть более простой ее вариант, когда нужный элемент заведомо имеется в таблице. Если при его разборе в классе возникнут трудности, то от более сложного варианта придется отказаться.

Идею алгоритма можно пояснить следующей аналогией. Пусть где-то на дороге лежит потерянная галоша, которую надо найти. Как это сделать? Идти по дороге и смотреть себе под ноги, пока галоша не обнаружится. Условная запись на алгоритмическом языке будет такой:


```
пока галоши нет под ногами
нц
    шаг вперед
кц
```

Теперь переведем сказанное на язык таблиц. Пусть нам нужно найти в таблице вещ таб ключ [1:1000] элемент, равный заданному числу L (предполагается, что такой элемент в таблице есть). Представим ее себе как дорогу, по которой мы будем идти от начала (1) к концу (1000). Элемент, равный L, и будет искомой галошей. Номер клетки таблицы, в которой мы находимся, обозначим через *k*.

Условие «галоши нет под ногами» переводится на язык таблиц как ключ[*k*]≠L. Команда «шаг вперед» переводится как *k*:=*k*+1. Получается алгоритм

```
алг  поиск (вещ таб ключ [1:1000],
вещ L, цел k)
   арг ключ[1:1000), L
   рез k
нач
   k:=1
   пока ключ[k]≠L
   нц
      k:=k+1
   кц
кон
```

После выполнения команды повторения входящее в нее условие (в данном случае ключ[*k*]≠L) не соблюдается, т. е. ключ[*k*]=L; нужный элемент найден.

А что будет, если число L не содержится в таблице? Условие в команде повторения будет соблюдаться, а величина *k* увеличиваться, пока не станет равной 1001. В этот момент при проверке условия произойдет «авария», так как элемента ключ[1001] в таблице нет. Таким образом, если мы хотим предусмотреть и случай безрезультатного поиска, нужно изменить алгоритм. Один из вариантов такого изменения приведен в УП, мы предложим другой.

Прежде всего алгоритм должен иметь возможность сообщить, что искомый элемент в таблице отсутствует; добавим к числу результатов литерную величину «ответ», которая будет равна «найден» или «не найден» в зависимости от эффекта поиска. Введем также целочисленную величину *t*, которая равна числу «просмотренных» элементов таблицы. Между величинами *k*, ответ и *t* будет поддерживаться такая связь:

если L не встречается среди ключ[1], ...ключ[*t*], то ответ=«не найден»;

если встречается среди ключ[1],...ключ[*t*], то ответ=«найден», ключ[*k*]=L.

Начальные значения величин: *t*=0 (не просмотрено ни одного элемента таблицы), ответ=«не найден».

```
алг  поиск (вещ таб ключ[1:1000],
вещ L, цел к, лит ответ)
   арг ключ[1:1000], L
   рез k, ответ
нач цел t
   t:=0; ответ:=«не найден»
   пока t≠1000
   нц
      t:=t+1
      если ключ[t]=L
         то
            k:=t
            ответ:=«найден»
      все
   кц
кон
```

Замечания. 1. После исполнения команды *t*:=*t*+1 значение *t* становится равным номеру первого непросмотренного элемента таблицы.

2. После исполнения команды повторения условие *t*≠1000 не соблюдается, т. е. *t*=1000 (все элементы просмотрены).

3. Если элементы, равные L, встречаются в таблице несколько раз, значение *k* после исполнения алгоритма будет равно номеру наибольшего из этих элементов.

4. Алгоритм можно изменить, если после обнаружения первого элемента, равного L, дальнейший поиск не нужен. Для этого условие команды повторения надо сделать таким:

пока $t \neq 1000$ и ответ=«не найден»

Остальная часть алгоритма остается без изменений. Заодно можно избавиться от дополнительной переменной t: достаточно убрать из текста алгоритма **цел** t и $k:=t$, а в остальных местах заменить t на k.

Если число L встречается в таблице несколько раз, этот алгоритм находит минимальное k, для которого ключ$[k]$=L.

Поиск наименьшего элемента в таблице. Аргументом этого алгоритма будет вещественная таблица **вещ таб** $a[1{:}1000]$. Результатом должен быть номер наименьшего элемента в таблице, т. е. такое число l, что $a[l] \leqslant a[1], \ldots, a[1000]$ (заметим сразу же, что таких элементов может быть несколько).

Идея алгоритма состоит в следующем. Пусть мы уже знаем, что наименьшим элементом таблицы $a[1],\ldots,a[n-1]$ является $a[i]$. Добавим к таблице еще один элемент $a[n]$. Что можно сказать о наименьшем элементе таблицы $a[1],\ldots,a[n]$?

Тут возможны два варианта. Если $a[n] < a[i]$, т. е. добавленный элемент меньше минимального из старых элементов, то он будет минимальным в новой таблице. Если же нет ($a[n] \geqslant a[i]$), то $a[i]$ по-прежнему будет минимальным.

Получаем такой алгоритм:

```
алг МИНЭЛЕМЕНТ (вещ таб a[1:1000],
цел l)
  арг a
  рез l
нач цел i, n
  i:=1; n:=1
  пока n≠1000
  нц
    n:=n+1
    если a[n]<a[i]
      то
        i:=n
    все
  кц
  l:=i
кон
```

В начале работы алгоритма $n=i=1$ (в таблице из одного элемента этот элемент минимален). После любого числа циклов i является номером элемента, минимального среди $a[1],\ldots,a[n]$. Команда $n:=n+1$ нарушает это соотношение: после ее выполнения можно лишь утверждать, что i — номер элемента, минимального среди $a[1],\ldots, a[n-1]$ (ведь n увеличилось). Следующая за ней команда ветвления восстанавливает нарушенное соотношение. После завершения выполнения команды повторения входящее в нее условие не соблюдается, $n=1000$. Следовательно, $a[i]$ является минимальным элементом всей таблицы. Остается лишь присвоить результату алгоритма (величине l) значение величины i. Можно было бы и не использовать промежуточную величину i, употребляя вместо нее l и исключив команду $l:=i$.

В УП эта задача рассматривается для таблиц с переменными границами. Во избежание путаницы мы заменили переменные границы на постоянные (правда, теперь для использования этого алгоритма как вспомогательного при решении задачи сортировки нам понадобится вернуться к нему и несколько изменить).

Продолжение следует

Я. ЗАЙДЕЛЬМАН
г. Уфа

Язык программирования

Цели изучения языка программирования сформулированы в книге для учителя: «первая — показать школьникам, как конструкции алгоритмического языка могут быть выражены средствами языка программирования (иллюстративная цель); вторая — дать учащимся возможность проработать на ЭВМ... алгоритмы, которые они освоили на уроках (прикладная цель)». Этим определяются ответы на два важнейших вопроса: какой язык изучать и как построить его изучение?

Прикладная цель диктует: изучать надо язык, имеющийся на доступной вычислительной технике; желательно, чтобы по форме и идеям он был как можно ближе к уже изученному алгоритмическому.

Иллюстративная цель определяет методику изучения: язык программирования рассматривается как одна из форм записи алгоритмов, изложение ведется в постоянном сопоставлении с алгоритмическим языком. Необходимо выделить известные по алгоритмическому языку элементы, показать об-

Инф. и об 6/87

Методика обучения [redacted]

А. ШЕНЬ

Информатика в IX классе

Вспомогательные алгоритмы и их использование

Знакомство со вспомогательными алгоритмами проще всего начать с примеров, связанных с рисованием на плоскости. В предыдущих номерах журнала мы говорили об исполнителе Черепашке и составляли программу рисования слова МИР. В ней естественно выделялись части, соответствующие рисованию букв М, И, Р. Использование вспомогательных алгоритмов позволяет «узаконить» это разделение, явно отразив его в самом тексте алгоритма.

```
алг написание слова МИР
нач
    буква М
    шаг вправо
    буква И
    шаг вправо
    буква Р
кон
алг буква М
нач
    вперед (4)
    налево (30)
    назад (2)
    направо (60)
    вперед (2)
    налево (30)
    назад (4)
кон
алг буква И
нач
    вперед (4)
    назад (4)
    направо (30)
    вперед (4.8)*
    налево (30)
    назад (4)
кон
алг буква Р
нач
    вперед (4)
    направо (90)
    вперед (2)
    направо (90)
    вперед (20)
    направо (90)
    вперед (2)
кон
алг шаг вправо
нач
    не рисуй
    направо (90)
    вперед (1)
    налево (90)
    рисуй
кон
```

Смысл алгоритмов, использующих другие алгоритмы (называемые вспомогательными), может быть определен различными способами (однако в рассмотренном примере между всеми этими способами не проявляются различия; к трудностям, возникающим в более сложных примерах, мы еще вернемся).

Способ 1. Текстовая подстановка. Можно считать, что перед исполнением алгоритма, содержащего ссылки на вспомогательные алгоритмы (такие ссылки называют «вызовами» вспомогательных алгоритмов), эти ссылки «раскрываются». Это значит,

---

* *Продолжение*. Начало см. Информатика и образование. 1987. №№ 3—5.

* Мы пишем «4.8» вместо «4,8», так как запятая используется в алгоритмическом языке для разделения аргументов вспомогательного алгоритма.

что строка с названием вспомогательного алгоритма заменяется его текстом. Если мы проделаем это в нашем примере, то получим тот самый алгоритм рисования слова МИР, который приведен в предыдущих номерах журнала.

Способ 2. Разные исполнители. Можно представить себе дело иначе. Вообразим, что у нас имеется Помощник, в обязанности которого входит исполнение вспомогательных алгоритмов (в данном примере — алгоритмов «буква М», «буква И», «буква Р», «шаг вправо»). Мы можем, например, скомандовать ему «буква М», после чего Помощник сам, без нашего участия, командует Черепашкой в соответствии с алгоритмом рисования буквы М. После этого Помощник сообщает нам, что команда выполнена и он готов к исполнению следующей команды. При этом Помощник знает лишь алгоритмы рисования букв «М», «И», «Р» и алгоритм «шаг вправо», ему безразлично, как они используются в основном алгоритме: ему командуют — он выполняет. Наоборот, мы, командуя Помощником, можем уже не интересоваться деталями рисования букв.

Способ 3. Расширение системы команд. Будем считать, что наша Черепашка может учиться. Это значит, что ее систему команд можно расширять, обучая ее новым командам. Обучение состоит в том, что мы сообщаем ей алгоритм выполнения соответствующей команды. После этого она добавляет эту команду в свою систему команд. Так, сообщив Черепашке алгоритмы «буква М», «буква И», «буква Р» и «шаг вправо», мы получаем нового исполнителя — Обученную Черепашку с системой команд: вперед (x), назад (x), налево (x), направо (x), рисуй, не рисуй, буква М, буква И, буква Р, шаг вправо. После этого можно исполнять основной алгоритм «написание слова МИР» обычным образом — все его команды входят в систему команд исполнителя Обученная Черепашка.

Все сказанное, конечно, лишь наглядные образы, иллюстрирующие исполнение вспомогательных алгоритмов. Для изложения в классе можно выбрать тот, который кажется наиболее наглядным.

Зачем нужны вспомогательные алгоритмы? Пример использования вспомогательных алгоритмов при рисовании слова МИР вызывает естественный вопрос: а зачем все это нужно? Почему нельзя сразу вписать алгоритмы рисования букв и «шаг вправо» внутрь основного алгоритма? Что мы выигрываем, выделяя их? Ответить на это можно следующее.

1. Прежде всего, выделение вспомогательных алгоритмов облегчает составление и проверку алгоритмов; при составлении основного алгоритма (в нашем примере — алгоритма рисования слова МИР) мы можем уже не заботиться о том, как работают вспомогательные алгоритмы, а нужно знать только, что они делают. Наоборот, при составлении вспомогательного алгоритма (в нашем примере рисование букв, «шаг вправо») мы не должны учитывать, где и зачем он используется. Нужно лишь следить за тем, чтобы он действительно делал то, для чего предназначен. Благодаря этому разделению основной и вспомогательный алгоритмы могут составлять даже разные люди — им только следует договориться о том, что должен делать вспомогательный алгоритм (но не о том, как он будет это делать).

2. Использование вспомогательных алгоритмов позволяет сократить решение задачи. В самом деле, при «раскрытии» вызовов вспомогательного алгоритма «шаг вправо» его текст придется писать дважды. Правда, в нашем случае эта экономия компенсируется «накладными расходами» на оформление (строки **алг, нач, кон**). Однако в других случаях использование вспомогательных алгоритмов может привести к существенному упрощению — это бывает, когда один и тот же вспомогательный алгоритм используется многократно (если бы, например, нужно было написать слово МИМ, то экономия была бы больше, так как в этом случае алгоритм «буква М» используется дважды).

Такой прием сокращения длины текста не нов. По существу он же применяется, например, в книгах по математике, где формулы и теоремы нумеруют или снабжают именами, а затем пишут «формула (7)» или «теорема Пифагора» вместо того, чтобы повторять формулу или формулировку теоремы. Такие записи аналогичны обращению к вспомогательным алгоритмам.

Заметим кстати, что уменьшение длины алгоритма обычно позволяет уменьшить объем памяти ЭВМ, занимаемый им.

3. Использование вспомогательных алгоритмов полезно, если нам нужно решать несколько похожих задач. Например, если после слова МИР нам потребуется составить алгоритм написания слова РИМ, то все вспомогательные алгоритмы можно оставить без изменений, изменив основной алгоритм:

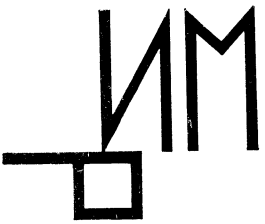
```
алг написание слова РИМ
нач
    буква Р
    шаг вправо
    буква И
    шаг вправо
```

```
буква М
кон
```

Прежде чем читать дальше, остановитесь и подумайте: правилен ли этот алгоритм?

Оказывается, нет. Результат его работы изображен на рис. 1. Вспомним: после рисования буквы Р Черепашка остается в середине буквы, головой влево. После этого буквы И и М она будет рисовать повернутыми. Как же исправить ошибку?

1

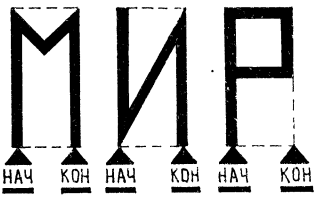


Сформулируем более точно: как составить алгоритмы рисования букв так, чтобы из них можно было составлять алгоритмы рисования любых слов, чередуя рисование букв с командой «шаг вправо»?

Ответ таков: надо договориться о том, где находится Черепашка в начале и конце рисования каждой буквы. Будем считать, что каждая буква вписана в прямоугольник (рис. 2), начальное положение Черепашки — головой вверх в левом нижнем углу, а конечное — головой вверх в правом нижнем углу. Алгоритмы «буква М» и «буква И» удовлетворяют этому требованию (если быть точным, то в алгоритме «буква И» следует заменить 4.8 на $8\sqrt{3}$ — проверьте!), а вот алгоритм «буква Р» требует изменений. Сделаем их.

```
алг буква Р
нач
    вперед (4)
    направо (90)
    вперед (2)
    направо (90)
    вперед (2)
    направо (90)
    вперед (2)
    назад (2)
    не рисуй
    направо (90)
    назад (2)
кон
```

2

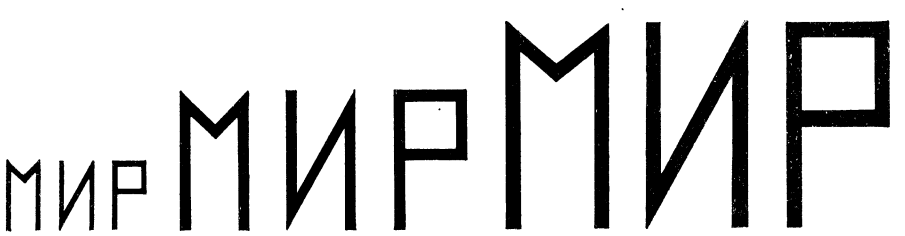


Заметим, что в наших требованиях к алгоритмам рисования букв ничего не сказано о том, рисует или не рисует Черепашка после их исполнения. В частности, после исполнения алгоритма «буква Р» Черепашка не рисует, а после исполнения остальных алгоритмов — рисует. Это не страшно (мы считаем, что повторное выполнение команды «не рисуй» допустимо).

Ошибка, допущенная при составлении алгоритма рисования слова РИМ, довольно типична. Она произошла из-за того, что при разделении задачи (рисование слова) на подзадачи (рисование букв) не были достаточно точно определены эти подзадачи: мы договорились о том, какие буквы должны рисовать вспомогательные алгоритмы, но не о том, где должна быть Черепашка до и после исполнения. Программисты сказали бы по этому поводу так: «спецификации алгоритмов рисования букв были недостаточно точными».

Вспомогательные алгоритмы с аргументами. Использование аргументов во вспомогательных алгоритмах можно пояснить на том же примере рисования слова МИР. Представим себе, что мы хотим это слово несколько раз подряд нарисовать с разным размером букв (рис. 3). Нужно ли составлять для этого три вспомогательных алгоритма рисования каждой буквы (для маленького, среднего и большого размеров)? Оказывается, нет. Вместо этого можно составить алгоритмы рисования букв в заданном масштабе d (число d показывает, во сколько раз буква крупнее, чем в приведенном выше алгоритме; так, $d=1/3$ означает, что все размеры буквы уменьшены в три раза). Заголовок алгоритма рисования буквы М будет выглядеть так:

3



```
алг буква М (вещ d)
арг d
```
(аналогично для других букв). Составим теперь основной алгоритм.
```
алг трижды МИР
нач
    буква М (0.25)
    шаг вправо
    буква И (0.25)
    шаг вправо
    буква Р (0.25)
    шаг вправо
    буква М (0.5)
    шаг вправо
    буква И (0.5)
    шаг вправо
    буква Р (0.5)
    шаг вправо
    буква М (1)
    шаг вправо
    буква И (1)
    шаг вправо
    буква Р (1)
кон
```
Выпишем теперь использованный здесь вспомогательный алгоритм рисования буквы М.
```
алг буква М (вещ d)
арг d
нач
    вперед (4*d)
    налево (30)
    назад (2*d)
    направо (60)
    вперед (2*d)
    налево (30)
    назад (4*d)
кон
```
(звездочка — знак умножения). Аналогично выглядят и алгоритмы рисования букв И и Р.

Упражнение. В построенном алгоритме интервалы между буквами не зависят от их размера. Как сделать так, чтобы между маленькими буквами были маленькие интервалы, а между большими — большие?

Проиллюстрируем теперь на этом примере некоторые способы исполнения вспомогательных алгоритмов. Если мы хотим перед исполнением основного алгоритма «раскрыть» вызовы вспомогательных алгоримов, то не следует переписывать их текст буквально (если это сделать, то в основном алгоритме будет фигурировать неизвестно чему равная величина d). Вместо этого при переписывании следует заменить букву d на число, стоящее в данном обращении к вспомогательному алгоритму. Так, строку «буква М (0.25)» надо заменить на
```
вперед (4*0.25)
налево (30)
назад (2*0.25)
направо (60)
вперед (2*0.25)
налево (30)
назад (4*0.25)
```
и т. д.

Другой способ — использование Помощника, исполняющего вспомогательные алгоритмы, — также несколько видоизменяется. В этом случае мы не просто командуем ему «буква М», но и сообщаем аргумент (к примеру, 0.25). После этого Помощник присваивает величине d значение 0.25 и приступает к исполнению алгоритма «буква М». Видя строку «вперед (4*d)», он умножает 4 на 0.25, получает 1 и командует Черепашке «вперед (1)», и т. д.

Вспомогательные алгоритмы с аргументами и результатами. Рассмотрим пример из УП: нахождение большего из трех чисел. Вот основной алгоритм:
```
алг БИТ (вещ a, b, c, y)
арг a, b, c
рез y
нач вещ z
    БИД (a, b, z)
    БИД (z, c, y)
кон
```
Вспомогательный алгоритм:
```
алг БИД (вещ a, b, c)
арг a, b
рез c
нач
    если a≥b
      то c:=a
      иначе c:=b
    все
кон
```
Рассмотрим разные варианты их исполнения.

Если мы «раскроем» обращения к вспомогательному алгоритму внутри основного буквально, то получим такой текст:
```
нач вещ z
    если a≥b
      то c:=a
      иначе c:=b
    все
    если a≥b
      то c:=a
      иначе c:=b
    все
кон
```
Ясно, что он не имеет никакого смысла (тут, в частности, вообще не фигурирует z — результат основного алгоритма).

Как же «раскрыть» вызовы правильно? Нужно, как говорят, «подставить фактические параметры вместо формальных». Это значит следующее. В описании вспомогательного алгоритма имеются величины a, b, c. Они называются формальными параметрами. В обращении к нему фигурируют другие имена величин — в первом обращении написано БИД (a, b, z), а во втором БИД (z, c, y). Величины, входящие в обращение к вспомогательному алгоритму, называются фактическими параметрами. Для разных вызовов они могут быть разными. Перед «раскрытием» вызова вспомогательного алгоритма следует заменить в тексте вспомогательного алгоритма формальные параметры на соответствующие им фактические. В первом обращении соответствие между формальными и фактическими параметрами таково:

формальные параметры a, b, c;
фактические параметры a, b, z.

Таким образом, при раскрытии имена a и b остаются, а c следует заменить на z:

```
если a≥b
  то z:=a
  иначе z:=b
все
```

Во втором обращении соответствие таково:

формальные параметры a, b, c;
фактические параметры z, c, y;

замена такова: $a \to z$, $b \to c$, $c \to y$. Получаем:

```
если z≥c
  то y:=z
  иначе y:=c
все
```

Таким образом, после раскрытия вызовов вспомогательных алгоритмов получаем следующее:

```
алг БИТ (вещ a, b, c, y)
арг a, b, c
рез y
нач вещ z
    если a≥b
      то z:=a
      иначе z:=b
    все
    если z≥c
      то y:=z
      иначе y:=c
    все
кон
```

Проследив за работой этого алгоритма, мы убеждаемся, что он действительно помещает в y наибольшее из a, b, c: после исполнения первой команды ветвления z равно большему из чисел a и b, а после второй y равно большему из чисел z и c, т. е. большему из чисел a, b, c:

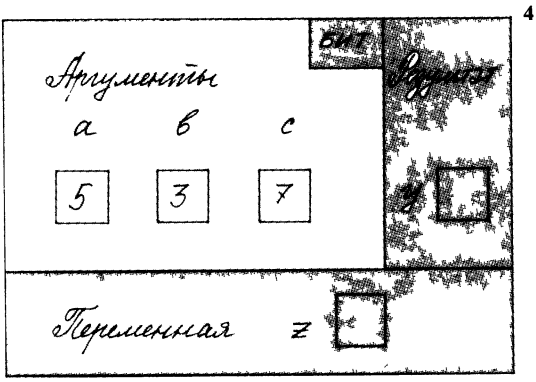
$$\max(a, b, c) = \max(\max(a, b), c) = \max(z, c)$$

Прежде чем перейти к другому способу исполнения, сделаем еще одно замечание. В данном случае во вспомогательном алгоритме не было промежуточных величин. Если же они есть, то случается, что их имена совпадают с именами величин, имеющихся в основном алгоритме. Чтобы из-за этого не возникла путаница, при подстановке текста вспомогательного алгоритма на место его вызова помимо замены формальных параметров на фактические необходимо предварительно изменить имена промежуточных величин вспомогательного алгоритма. Приведем соответствующий фрагмент из описания алгоритмического языка Алгол-60:

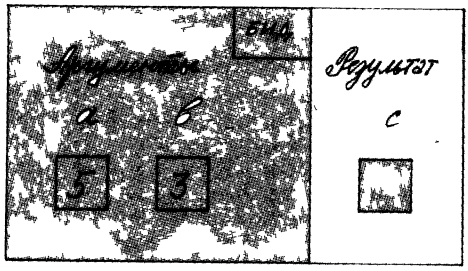
«Любой формальный параметр... повсюду в теле процедуры (вспомогательного алгоритма — *А. Ш.*) заменяется на соответствующий фактический параметр... Возможность противоречий между идентификаторами (именами величин — *А.Ш.*), вставляемыми в тело процедуры... и идентификаторами, уже присутствующими в теле процедуры... устраняется систематическими изменениями... идентификаторов, затронутых такими противоречиями» (мы пропустили в этой цитате части, которые непонятны без детального знакомства с Алголом-60).

Теперь опишем исполнение алгоритма БИТ, если алгоритм БИД исполняется отдельно. Представим себе учителя Тройкина, исполняющего алгоритм БИТ, и ученика Двойкина, умеющего исполнять алгоритм БИД.

Начиная работу, учитель Тройкин рисует на доске картинку (рис. 4). Будем рассматривать исполнение алгоритма для аргументов 5, 3, 7. Эти числа записаны в рамках для аргументов. Рамки результатов и промежуточной величины пусты. Тройкин приступает к работе. Первой командой, которую нужно исполнить, является вызов

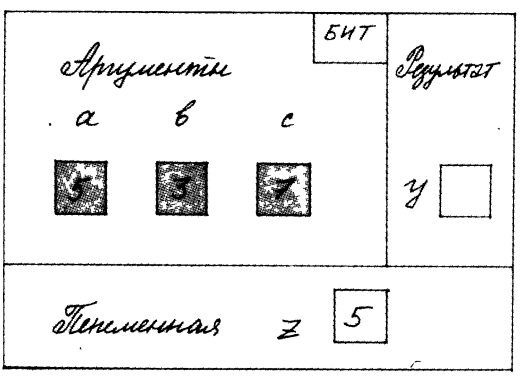


4

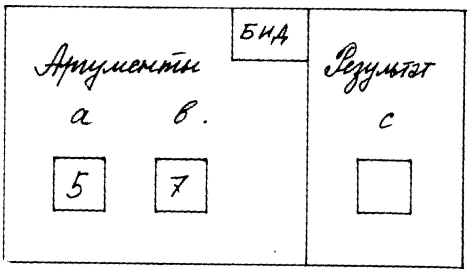


5

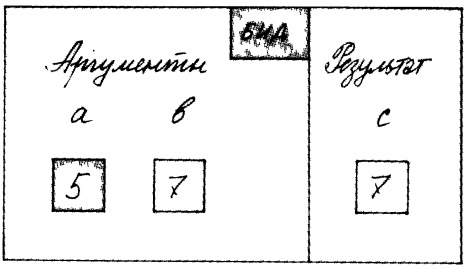
вспомогательного алгоритма БИД. И Тройкин командует: «Двойкин, к доске! Исполните алгоритм БИД для аргументов 5 и 3 и сообщите результат!» Двойкин выходит к доске (другой!) и принимается за работу (рис. 5). Обратите внимание, что рамки с надписями «*a*», «*b*», «*c*» на доске БИД не имеют ничего общего с рамками «*a*», «*b*», «*c*» на доске БИТ, кроме названия. В рамках «*a*», «*b*» доски БИД записаны числа, сообщенные Тройкиным Двойкину — числа 5 и 3 (они совпадают с записями в рамках «*a*» и «*b*» доски БИТ, но это случайность). Двойкин приступает к исполнению алгоритма БИД. Условие в команде ветвления ($a \geqslant b$) соблюдается, выполняется команда $c:=a$, записывающая в рамку *c* число 5. Двойкин говорит учителю: «Результат равен 5», стирает с доски и садится на место. Учитель записывает сообщенный Двойкиным результат в рамку *z*, на этом выполнение первой команды алгоритма БИТ заканчивается (рис. 6). Но Двойкину не удается спокойно посидеть на месте: следующая команда алгоритма БИТ — снова вызов вспомогательного алгоритма БИД. И Тройкин командует: «Двойкин, к доске! Исполните алгоритм БИД для аргументов 5 и 7 и сообщите результат!» Вызов БИД (*z*, *c*, *y*) происходит при $z=5$, $c=7$. Вызванный снова Двойкин рисует картинку (рис. 7), исполняет алгоритм (рис. 8), полученный результат сообщает Тройкину, стирает с доски и садится на место. Тройкин записывает его в рамку «*y*», и на этом



6



7



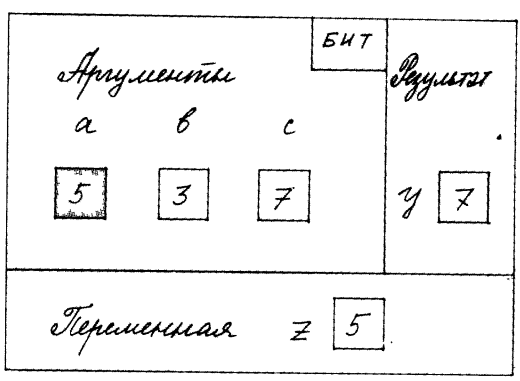
8

исполнение алгоритма БИТ кончается (рис. 9).

Оба разобранных варианта исполнения вспомогательных алгоритмов иллюстрируют такое свойство: имена величин, использованные во вспомогательном алгоритме, не имеют значения и могут быть выбраны произвольно; если бы мы написали

алг БИД (**вещ** α, β, γ)
арг α, β
рез γ
нач
  если $\alpha \geqslant \beta$
    то $\gamma := \alpha$
    иначе $\gamma := \beta$
  все
кон

(как это сделано в УП), в исполнении основного алгоритма ровным счетом ничего бы не изменилось. В частности, при выборе названий для величин вспомогательного алгоритма можно, как мы видели, использовать имена, используемые и в основном алгорит-



9

ме. При этом одинаково именованные величины основного и вспомогательного алгоритмов — совершенно разные переменные.

Трудные случаи использования вспомогательных алгоритмов. Приведем два примера, в которых возможно различное толкование смысла вспомогательных алгоритмов.

Пример 1. Пусть основной алгоритм содержит фрагмент

.....
i:=2
F(a[i],i)
.....

(i — целая переменная, a — целочисленная таблица), а вспомогательный алгоритм F таков:

алг F (**цел** p, q)
рез p, q
нач
  q:=3
  p:=7
кон

Если раскрыть вызов вспомогательного алгоритма так, как мы это делали, то получится
i:=2
i:=3
a[i]:=7
т. е. i станет равным 3, элементу a[3] будет присвоено значение 7.

С другой стороны, если мы рассмотрим вариант с двумя исполнителями, получится вот что. Исполнитель основного алгоритма делает i равным 2, затем командует «исполнить алгоритм F и сообщить результаты». Результаты будут равны 7 и 3. Исполнитель основного алгоритма выполняет присваивания a[i]:=7 (т. е. a[2]:=7, так как в этот момент i=2) и i:=3. Итак, i станет равным 3, а a[2] станет равным 7. Получается, что такой способ исполнения приводит к другим результатам.

Пример 2. Пусть основной алгоритм содержит строку F(a, a) (a — целая переменная), а вспомогательный алгоритм таков же, как и в предыдущем примере. Тогда при раскрытии вызова получим
a:=3
a:=7
и окончательное значение a будет равно 7.

Если же мы попытаемся проследить за исполнением алгоритма двумя исполнителями, то обнаружим, что исполнитель основного алгоритма говорит: «Исполнить алгоритм F и сообщить результаты», ему сообщают «7, 3», и теперь он стоит перед неразрешимой задачей: нужно в рамку «a» записать 7 и 3 одновременно.

А теперь изменим вспомогательный алгоритм, переставив в нем строки:

алг F1 (**цел** p, q)
рез p, q
нач
  p:=7
  q:=3
кон

Казалось бы, алгоритмы F и F1 с точки зрения результатов их работы неотличимы. Однако при раскрытии вызовов F (a, a) и F1 (a, a) мы получим:

для F	для F1
a:=3	a:=7
a:=7	a:=3

и это уже совсем не одно и то же.

После рассмотрения этих примеров у читателя может возникнуть вопрос: как же правильно исполнять вспомогательные алгоритмы? Какие результаты должны получаться «на самом деле»?

Однако вопрос этот сформулирован неверно. Смысл тех или иных конструкций алгоритмического языка определяется его разработчиками, так что правильная постановка вопроса такова: какой способ исполнения вспомогательных алгоритмов имели в виду разработчики? УП оставляет его открытым.

Все эти трудности, скорее всего, останутся не замеченными школьниками, тем не менее полезно иметь их в виду (хотя, конечно, не стоит касаться их на уроках, если школьники сами не обратят на них внимания).

Упорядочение линейной таблицы

Задача упорядочения линейной таблицы состоит в следующем: имеется целочисленная таблица a[1:1000]; надо ее упорядочить, т. е. переставить числа в таблице так, чтобы выполнялось $a[1] \leqslant a[2] \leqslant ... \leqslant a[1000]$ (отличие от формулировки задачи в УП состоит в том, что размер таблицы фиксирован и равен 1000; о причинах такого рода изменений мы говорили в предыдущих номерах журнала). Заметим, что в условии задачи нельзя заменить «$\leqslant$» на «$<$», так как в таблице a могут встречаться одинаковые элементы.

Для начала изложим решение, следующее схеме из УП (с изменениями, вызванными тем, что все таблицы имеют фиксированную длину).

Какой элемент должен стоять после упорядочения первым? Ясно, что минимальный. А вторым? Тоже ясно — минимальный из оставшихся (кроме первого). Это наводит на такую идею алгоритма: будем заполнять новую (упорядоченную) таблицу от начала к концу. Введем переменную k, равную числу уже поставленных на свои места эле-

ментов. Точнее, мы будем требовать соблюдения такого условия:

первые k элементов таблицы a уже стоят на своих местах (т. е. совпадают с первыми k элементами упорядоченной таблицы, которую мы хотим построить) (1)

Вначале $k=0$; постепенно k будет увеличиваться. Когда k станет равным 1000, упорядочение закончено; все элементы стоят на своих местах. Впрочем, упорядочение можно закончить при $k=999$: если первые 999 элементов стоят на своих местах, то и последний тоже — для него осталось единственное место.

Итак, схема алгоритма такова:

```
k:=0
пока k≠1000
нц
  увеличить k на 1, не нарушая условия (1)
кц
```


Осталось понять, как увеличить k на 1, не нарушая (1). Пусть первые k мест таблицы уже заполнены правильно, и нам надо, чтобы были заполнены правильно первые $k+1$ место. Другими словами, нужно поставить на $k+1$-е место нужный элемент. Какой? Наименьший среди элементов таблицы, не считая первых k. Запишем это.

```
k:=0
пока k≠1000
нц
  положить s равным номеру минимального
  элемента среди a[k+1], ..., a[1000]
  поменять местами k+1-й и s-й элементы
  k:=k+1
кц
```

Выделим два указанных действия во вспомогательные алгоритмы.

```
алг минэлемент (цел таб a[1:1000], k, s)
арг a, k
рез s
```

Он исполняется при $k<1000$; после его исполнения s равно номеру минимального элемента среди $a[k+1]$, ..., $a[1000]$.

```
алг обмен (цел таб a[1:1000], i, j)
арг a, i, j
рез a
```

Он исполняется при $1\leqslant i, j\leqslant 1000$; после его исполнения значения элементов $a[i]$ и $a[j]$ меняются местами; при $i=j$ таблица a не меняется.

Теперь можно составить основной алгоритм.

```
алг упорядочение (цел таб a[1:1000])
арг a
рез a
нач цел k, s
  k:=0
  пока k≠1000
  нц
    минэлемент (a, k, s)
    обмен (a, k+1, s)
  кц
кон
```

Теперь составим вспомогательные алгоритмы. Первый из них аналогичен приведенному в предыдущем номере журнала.

```
алг минэлемент (цел таб a[1:1000], k, s)
арг a, k
рез s
нач цел t
  t:=k+1
  s:=k+1
  пока t≠1000
  нц
    t:=t+1
    если a[t]<a[s]
      то s:=t
    все
  кц
кон
```

Идея второго алгоритма также нам уже знакома.

```
алг обмен (цел таб a[1:1000], i, j)
арг a, i, j
рез a
нач цел t
  t:=a[i]
  a[i]:=a[j]
  a[j]:=t
кон
```

Легко проверить, что при $i=j$ этот алгоритм работает правильно (не меняет таблицу a).

Задача решена. Отметим, что при ее решении мы использовали метод, который называют «методом последовательных уточнений», «пошаговой детализацией», «программированием сверху вниз» и т. д. Его суть в том, что мы сначала составляем основной алгоритм с использованием вспомогательных, не составляя самих вспомогательных алгоритмов, а лишь определяя, какие задачи они должны решать, и лишь потом переходим к составлению вспомогательных алгоритмов.

Заметим также, что в основном алгоритме (как мы уже говорили) можно заменить условие «$k\neq 1000$» на «$k\neq 999$».

Упорядочение линейной таблицы: другое решение. Это решение по своим характеристикам (длина алгоритма, скорость его работы и т. д.) весьма близко к уже изложенному. Тем не менее идея его поучи-

тельна. В качестве условия выберем такое:

первые k элементов таблицы стоят в возрастающем порядке: $a[1] \leqslant \ldots \leqslant a[k]$ (2)

Начальное значение k равно 1, конечное — 1000 (здесь уже 999 не годится!). Приходим к такой схеме алгоритма:

```
k:=1
пока k≠1000
нц
  увеличить k на 1, не нарушая условия (2)
кц
```

Как же увеличить k? Если $a[k] \leqslant a[k+1]$, то нам повезло и первые $k+1$ элементов случайно оказались в нужном порядке. Если же нет, то элемент $a[k+1]$ стоит правее, чем нам нужно и его надо сдвигать к началу таблицы, пока он не встанет на свое место. Другими словами, его надо менять местами с предыдущим элементом до тех пор, пока он не окажется больше или равен предыдущему элементу или пока он не попадет в начало таблицы. Обозначив через s «текущий» номер сдвигаемого элемента, получаем вот что:

```
алг упорядочение (цел таб a[1:1000])
арг a
рез a
нач цел k, s
  k:=1
  пока k≠1000
  нц
    s:=k+1
    пока s≠1 и a[s]<a[s—1]
    нц
      обмен (a, s—1, s)
      s:=s—1
    кц
    k:=k+1
  кц
кон
```

Алгоритм «обмен» уже приводился.

Здесь есть одна тонкость: что будет, если $s=1$ при проверке условия внутренней команды повторения? С одной стороны, условие $s \neq 1$ и $a[s]<a[s-1]$ вроде бы можно считать ложным, так как уже условие $s \neq 1$ ложно. С другой стороны, в условие входит часть $a[s]<a[s-1]$, которая при $s=1$ не имеет никакого смысла, так как $s-1=0$, а нулевого элемента в таблице нет. Поскольку способ проверки условий в УП не описан, можно придерживаться любой из приведенных точек зрения. Во втором случае можно видоизменить алгоритм, например, так:

```
нач цел k, s, лит готово
  k:=1
  пока k≠1000
  нц
    s:=k+1
    готово:=«нет»
  пока готово=«нет»
  нц
  если s=1
    то готово:=«да»
    иначе
      если a[s]<a[s—1]
        то
          обмен (a, s—1, s)
          s:=s—1
        иначе
          готово:=«да»
      все
    все
  кц
кон
```


Упорядочение таблицы: эффективный алгоритм. Приводимый ниже алгоритм упорядочения более сложен, чем разобранные, но и более эффективен. Можно убедиться, что используемый в нем метод позволяет упорядочить таблицу из n элементов за время, примерно пропорциональное $n \log n$, а время упорядочения этой же таблицы с помощью любого из предыдущих методов примерно пропорционально n^2 (мы говорим о «времени» работы алгоритма для краткости: следовало бы говорить о числе выполняемых действий — проверок условий и исполнения команд присваивания).

Идею эффективного алгоритма упорядочения (его называют «сортировка слиянием») можно пояснить наглядно следующим образом. Пусть у нас есть ящик с библиотечными карточками, которые нужно разложить по алфавиту. Сначала разложим все карточки в группы по одной карточке в каждой. Затем будем соединять группы попарно: из двух групп по одной карточке получится группа из двух карточек, которые мы расположим в алфавитном порядке. Снова соединим группы попарно, делая из двух групп по две карточки одну из четырех (в которой карточки также идут в алфавитном порядке). На следующих этапах мы получим группы по 8 карточек, затем по 16 и так далее до тех пор, пока все карточки не попадут в одну группу, внутри которой они будут идти в алфавитном порядке (если число карточек не является целой степенью двойки, то некоторые группы будут неполными).

При таком способе упорядочения важную роль играет процесс «слияния» двух упорядоченных стопок (групп) карточек в

одну. Его можно описать так:

пока хоть одна из исходных стопок непуста
нц
  выбор
    при обе стопки непусты:
      переложить ту из двух верхних карточек, которая идет по алфавиту раньше, в конец новой стопки
    при одна из стопок пуста:
      переложить верхнюю карточку другой стопки в конец новой стопки
  все
кц

Здесь использована команда выбора, описанная в УП для X класса; при желании ее можно заменить на команду ветвления.

Возвратимся теперь от карточек к задаче упорядочения таблицы. Схема алгоритма будет такова. Введем переменную k (длина упорядоченных участков) и будем требовать выполнения такого условия:

участки длины k, на которые разбивается таблица a (от начала к концу), упорядочены (3)

Например, при $k=2$ условие (3) означает: $a[1] \leqslant a[2]$, $a[3] \leqslant a[4]$, ...

Вначале $k=1$ (участки длины 1 упорядочены автоматически). В конце $k \geqslant 1000$, все элементы таблицы оказываются в одном участке. На каждом шаге слияния длина упорядоченных участков увеличивается вдвое. Итак:

$k:=1$
пока $k<1000$
нц

  увеличить k вдвое, не нарушая условия (3)

кц

Удвоение k без нарушения (3) требует слияния 1-го и 2-го участков, 3-го и 4-го и т. д. с сохранением порядка. Это слияние будет выполняться с помощью команды повторения. Введем переменную s, играющую роль границы между обработанной и необработанной частями: на отрезке $a[1]$, ... $a[s]$ упорядочены участки длиной $2k$, а на остальной части — длиной k. Вначале $s=0$. Схема удвоения k становится такой:

$s:=0$
пока $s<1000$
нц

  слить очередную пару участков, соответственно увеличив s

кц

Очередная пара участков — это $a[s+1]$... ...$a[s+k]$ и $a[s+k+1]...a[s+2k]$, если только мы не подошли уже к правой границе таблицы. Если же мы уже подошли к правой границе, то второй участок может быть короче или вообще отсутствовать. Запишем разные варианты действий в виде команды ветвления:

если $s+k+1 \leqslant 1000$ (второй участок непуст)
то
  слияние (a, s, $s+k$, min(1000, $s+2*k$))
  $s:=\min(1000, s+2*k)$
иначе (второй участок пуст)
  $s:=1000$
все

Здесь слияние (a, p, q, r) — вспомогательный алгоритм, работающий при $0 \leqslant p < q < r \leqslant 1000$ и соединяющий упорядоченные участки $a[p+1]...a[q]$ и $a[q+1]...a[r]$ в упорядоченный участок $a[p+1]...a[r]$.

Приведем теперь полный текст основного алгоритма.

алг сортировка слиянием (**цел таб** a [1:1000])
арг a
рез a
нач цел k, s
  $k:=1$
  пока $k<1000$
  нц
    $s:=0$
    пока $s<1000$
    нц
      если $s+k+1 \leqslant 1000$
      то
        слияние (a, s, $s+k$, min (1000, $s+2*k$))
        $s:=\min(1000, s+2*k)$
      иначе
        $s:=1000$
      все
    кц
    $k:=2*k$
  кц
кон

Осталось составить алгоритм слияния. Мы будем следовать описанной схеме с карточками. Введем переменные $i1$ и $i2$ — число карточек, взятых из первой и второй групп. «Взятые» карточки будем помещать не в саму таблицу a (чтобы они не перепутались со старыми), а во вспомогательную таблицу **цел таб** b[1:1000], начиная с $b[p+1]$. Вначале в первой группе $q-p$ карточек, во второй $r-q$ карточек. Первая карточка первой группы есть $a[p+i1+1]$, второй — $a[q+i2+1]$. Первая группа непуста, если $i1<q-p$, вторая — если $i2<r-q$. Место в новой группе для очередной карточки $b[p+i1+i2+1]$. В конце работы нужно переписать элементы $b[p+1]...b[r]$ в таблицу a;

при этом используется переменная величина i — число переписанных элементов.

Заметим, что действия в четырех вариантах команды выбора бывают всего двух видов: перекладывание карточки из первой исходной группы в новую (варианты 1 и 3) и перекладывание карточки из второй исходной группы в новую (варианты 2 и 4).

```
алг слияние (цел таб a[1:1000], p, q, r)
арг a, p, q, r
рез a
нач цел i1, i2, i, цел таб b[1:1000]
  i1:=0; i2:=0
  пока i1≠(q—p) или i2≠(r—q)
  нц
    выбор
      при i1<q—p и i2<r—q и a[p+i1+
          +1]≤a[q+i2+1] (обе группы
          непусты, в первой карточка
          меньше):
          b[p+i1+i2+1]:=a[p+i1+1]
          i1:=i1+1
      при i1<q—p и i2<r—q и a[p+i1+
          +1]≥a[q+i2+1] (обе группы
          непусты, во второй карточка
          меньше):
          b[p+i1+i2+1]:=a[q+i2+1]
          i2:=i2+1
      при i1<q—p и i2=r—q (первая
          группа непуста, вторая пуста):
          b[p+i1+i2+1]:=a[p+i1+1]
          i1:=i1+1
      при i1=q—p и i2<r—q (вторая
          группа непуста, первая пуста):
          b[p+i1+i2+1]:=a[q+i2+1]
          i2:=i2+1
    все
  кц
  i:=0
  пока i≠r—p
  нц
    a[p+i+1]:=b[p+i+1]
    i:=i+1
  кц
кон
```

Завершая изложение эффективного алгоритма сортировки, отметим, что этот алгоритм довольно сложен — он заведомо сложнее всех алгоритмов, приведенных в УП (однако среди задач в конце УП есть и гораздо более сложные). Мы привели его для того, чтобы показать, что алгоритмы бывают весьма нетривиальными и что их составление — трудное, но зачастую увлекательное занятие.

Продолжение следует

Т. ПОДДУБНАЯ
Томский государственный университет

Запись алгоритмов на Бейсике

Первые шаги

Начнем с простой программы, которая читает два вещественных числа, присваивает их переменным A и B и меняет их местами, если A>B, а затем печатает результат своей работы. Для удобства справа от текста программы помещены английские слова, используемые в операторах, и их перевод на русский язык.

```
20 INPUT A, B              INPUT — ввод
25 IF A<=B THEN GO TO 45   IF — если
30 LET C=A                 THEN — то
35 LET A=B                 GO TO — иди к
40 LET B=C                 LET — пусть
45 PRINT A, B              PRINT — печатать
50 STOP                    STOP — останов
```

Что сразу обращает на себя внимание?

Прежде всего, нумерация операторов (команд) целыми числами.

Использование в записи команд ввода данных (INPUT) и вывода результатов (PRINT).

Хотя программа и понятна, она не похожа на изученный ранее способ оформления алгоритмов.

Совершенно очевидно, что операторы с номерами 30, 35 и 40 — операторы присваивания. Для их записи используется служебное слово LET и знак = (на некоторых ДВК слово LET можно опускать).

Оператор с номером 25 организует ветвление в зависимости от истинности условия A>B. Отметим, что он тоже отличается от команды ветвления типа **если—то—иначе.**

В программе используется так называемый оператор безусловного перехода (GO TO). Без него невозможно написать ни одну программу на Бейсике, в которой требуется организовать ветвление команд. Таков уж этот язык!

Вторая программа последовательно читает

Методика обучения

1/1988

А. ШЕНЬ

Информатика в IX классе

Построение алгоритмов для решения задач из курса математики. В УП разбираются три математические задачи: вычисление значения многочлена по схеме Горнера, приближенное вычисление площадей и поиск корня.

По поводу первой задачи мы ограничимся единственным комментарием к приведенному в УП алгоритму. Его работу легче понять, если иметь в виду, что после любого числа циклов исполнения команды повторения (с. 63) выполняется такое соотношение:

$$y=(\ldots((a[0]x+a[1])x+a[2])x+\ldots+a[i-1])x+a[i]$$

(т. е. сделано i шагов вычислений по схеме Горнера, вплоть до коэффициента $a[i]$). В начале $i=0$ и $y=a[0]$. На каждом шаге мы увеличиваем i на 1 и выполняем еще один этап вычислений по схеме Горнера. Цикл завершается, когда $i=n$, т. е. использованы все коэффициенты многочлена вплоть до $a[n]$.

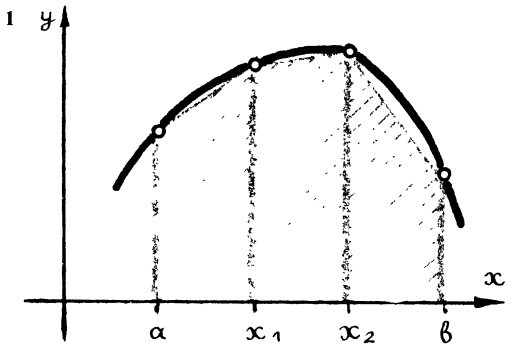
Второй пример — традиционная задача численного интегрирования, т. е. приближенного нахождения интеграла. Однако слово «интеграл» в УП заменяется словами «площадь криволинейной трапеции». Рассматриваемый метод вычислений называется «метод прямоугольников». На практике он применяется редко, так как дает меньшую точность, чем другие (ничуть не более сложные) методы. Кроме того, у школьников может вызвать недоумение то обстоятельство, что он не дает точного ответа даже в случае, когда криволинейная трапеция является обычной «прямолинейной» трапецией (т. е. для функции $f(x)=ax+b$).

Поэтому можно предложить разобрать другой алгоритм приближенного вычисления площадей — метод трапеций. Его идея в том, что мы заменяем кривую ломаной, составленной из отрезков прямых (рис. 1). При этом криволинейная трапеция заменяется фигурой, состоящей из нескольких обыкновенных трапеций, площадь которой будет равна сумме площадей трапеций. Подсчитаем эту сумму, предположив, что интересующий нас отрезок $[a, b]$ разбит на n равных частей точками x_1, x_2, ..., x_{n-1}. В этом случае высота h каждой трапеции равна $(b-a)/n$, основания равны $f(x_i)$ и $f(x_{i+1})$ (а площадь трапеции равна произведению высоты на полусумму оснований). Например, для $n=3$ (изображенный на рис. 1 случай) получаем

$$S=\frac{h}{2}f(a)+hf(x_1)+hf(x_2)+\frac{h}{2}f(b).$$

Видно, что все значения функции входят в эту формулу с одинаковыми коэффициентами, кроме двух крайних, коэффициенты при которых вдвое меньше (это не удивительно: ведь значе-

Окончание. Начало см.: Информатика и образование. 1987. № 4—6.



ния функции в промежуточных точках влияют на площадь и слева, и справа от себя, а значения функции в крайних точках — только в одну сторону). Получаем такой алгоритм (вначале вычисляются члены, соответствующие крайним точкам, а затем добавляются слагаемые для внутренних точек):

```
алг ПЛОЩАДЬ (вещ a, b, s, цел n)
    арг a, b, n
    рез s
нач вещ h, x, цел i
    (h — высота трапеций, i — число
    учтенных внутренних точек,
    x — первая еще не учтенная внутренняя точка; s — сумма по крайним
    и первым i внутренним точкам)
    h:=(b—a)/n
    i:=0
    x:=a+h
    s:=(f(a)+f(b))*h/2
    пока i≠n—1
    нц
      s:=s+f(x)*h
      x:=x+h
      i:=i+1
    кц
кон
```

Видно, что этот алгоритм имеет практически ту же сложность, что и алгоритм метода прямоугольников, приведенный в УП. Точность его, однако, гораздо выше (грубо говоря, он дает вдвое больше верных цифр при том же числе n отрезков деления). Существуют и другие, еще более точные алгоритмы — метод парабол (формула Симпсона), интегрирование по нулям ортогональных многочленов (метод Гаусса) и др. Разработка и анализ таких алгоритмов являются задачами раздела математики, называемого вычислительной математикой.

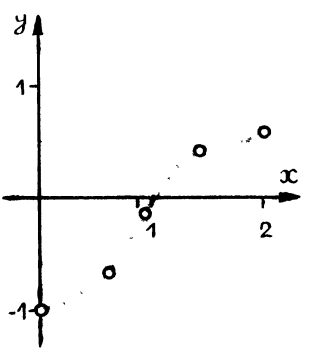
К этому же разделу математики примыкает следующий пример из УП — отыскание приближенных значений корней. Эта задача делится на две части — «отделение» корня (поиск интервала, в котором он заключен) и «уточнение» (нахождение значения корня с нужной точностью). Решение первой части требует общих сведений о свойствах уравнения, корень которого ищется; ничего содержательного в рамках школьного курса о ее решении сказать нельзя. Напротив, вторая часть может стать предметом подробного обсуждения на уроке.

В курсе математики постоянно решают самые разные уравнения, т. е. ищут их корни (если они, конечно, есть). На самом деле эти уравнения специально подобраны так, чтобы их можно было решить теми методами, которые проходят на уроках математики. Стоит чуть-чуть отклониться от стандартных типов уравнений, и мы сразу окажемся беспомощными. Попробуйте, например, найти корень у $x\sin x - 1 = 0$.

Может быть, решение не удается найти потому, что его нет? Давайте составим таблицу значений левой части этого уравнения ($x\sin x - 1$) при нескольких значениях x:

c	0,0	0,5	1,0	1,5	2,0
$x\sin x - 1$	—1,0	—0,75	—0,15	0,49	0,81

Нарисуем примерный график функции $y = x\sin x - 1$ (рис. 2), отметив на рисунке точки из приведенной таблицы (пунктир-



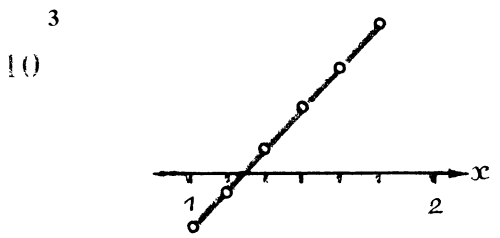
2

ная линия проведена наугад через эти точки). Корень уравнения $x\sin x - 1 = 0$ — это такое x, при котором $x\sin x - 1$ обращается в 0, т. е. график пересекает ось абсцисс. Посмотрев на график, мы видим, что участок от 1,0 до 1,5 является «подозрительным», поскольку при переходе от 1,0 к 1,5 выражение $x\sin x - 1$ меняет знак: из отрицательного становится положительным. Изучим этот участок графика более подробно.

x	1,0	1,1	1,2	1,3	1,4	1,5
$x\sin x - 1$	−0,15	−0,02	0,12	0,25	0,38	0,49

Похоже, что корень находится где-то между 1,1 и 1,2 (рис. 3).

3



На самом деле существует математическая теорема, из которой следует, что действительно на участке от 1,1 до 1,2 имеется корень нашего уравнения. Эта теорема утверждает, что если функция f такова, что $f(a) \leqslant 0$, $f(b) \geqslant 0$, причем f непрерывна (это означает, грубо говоря, что график f — линия, не имеющая разрывов), то на отрезке $[a, b]$ имеется такое x, что $f(x) = 0$. В нашем случае $f(x) = x \sin x - 1$, $a = 1{,}1$, $b = 1{,}2$. Функция $f(x) = x \sin x - 1$, как доказывается в курсах высшей математики, непрерывна. Поэтому на отрезке [1,1, 1,2] наше уравнение имеет корень (на самом деле этот корень равен 1,114157...).

Итак, что же получается? Уравнение не решается методами, разбираемыми в курсе математики, но имеет корень. Как же этот корень найти?

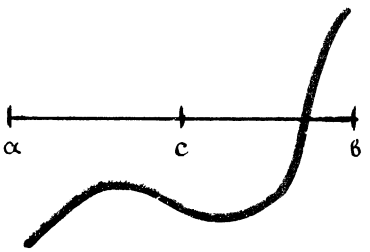
Вернемся на минуту к обыкновенным квадратным уравнениям. Пусть, например, нам нужно изготовить стержень длиной x сантиметров, где x — корень уравнения $x^2 - x - 1 = 0$ (у него два корня, но один отрицателен и потому отпадает). Решая это уравнение по известной формуле, получаем: $x = (1+\sqrt{5})/2$. Но эта форма ответа может удовлетворить нас, лишь пока мы занимаемся математикой. Как только мы начинаем изготавливать стержень длиной x сантиметров, мы убеждаемся, что на линейке нет деления «$(1+\sqrt{5})/2$ см», а есть лишь сантиметровые и миллиметровые деления. Что же делать? Надо посмотреть в таблице — или вычислить с помощью микрокалькулятора — приближенное значение $\sqrt{5}$. Оно равно 2,236068...; подставляя его, получаем, что $x = 1{,}6180...$ Теперь остается лишь отмерить линейкой нужный отрезок стержня. (При этом наша точность даже избыточна: вряд ли на линейке есть более мелкие, чем миллиметровые деления, так что хватило бы двух цифр после запятой.)

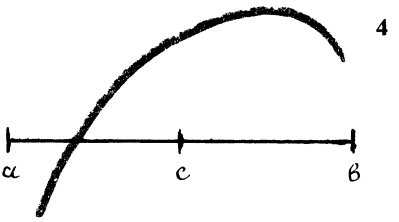
Мы искали $\sqrt{5}$ в таблице или пользовались калькулятором. Но как узнали составители таблиц, что $\sqrt{5}$ примерно равен 2,23606...? Для калькулятора вопрос звучит так: что делает программа, управляющая калькулятором, когда мы нажимаем клавиши, требующие вычислить $\sqrt{5}$?

Изложим один из возможных способов приближенного нахождения $\sqrt{5}$. Будем искать его «подбором». Нам надо найти такое x, что $x^2 = 5$. Попробуем $x = 2$. Тогда $x^2 = 4$. Это мало. Пробуем $x = 3$. Тогда $x^2 = 9$. Это много. Значит, нужное x расположено где-то посередине между 2 и 3. Пробуем $x = 2{,}5$. Тогда $x^2 = 6{,}25$. Много. Значит, нужное x где-то между 2 и 2,5. Пробуем $x = 2{,}25$, тогда $x^2 = 4{,}415625$. Мало. Значит, x между 2,125 и 2,25. Пробуем середину: $x = 2{,}1875$, $x^2 = 4{,}785...$ — снова мало, значит, x между 2,1875 и 2,25, пробуем $x = 2{,}21875$, $x^2 = 4{,}922...$, снова мало, x между 2,21875 и 2,25, пробуем середину: $x = 2{,}234374$, $x^2 = 4{,}9824...$, снова мало, x между 2,234375 и 2,25.

Этот процесс можно продолжать бесконечно, но остановимся здесь. Что мы получили? Мы узнали, что $\sqrt{5}$ лежит между 2,23 и 2,25, так что если мы примем его равным 2,24, то совершим ошибку не более одной сотой. Если такая точность нас устраивает, то на достигнутом можно остановиться. Если же нет, то можно продолжать описанный процесс поиска корня делением пополам, пока не достигнем нужной точности.

Подобный способ отыскания корня





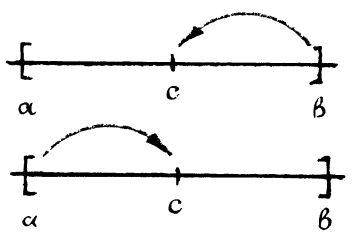
4

можно применить и к уравнению $x \sin x - 1 = 0$, а также к многим другим уравнениям вида $f(x) = 0$, где $f(x)$ — какое-то выражение, содержащее x. Назовем отрезок $[a, b]$ отрезком перемены знака, если $(f(a) \leqslant 0$ и $f(b) \geqslant 0)$ или $(f(a) \geqslant 0$ и $f(b) \leqslant 0)$. Эквивалентное определение: $[a, b]$ — отрезок перемены знака, если $f(a) \cdot f(b) \leqslant 0$.

Основой рассматриваемого способа отыскания корней является такое замечание: если $[a, b]$ — отрезок перемены знака, а c — его внутренняя точка, то один из отрезков $[a, c]$ и $[c, b]$ является отрезком перемены знака. На рис. 4 слева отрезком перемены знака является $[c, b]$, справа $[a, c]$.

Способ отыскания корня состоит в том, что мы делим отрезок перемены знака пополам и выбираем из двух его половин ту, которая сама является отрезком перемены знака. Этот процесс повторяется до тех пор, пока длина отрезка не станет достаточно малой. После этого середина полученного отрезка считается приближенным значением корня.

Алгоритм этот приведен в библиотеке алгоритмов (А11). При его исполнении после любого цикла отрезок $[a, b]$ является отрезком перемены знака. Команда ветвления действует так: если левая половина отрезка является отрезком перемены знака, то мы сужаем отрезок, сдвигая влево его правый конец (рис. 5). В противном случае отрезком перемены знака является правая половина исходного отрезка и мы сужаем отрезок, сдвигая его левый конец (рис. 6).



5

6

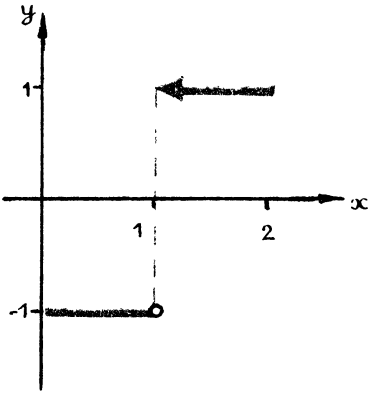
После завершения исполнения команды повторения входящее в нее условие не соблюдается, т. е. $b - a \leqslant 2\varepsilon$, и середина отрезка $[a, b]$ (т. е. число $(a+b)/2$) является приближением к корню с точностью не хуже ε.

Для каких функций имеет смысл применять наш алгоритм? Мы уже говорили, что функция f должна менять знак. В этом случае в ходе исполнения алгоритма будет найден отрезок изменения знака, имеющий малую длину, и в качестве результата работы алгоритма будет выдана его середина. Правда, для того чтобы мы могли заключить, что найдено приближение к корню, мы должны еще знать, что функция непрерывна. Например, для функции

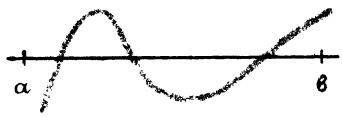
$$f(x) = \begin{cases} -1, & \text{если } x \in [0,1] \\ 1, & \text{если } x \in [1,2] \end{cases}$$

(рис. 7) отрезок [2, 2] является отрезком перемены знака, но корней там нет (данная функция не является непрерывной); применение нашего алгоритма приведет к отрезку малой длины, окружающему точку $x = 1$.

Кроме того, уравнение $f(x) = 0$ (в том числе и для непрерывной f) может иметь



7



несколько корней на отрезке перемены знака (рис. 8). В этом случае наш способ позволит отыскать один из них. Если функция f монотонна (как написано в описании рассматриваемого алгоритма на с. 95 УП), то корень может быть только один. Так что для монотонной непрерывной функции, меняющей знак на некотором отрезке, этот алгоритм действительно позволяет найти ее единственный корень с заданной точностью.

Этот раздел УП содержит три примера. Примеры трудные — в первую очередь тем, что для их понимания требуется свободное владение соответствующим материалом из курса физики. Если вы почувствуете, что хорошо разобрать их все за отведенное для прохождения этой темы время не удается, мы рекомендуем ограничиться разбором любых двух (по вашему выбору).

Метод наименьших квадратов. Пример 1 демонстрирует математические методы обработки полученной в результате эксперимента информации на простейшем материале. Приведенные в УП числовые данные могут быть использованы на уроке, если у школьников есть возможность пользоваться непрограммируемыми микрокалькуляторами. Если же такой возможности нет, то нужно выбрать более простой для расчета вариант, иначе вычисления отнимут слишком много времени. Вот один из возможных вариантов.

$U_1=1$В $\quad I_1=1$А
$U_2=3$В $\quad I_2=2$А
$U_3=5$В $\quad I_3=4$А

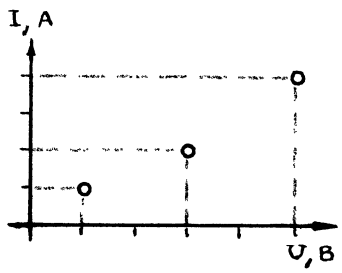
Обсуждая эти данные, следует добиться от школьников понимания следующих вещей.

А. Результаты измерений не соответствуют закону Ома: если вычислять сопротивление по формуле $R=U/I$, то для каждого из трех измерений оно будет свое (1 Ом, 1,5 Ом и 1,25 Ом).

Б. Это несоответствие можно объяснить двумя причинами: либо наши измерения не совсем точные, либо исследуемый «резистор» на самом деле является не резистором, и ток через него зависит от приложенного напряжения не линейно, как того требует закон Ома, а более сложным образом.

В. Выбор между этими двумя объяснениями зависит от точности применяемых нами приборов: чем точнее приборы, тем менее вероятно, что отклонения результатов измерений от теоретической линейной зависимости — всего лишь ошибки.

Г. Если мы пришли к выводу о том, что отклонения от закона Ома объясняются погрешностями измерения, связанными с недостаточной точностью приборов, то для оценки сопротивления резистора следует провести прямую $I=kU$ как можно ближе к отмеченным на графике (рис. 9) точкам и считать сопротивлением резистора число $1/k$.



9

После того как школьники все это поняли, можно сообщить им формулу для вычисления коэффициента k. Хотя ученикам предлагается готовая формула для вычисления k, происхождение которой остается для них неизвестным, учителю следует знать ее смысл. Если

$$(U_1, I_1), (U_2, I_2), \ldots, (U_n, I_n)$$

экспериментальные точки, то с помощью этой формулы можно найти такое k, что

$$(I_1-kU_1)^2+\ldots+(I_n-kU_n)^2$$

(сумма квадратов отклонений) будет минимальной (это объясняет название «метод наименьших квадратов»). Убедиться в том, что это действительно так, можно следующим образом: раскрыв скобки, получаем квадратный трехчлен относительно k:

$$k^2(U_1^2+\ldots+U_n^2)-2k(U_1I_1+\ldots+U_nI_n)+$$
$$+(I_1^2+\ldots+I_n^2).$$

Квадратный трехчлен ak^2+bk+c как функция от k при $a>0$ принимает наименьшее значение в точке $k=-b/2a$. В данном случае $a=U_1^2+\ldots+U_n^2$, $b=-2(U_1I_1+\ldots+U_nI_n)$, откуда получается приведенная в учебнике формула.

Прежде чем применять формулу метода наименьших квадратов к экспериментальным данным, полезно опробовать ее вместе с учениками на простых примерах. Самый простой пример получится, если есть только одна экспериментальная точка. В этом случае получаем $k=UI/U^2=I/U$ т. е. прямая $I=kU$ проходит как раз через эту точку. Более сложный пример (разобранный в УП) получится, когда точек несколько, но все они в точности лежат на прямой $I=kU$. И в этом случае формула дает ожидаемый результат.

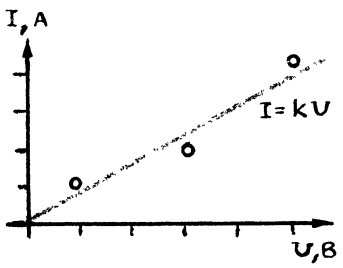
Теперь подставим в формулу имеющиеся экспериментальные данные:

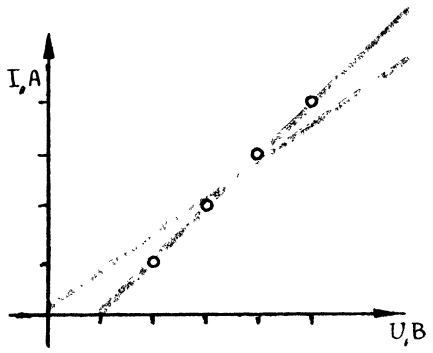
$$k=\frac{U_1I_1+U_2I_2+U_3I_3}{U_1^2+U_2^2+U_3^2}=\frac{1+6+20}{1+9+25}=\frac{27}{35};$$
$$R=1/k=35/27\approx 1{,}3\ \text{Ом}.$$

Чтобы убедиться еще раз в правдоподобности полученного ответа, нанесем прямую $I=kU$ на график (рис. 10). Видно, что она действительно проходит близко от экспериментальных точек и что отклонения действительно можно истолковать как погрешности измерения. Можно еще сравнить полученное значение сопротивления (1,3 Ом) с результатами, получающимися из отдельных измерений (1 Ом, 1,5 Ом, 1,25 Ом), и убедиться, что оно действительно лежит где-то посередине между ними.

Главная мысль, которую следует донести до школьников в ходе обсуждения, состоит в том, что при математической обработке результатов экспери-

10





ментов нельзя ни на минуту забывать о физическом смысле, стоящем за формулами и алгоритмами, иначе можно получить абсурдные результаты. В этом отношении поучителен такой пример.

Пусть в результате измерений мы получили:

$U_1=2$В	$I_1=1$А
$U_2=3$В	$I_2=2$А
$U_3=4$В	$I_3=3$А
$U_4=5$В	$I_4=4$А

Воспользовавшись формулой метода наименьших квадратов, находим, что

$$k=27/20,\ R=1{,}35\ \text{Ом}.$$

Однако, нанеся соответствующую прямую на график (рис. 11, синяя линия), мы видим, что наблюдается систематическое отклонение экспериментальных точек от этой линии. Зато они точно ложатся на зеленую прямую, задаваемую формулой $I=U-1$. Хорошо знающие физику ребята сразу сообразят, что такая закономерность может быть объяснена тем, что наш «резистор» на самом деле вовсе не резистор, а батарейка с электродвижущей силой в 1 В и внутренним сопротивлением 1 Ом.

Метод наименьших квадратов применим в тех случаях, когда одна величина изменяется пропорционально другой и нас интересует коэффициент пропорциональности. Среди упражнений к § 8 есть два, которые можно использовать в качестве иллюстрации его применения.

В первом из них длина участка карандаша, покрытого нитью, пропорциональна числу витков. Вот результаты проведенного нами эксперимента (хлопчатобумажные нитки № 40):

Число витков	10	20	30	40	50	60	70	80	90	100
Длина в мм	3	5	8	11	14	16	18	21	24	27

По формуле метода наименьших квадратов получаем, что коэффициент k (в данном случае — толщина нити) равен примерно 0,27 (мм).

Упражнение 3 (измерение толщины листа бумаги) удобно несколько видоизменить, проводя его с книгой и измеряя толщину заданного числа листов в книге с помощью штангенциркуля. Вот результаты такого эксперимента (проведенного с книгой «Изучение основ информатики и вычислительной техники». М.: Просвещение, 1985):

Число листов									
5	10	15	20	25	30	35	40	45	50
Толщина, мм									
0,4	0,9	1,3	1,8	2,4	2,8	3,3	4,0	4,3	4,9

По формуле метода наименьших квадратов находим коэффициент k (толщину листа в миллиметрах), который оказывается примерно равным 0,096.

Колебания шарика на пружине. Этот пример иллюстрирует методы математического моделирования физических процессов. Данный физический процесс описывается дифференциальным уравнением $mx''t=-kx(t)$ (m — масса шарика, k — жесткость пружины, x(t) — координата шарика в момент t). Оно называется уравнением гармонических колебаний и может быть решено аналитически. При этом получается такой ответ: если в начальный момент времени шарик отклонен на x_0 и отпущен без начальной скорости, то его координата в произвольный момент t находится по формуле

$$x(t)=x_0 \cos(\sqrt{k/m}\,t).$$

Это точное решение, однако, пока недоступно школьникам. Зато численное, приближенное, решение этого уравнения вполне доступно им уже сейчас. Подобная ситуация складывается и в реальных физических исследованиях: точное аналитическое решение возникающих задач зачастую оказывается невозможным, в то время как приближенное решение с помощью ЭВМ возможно и дает нужный результат с достаточной для нас точностью.

Приступая к разбору примера, полезно сначала предложить ученикам несколько простых вопросов подготовительного характера, например таких.

А. Координата движущегося по прямой шарика в момент t равна x, а скорость равна v. Чему будет равна координата шарика через промежуток времени Δt? (Ответ: $x+v\Delta t$. Он является приближенным — мы не учитываем изменение скорости шарика; точность приближения тем выше, чем меньше Δt.)

Б. Скорость движущегося по прямой шарика в момент времени t равна v, а ускорение равно a. Чему будет равна скорость шарика через промежуток времени Δt? (Ответ: $v+a\Delta t$. Он, как и предыдущий, является приближенным: мы не учитываем изменения ускорения шарика.)

Далее полезно напомнить о законе Гука, согласно которому сила, с которой растянутая пружина действует на прикрепленное к ней тело, пропорциональна ее удлинению: $F=-kx$ (знак минус поставлен, так как сила направлена противоположно удлинению); коэффициент k называется жесткостью пружины. После этого в беседе с классом следует добиться понимания следующих трех вещей:

А. В положении равновесия на висящий на пружине шарик действуют две равные по величине и противоположные по направлению силы: сила тяжести и сила упругости. Их сумма (результирующая сила, действующая на шарик) равна 0, потому-то он и находится в равновесии.

Б. Если отклонить шарик на расстояние x от положения равновесия, то сила тяжести останется неизменной, а к силе упругости прибавится число $-kx$. Таким образом, результирующая сила будет равна $-kx$.

В. Ускорение шарика можно найти, использовав второй закон Ньютона: $F=ma$, откуда $a=F/m=-(k/m)x$.

Теперь школьники готовы к заполнению таблицы. Заполнение производится по строкам, строки заполняются слева направо. Ускорение в момент начала интервала определяется по формуле $a=-kx/m$, скорость в середине интервала определяется по ускорению в начале интервала и по скорости в начале интервала (первая строка таблицы) или в середине предыдущего интервала (все следующие строки). Затем эта скорость ис-

пользуется для нахождения координаты к началу следующего интервала. Весь этот процесс подробно описан в УП. Заполнив несколько строк таблицы и сверив их с УП, можно построить график движения шарика (зависимость координаты шарика от времени). Построив график, следует обратить внимание школьников на характер происходящего процесса: сначала скорость шарика мала (координата меняется медленно, наклон касательной к графику мал). Затем шарик разгоняется (модуль скорости увеличивается, кривая идет все круче); между $t=0{,}07$ и $t=0{,}08$ шарик проходит положение равновесия (координата равна нулю, модуль скорости максимален) и начинает замедляться. Эта картина колебаний хорошо знакома школьникам: ее можно наблюдать, например, с помощью качающегося маятника.

Полезно сравнить данные в таблице с полученными по точной формуле $x(t)=x_0 \cos(\sqrt{k/m}\,t)$.

Номер	1	2	3	4	5	6	7	8	9	10
Точный ответ	1,00	0,98	0,92	0,83	0,70	0,54	0,36	0,17	—0,08	—0,23
Результат вычислений	1,00	0,98	0,92	0,82	0,69	0,53	0,35	0,16	—0,004	—0,24

Видно, что они довольно близки. Это показывает, что выбранное нами приближение (мы считали, что на интервале длиной $\Delta t=0{,}01$ скорость и ускорение шарика меняются мало) достаточно точно. Даже если мы увеличим интервал времени Δt, взяв его равным 0,02 с, как предлагается в упражнении 5, то все равно согласие между точным решением и приближенным остается достаточно хорошим.

Номер	1	2	3	4	5
Точный ответ	1,00	0,92	0,70	0,36	—0,08
Результат вычислений	1,00	0,92	0,69	0,35	—0,04

Следует подчеркнуть, что излагаемый в данном примере материал весьма труден. По существу, школьники познакомились с простейшим методом численного решения дифференциальных уравнений (он называется методом ломаных Эйлера) и с его помощью численно решили дифференциальное уравнение гармонических колебаний. Многие другие задачи школьного курса физики также могут решаться таким методом (например, нахождение орбит планет). Использование компьютеров для решения физических задач может стать увлекательной темой факультативных занятий.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ПРОВЕДЕНИЮ ПРАКТИКУМА В X КЛАССЕ

Е. УТЛИНСКИЙ, Д. СМЕКАЛИН, А. ДОДОНОВ

Знакомство с прикладным программным обеспечением

Описанию прикладного программного обеспечения в пробном учебном пособии (УП) посвящен фактически только один параграф; это объясняется тем, что оно ориентировано прежде всего на «безмашинный» вариант курса. Однако материал этот чрезвычайно важен как с точки зрения иллюстрирования возможностей ЭВМ и их применений, так и с точки зрения освоения школьниками компьютерной грамоты. Поэтому на практических занятиях необходимо не только организовать работу учащихся с готовыми программами, но и подробно разобрать теоретические вопросы, связанные с их использованием и областями применения.

Прикладное программное обеспечение определяет реальные возможности применения ЭВМ во всех областях народного хозяйства, не связанных непосредственно с разработкой новых программ. Прикладные программы должны облегчать человеку решение с помощью