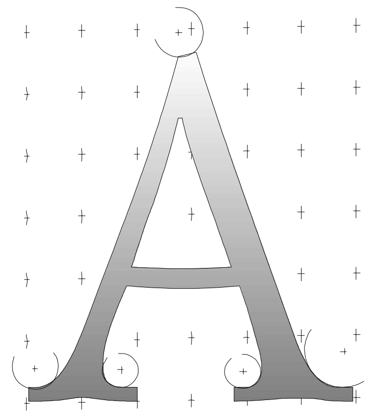
№ 8 (201) февраль 1999

еженедельное
приложение к газете
«ПЕРВОЕ СЕНТЯБРЯ»

ИНФОРМАТИК



Электронные версии газеты "Первое сентября" и приложений http://www.1september.ru

Кривые Гильберта и Серпинского, или Снова рекурсия

Д.М. ЗЛАТОПОЛЬСКИЙ

В статье [1] был приведен ряд рекурсивных алгоритмов для построения изображений, составленных из повторяющихся фигур. Ряд аналогичных программ представлен в [2]. С использованием рекурсии можно получить также фигуры, показанные на *рис.* 1 и 2. Первая из них называется кривая Гильберта, вторая — кривая Серпинского.

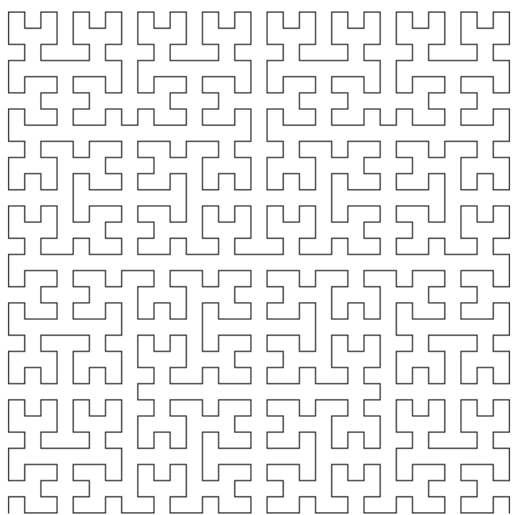


Рис. 1

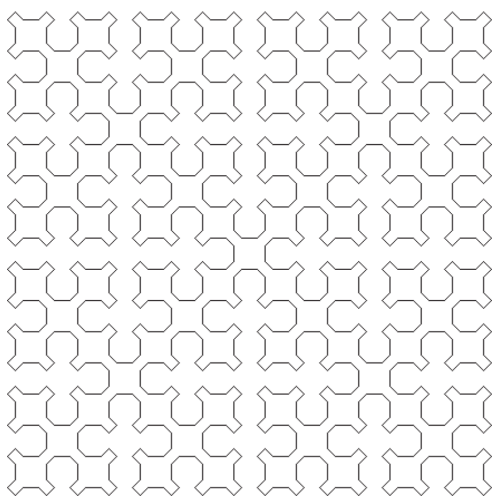
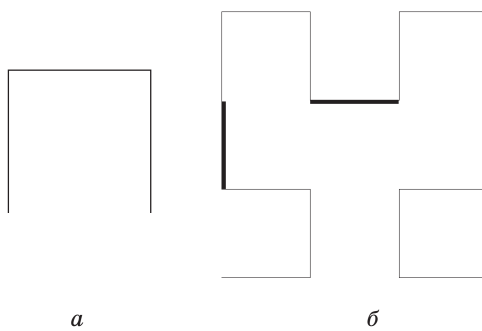


Рис. 2

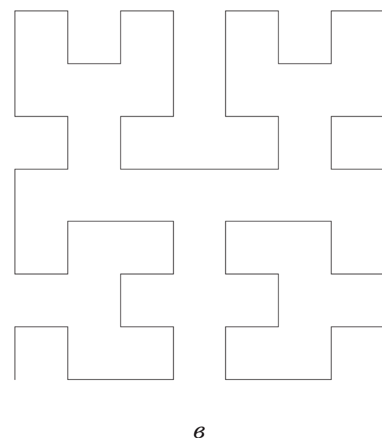
Эти кривые связаны с любопытным понятием теории функций, а именно — *всюду плотными кривыми* [7]. Кривая на плоскости называется всюду плотной в некоторой области, если она проходит через любую сколь угодно малую окрестность каждой точки этой области. Несколько упрощенно можно считать, что всюду плотные кривые целиком заполняют указанную область. Известные математики Гильберт и Серпинский построили примеры всюду плотных кривых. Хотя эти примеры различны, схема получения соответствующих кривых одинакова. По определенному правилу строятся кривые (соответственно Гильберта и Серпинского) первого, второго, ..., n-го порядка, вписанные в заданный квадрат. При неограниченном возрастании n они стремятся к некоторой предельной кривой, которая является всюду плотной в заданном квадрате.

В этом номере мы рассмотрим алгоритм построения кривой Гильберта, а в следующем — кривой Серпинского.

Кривая Гильберта первого порядка, обозначаемая H_1, похожа на изображение буквы *П*, вычерченной в виде трех сторон квадрата, как показано на *рис. 3а*. На *рис. 3б* изображена кривая Гильберта второго порядка H_2. Видно, что кривая H_2 состоит из кривых H_1, ориентированных в разные стороны (вправо, вверх и влево). Кривые H_1, составляющие кривую H_2, соединены тремя отрезками прямых, называемых связками (на *рис. 3б* они вычерчены утолщенными линиями). В действительности эти отрезки должны иметь одинаковую толщину с другими линиями, утолщенными они показаны единственно с целью демонстрации способа получения H_2 из H_1.



а *б*



в

Рис. 3. Кривые Гильберта первого, второго и третьего порядков

Аналогично фигуру H_3 (*рис. 3в*) можно рассматривать как состоящую из четырех кривых H_2 (ориентированных в разные стороны) и трех связок.

Заметим, что отрезки, образующие линию H_1, можно рассматривать как связки, соединяющие 4 точки — кривые Гильберта нулевых порядков.

Таким образом, кривую Гильберта i-го порядка H_i можно получить из четырех кривых H_{i-1}, ориентированных в разные стороны, и трех связок. Если процедуры рисования кривых H_i, ориентированных вверх, вниз, влево и вправо, обозначить соответственно GU(i), GD(i), GL(i) и GR(i), то можно составить следующие рекурсивные схемы построения этих кривых:

```
GU(i): GR(i–1)↑ GU(i–1)→ GU(i–1)↓ GL(i–1)
GR(i): GU(i–1)→ GR(i–1)↑ GR(i–1)← GD(i–1)
GD(i): GL(i–1)↓ GD(i–1)← GD(i–1)↑ GR(i–1)
GL(i): GD(i–1)← GL(i–1)↓ GL(i–1)→ GU(i–1)
```

Обозначим через h длину элементарного отрезка прямых в кривых H_i. Тогда, например, процедура GU в программе на школьном алгоритмическом языке [3, 4] может быть оформлена следующим образом:

```
алг GU(арг цел i)
нач
  если i>0
    то
      GR(i-1)
      LineUp
      GU(i-1)
      LineRight
      GU(i-1)
      LineDown
      GL(i-1)
  все
кон
```

Здесь LineUp, LineRight, LineLeft, LineDown — процедуры рисования связок, направленных соответственно вверх, вправо, влево и вниз (напомним, что ось Y на экране направлена сверху вниз).

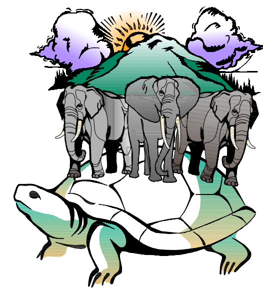
Продолжение на с. 2

НАШИ ДЕТИ БУДУТ ЖИТЬ В XXI ВЕКЕ

12 лекций о том, для чего нужен школьный курс информатики и как его преподавать

Лекции 6, 7—8

А.Г. КУШНИРЕНКО, Г.В. ЛЕБЕДЕВ



Об основных понятиях, идеях и целях школьного курса информатики "по учебнику" *А.Г. Кушниренко, Г.В. Лебедева, Р.А. Свореня* "Основы информатики и вычислительной техники" (М.: Просвещение, 1990, 1991, 1993, 1996). Дается также ряд практических советов, предлагаются соответствующие методические приемы.

Авторы надеются, что материал окажется полезным для учителей и методистов, использующих указанный учебник, а также для тех, кто желает сравнить разные подходы к преподаванию школьного курса информатики или разработать свой собственный курс.

Продолжение следует

2 15

ЗАДАЧИ

• КРИВЫЕ ГИЛЬБЕРТА И СЕРПИНСКОГО, ИЛИ СНОВА РЕКУРСИЯ

Д.М. ЗЛАТОПОЛЬСКИЙ

В первой части статьи рассматривается рекурсивный алгоритм построения кривой Гильберта, приводятся его реализации на различных языках программирования. Во второй части статьи, которая будет опубликована в следующем номере, обсуждается алгоритм построения кривой Серпинского.

16

ВНЕКЛАССНАЯ РАБОТА ПО ИНФОРМАТИКЕ

• ИНФОРМАТИКА ПОСЛЕ УРОКОВ

Вопросы для проведения школьных конкурсов "Что? Где? Когда?", "Брейн-ринг", викторин

Д.М. ЗЛАТОПОЛЬСКИЙ

Отвечая на вопросы в наших анкетах, читатели часто отмечают необходимость публикации материалов для организации внеклассной работы по информатике, проведения конкурсов, викторин, предметных недель. Сегодня мы публикуем первый материал новой рубрики и приглашаем всех стать ее авторами.

Кривые Гильберта и Серпинского, или Снова рекурсия

Продолжение. Начало на с. 1

```
алг LineUp
нач
| вектор(0, -d)
кон

алг LineRight
нач
| вектор(d, 0)
кон

алг LineLeft
нач
| вектор(-h, 0)
кон

алг LineDown
нач
| вектор(0, d)
кон
```

Аналогично можно оформить процедуры рисования кривых Гильберта, ориентированных вниз, влево и вправо:

```
алг GD(арг цел i)
нач
  если i>0
    то
    GL(i-1)
    LineDown
    GD(i-1)
    LineLeft
    GD(i-1)
    LineUp
    GR(i-1)
  все
кон

алг GL(арг цел i)
нач
  если i>0
    то
    GD(i-1)
    LineLeft
    GL(i-1)
    LineDown
    GL(i-1)
    LineRight
    GU(i-1)
  все
кон

алг GR(арг цел i)
нач
  если i>0
    то
    GU(i-1)
    LineRight
    GR(i-1)
    LineUp
    GR(i-1)
    LineLeft
    GD(i-1)
  все
кон
```

Обратим внимание на то, что в приведенных процедурах рисования кривых Гильберта используется так называемая косвенная рекурсия — ситуация, когда процедура вызывает себе как вспомогательную не только непосредственно, а также и через другую процедуру [1].

Квадрат, в который вписывается кривая Гильберта, будем называть опорным, длину его стороны (в пикселях) обозначим через *S*. Обсудим теперь вопрос определения значения величины *h* в зависимости от порядка кривой *n*. Из *рис.* 3 видно, что при $n=2$ длина элементарного отрезка линии в три раза меньше стороны опорного квадрата, при $n=3$ — в семь раз. Отсюда получаем, что коэффициенты уменьшения для этих элементарных отрезков в фигурах H_1, H_2, H_3, ... образуют ряд чисел 1, 3, 7, ..., то есть в общем случае коэффициент уменьшения для фигуры H_n может быть вычислен по формуле 2^n-1.

Естественно расположить изображаемую кривую Гильберта по центру экрана. Для этого надо найти координаты x_0, y_0 начальной точки кривой. Проанализировав приведенные выше процедуры, можно убедиться, что при ориентации кривой вверх и влево она начинается с левой нижней точки опорного квадрата, т.е.

$$x_0=X_c - S/2; \qquad y_0=Y_c+S/2,$$

а в остальных случаях — с правой верхней точки опорного квадрата, т.е. в этих случаях

$$x_0=X_c + S/2; \qquad y_0=Y_c - S/2$$

Здесь X_c, Y_c — координаты центра экрана.

Кроме того, удобно задавать размер опорного квадрата в процентах от высоты экрана, поскольку она всегда меньше ширины. Эту величину обозначим `PrS`. В программе построения кривой Гильберта используем, помимо указанных обозначений, еще переменную `orient` — число, определяющее ориентацию кривой (вверх — 1, вниз — 2, вправо — 3, влево — 4).

Школьный алгоритмический язык

```
цел h  |Величину h описываем как глобальную
алг Кривая Гильберта
нач цел n, x0, y0, s, orient, Hscr, Wscr
  вещ PrS
  Hscr:=480 |Высота экрана
  Wscr:=640 |Ширина экрана
  |Вводим исходные данные для построения
  | кривой Гильберта
  нц
    вывод нс, "Введите длину стороны
               опорного квадрата"
    вывод  "в % от высоты экрана"
    ввод PrS
  кц при PrS<100
  вывод нс, "Введите порядок кривой"
  ввод n
  нц
    вывод нс, "Введите ориентацию кривой"
    вывод "(вверх - 1, вниз - 2,
           вправо - 3, влево - 4)"
    ввод orient
  кц при  (orient>=1) и (orient<=4)
  S:=Int(PrS/100*Hscr)
  |Сторона опорного квадрата
  h:=div(S, 2**n-1)| Длина связок
  |Находим координаты начальной точки кривой
  если (orient=1) или (orient=3)
    то
      x0:=Div(Wscr,2)- Div(S,2)
      y0:=Div(Hscr,2)+ Div(S,2)
    иначе
      x0:=Div(Wscr,2)+ Div(S,2)
      y0:=Div(Hscr,2)- Div(S,2)
  все
  |Устанавливаем графический режим работы экрана
  видео(17)|VGA экран 640*480
  поз(x0, y0)  |Начальная точка кривой
  |Рисуем соответствующий вариант
  |кривой Гильберта
  выбор
    при orient=1: GU(n)
    при orient=2: GD(n)
    при orient=3: GR(n)
    иначе GL(n)
  все
  |Переходим в текстовый режим
  видео(0)
кон
```

Чего не хватает в этой программе? Хотелось бы наблюдать последовательность построения кривой, а для этого в конец процедур `GU`, `GD`, `GR`, `GL` необходимо включить процедуру задержки. Для школьного алгоритмического языка мы оставим указанные процедуры без изменений, а реализуем эту идею в программах на других языках программирования. Отметим, что параметр процедуры задержки надо подбирать экспериментально, поскольку его величина зависит от быстродействия компьютера и порядка кривой Гильберта.

Язык Паскаль

При реализации алгоритма построения кривой Гильберта на Паскале возникает проблема, связанная с тем, что все четыре процедуры рисования кривых Гильберта (направленных в разные стороны) используют друг друга в качестве вспомогательных. Это не дает возможности соблюсти правило, согласно которому каждый идентификатор перед употреблением должен быть описан. Выходом является так называемое опережающее описание процедур, обращение к которым фигурирует раньше их описания [4]. Такими процедурами являются `GD` и `GU`.

```
Uses crt, graph;
const
del=5000;{Время задержки}
PATH='';
{Файлы *.BGI находятся в рабочем каталоге}
Var
  d,r :integer;
  n, orient :byte;
  x0, y0, S, h,Hscr,Wscr : word;
  PrS : real;
{Процедуры рисования связок. От последней
точки (на нее указывает графический курсор)
проводится вниз, вверх, влево, вправо отрезок
длиной h пикселей. Напомним, что ось Y
графического экрана направлена сверху вниз}
Procedure LineDown; begin Linerel(0, h) end;
Procedure LineUp;   begin Linerel(0, -h) end;
Procedure LineLeft; begin Linerel(-h, 0) end;
Procedure LineRight; begin Linerel(h, 0) end;
{Опережающее описание процедур GD и GU,
 вызываемых до своего определения}
Procedure GD(i: byte); forward;
Procedure GU(i: byte); forward;
{Процедуры рисования четырех разновидностей
кривых Гильберта}
Procedure GL (i: byte);
begin
 if i > 0 then begin
  GD(i-1); LineLeft;
  GL(i-1); LineDown;
  GL(i-1); LineRight;
  GU(i-1); Delay(del);
             end
end;
Procedure GR(i: byte);
begin
 if i>0 then begin
  GU(i-1); LineRight;
  GR(i-1); LineUp;
  GR(i-1); LineLeft;
  GD(i-1); Delay(del);
           end
end;
Procedure GU;
{Параметр i процедуры GU указан при
опережающем описании}
begin
 if i>0 then begin
  GR(i-1); LineUp;
  GU(i-1); LineRight;
  GU(i-1); LineDown;
  GL(i-1); Delay(del);
           end
end;
Procedure GD;
{Параметр i процедуры GD указан при
опережающем описании}
begin
 if i>0 then begin
  GL(i-1); LineDown;
  GD(i-1); LineLeft;
  GD(i-1); LineUp;
  GR(i-1); Delay(del);
           end
end;
Function Power2(n: byte): word;
{Возведение 2 в степень n}
var p,i: word;
begin
  p:=2; for i:=1 to n-1 do p:=p*2;
  Power2:=p
end;
BEGIN
 clrscr;{Чистка экрана}
 {Вводим исходные данные для построения
 кривой Гильберта}
 repeat
  write('Введите длину стороны опорного
            квадрата');
  write(' в % от высоты экрана ');
 readln(PrS);
until PrS<100;
write('Введите порядок кривой ');
readln(n);
repeat
 write('Введите ориентацию кривой ');
 write('вверх - 1, вниз - 2, вправо - 3,
        влево - 4 ');
 readln(orient);
until (orient>=1) and (orient<=4);
```

Окончание на с. 15

```
нач вещ N, M, x
  x: = температура в первой клетке коридора
  M: = x; N: = x
  |задание начальных значений в "ведре"
  вправо |переход в следующую клетку коридора
  нц пока снизу стена пока не вышли из коридора
    x:=температура получили очередной элемент
    N: = max(x, x + N) вычисление N нового
    M: = max(M, N) вычисление M нового
    вправо |переход в следующую клетку
  кц
  знач: = M |ответ
кон
```

И посмотрите, какой простой у нас получился опять алгоритм! Глядя на него, даже трудно поверить, что он считает такую "заковыристую" величину — не правда ли? По существу у нас в цикле всего две содержательных команды присваивания, а задача-то вначале казалась, ох, какой сложной!

И это все, я еще раз подчеркиваю, результат применения методов алгоритмизации. Когда вы знаете метод, умеете его применять, вы можете решать сложные и непонятные на первый взгляд задачи таким вот образом. И вся сложность уходит в метод, а алгоритм получается простой почти до предела.

Это, конечно, уже посложнее, чем корни квадратного уравнения. При добавлении очередного элемента мы должны разобраться с тем, что надо хранить в "ведре", понять, что делать с "ведром" и очередным элементом, чтобы двигаться дальше. Но тем не менее, действуя по общей схеме, мы через какое-то время ответ получим. И обратите внимание еще раз, что при этом мы думаем совсем не так и не над тем, над чем думали бы без применения метода. Мы вообще не думаем, как перебрать элементы, куда записать суммы и т.п. Мы думаем над переходом от старой "обработанной части" к новой. И в наших рассуждениях алгоритмическая и математическая составляющие задействованы обе одновременно. Потому-то я и говорю, что § 16 является сложным — здесь происходит переплетение логической и алгоритмической культур. Мы решаем алгоритмические задачи, думаем про действия и процессы, но наши рассуждения носят скорее логико-математический характер. Именно от этого задействования математической культуры и переплетения математической и алгоритмической составляющих и возникают сложности.

Возвращаясь к учебнику, замечу, что приведенные в нем задачи, как и большинство упражнений на однопроходные алгоритмы, достаточно просты. Рассмотренную выше задачу разумно давать в воспитательных целях сильным учащимся до изучения метода. Как правило, они напишут очень сложный алгоритм, несколько вложенных друг в друга циклов, естественно, с ошибками и т.п. Да и его они "добывают" с огромными мучениями. А когда после этого вы покажете метод и получившийся элементарный алгоритм, у них будет некоторый шок. И уж будьте уверены: теперь они в методе разберутся и решать задачи с его помощью научатся.

И еще одно замечание. Эта задача и предыдущая — с некоторым количеством стенок над Роботом (количество слов в строке) — примерно одинаковой сложности. Ключевая разница состоит в том, что в случае стенок школьники алгоритм в целом пишут. Их надо ловить, указывать, что они не учли одноклеточную стенку у правого края или одноклеточную стенку у левого края. Алгоритм в целом у них вроде есть, только иногда ("в особых случаях") не работает. А у задачи с суммами формулировка такая, что не надо ничего говорить про отдельные случаи, просто они ее не решают — и все. А вы можете показать, что эту задачу не только можно решить, но и более того, алгоритм получается очень простой, никакой алгоритмической сложности как бы нет.

Ну и, наконец, я еще раз скажу, что изложение в школьном учебнике — очень сильное упрощение. Собственно, все начинается и заканчивается на уровне "ведра". Если кого-то эта область интересует на более глубоком уровне, если кто-то хочет сам по-настоящему разобраться в этом методе или организовать факультатив для сильных ребят, то я рекомендую наш вузовский учебник "Программирование для математиков" [ПдМ], раздел "Индуктивное вычисление функций на пространстве последовательностей". Там этот метод алгоритмизации изложен целиком, со всеми деталями и подробностями, строго математически, с математическими доказательствами существования и единственности минимального однопроходного алгоритма для вычисления любой функции (т.е. для любой задачи), критериями проверки минимальности и т.п. Это все, конечно, требует гораздо более высокого уровня математической подготовки, но если доказательства пропустить, то все остальное вполне можно воспринять примерно в таком стиле, как здесь изложено, — без доказательств. Поэтому, если вы захотите этим методом овладеть более глубоко и по-нормальному, можете попробовать изучить эту книгу.

Если взять задачи, встречающиеся просто "по жизни" — на практике, то наиболее применимыми методами алгоритмизации, на мой взгляд, являются два — метод однопроходных алгоритмов и "инвариант цикла". Задачи на эти два метода встречаются чаще всего, это методы с наиболее широким классом применений.

Список литературы

[Авербух] *Авербух А.В., Гисин В.Б., Зайдельман Я.Н., Лебедев Г.В.* Изучение основ информатики и вычислительной техники. М.: Просвещение, 1992.

[ПдМ] *Кушниренко А.Г., Лебедев Г.В.* Программирование для математиков. М.: Наука, 1988.

[Звенигородский] *Звенигородский Г.А.* Вычислительная техника и ее применение. М.: Просвещение, 1987.

№ 8 (201) февраль 1999

еженедельное ПРИЛОЖЕНИЕ К ГАЗЕТЕ «ПЕРВОЕ СЕНТЯБРЯ»

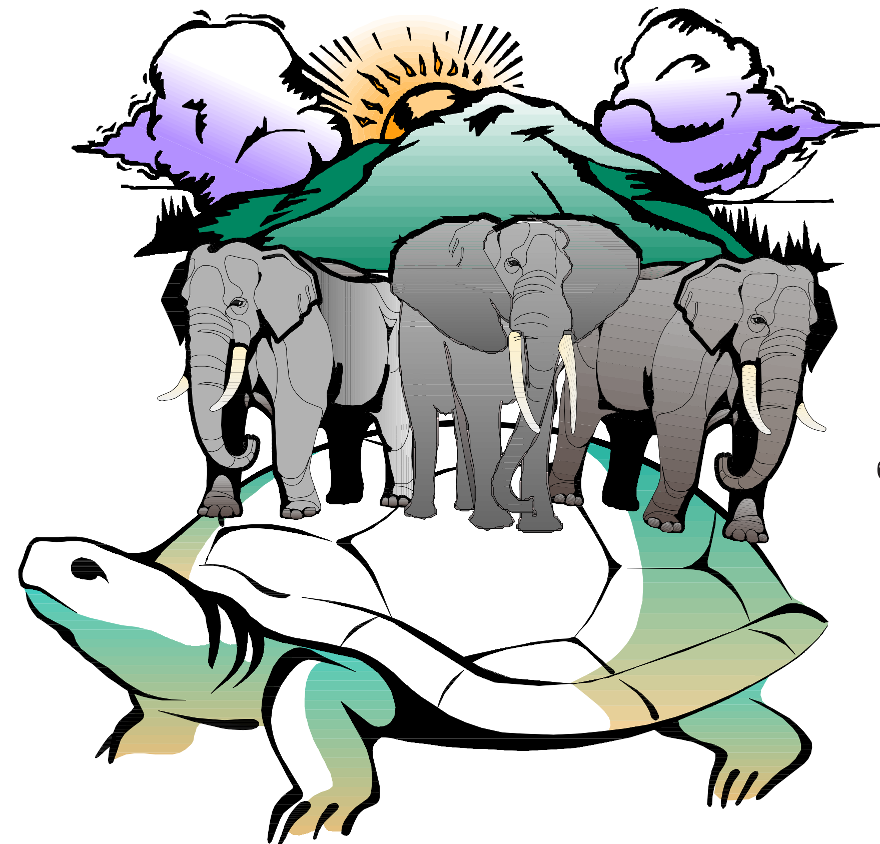
ИНФОРМАТИКА

12 лекций

о том, для чего нужен школьный курс информатики и как его преподавать

Лекции 6, 7—8

А.Г. КУШНИРЕНКО, Г.В. ЛЕБЕДЕВ



Введение, Лекции 1, 2, 3, 4, 5 были опубликованы в № 1, 3, 5, 6/99

Книга готовится к изданию в издательстве "Дрофа".
Выражаем признательность издательству "Дрофа" за содействие в подготовке публикации.

© ИнфоМир. Печатается с сокращениями.

СОДЕРЖАНИЕ

Предисловие
Введение
Лекция 1
A. Основные цели, или Три "кита" курса
A1. Главная цель курса – развитие алгоритмического стиля мышления
A2. Курс должен быть "настоящим"
A3. Курс должен формировать адекватное представление о современной информационной реальности
Лекция 2
B. Методика построения курса
B1. "Черепаха" курса – все познается через работу
B2. Проблемный подход
B3. Выделение алгоритмической сложности "в чистом виде"
C. Общий обзор учебника
C1. Распределение материала в учебнике
C2. Понятие исполнителя в курсе и учебнике
C3. Относительная важность и сложность материала в учебнике
C4. Несколько слов о месте курса в школьном образовании
Лекция 3
Введение
§ 1. Информация и обработка информации
§ 2. Электронные вычислительные машины
§ 3. Обработка информации на ЭВМ
§ 4. Исполнитель Робот. Понятие алгоритма
§ 5. Исполнитель Чертежник и работа с ним
Лекция 4
§ 6. Вспомогательные алгоритмы и алгоритмы с аргументами
§ 7. Арифметические выражения и правила их записи
§ 8. Команды алгоритмического языка. Цикл n раз
Лекция 5
§ 9. Алгоритмы с "обратной связью". Команда пока
§ 10. Условия в алгоритмическом языке. Команды если и выбор. Команды контроля
Лекция 6
§ 11. Величины в алгоритмическом языке. Команда присваивания 3
§ 12. Алгоритмы с результатами и алгоритмы-функции 5
§ 13. Команды ввода-вывода информации. Цикл для 7
§ 14. Табличные величины 8
§ 15. Логические, символьные и литерные величины 9
Лекции 7–8
§ 16. Методы алгоритмизации 12
16.1. Метод 1 — "рекурентные соотношения" 12
16.2. Метод 2 — "однопроходные алгоритмы" 18
16.3. Метод 3 — "инвариант цикла"
Лекция 9
§ 17. Физические основы вычислительной техники
§ 18. Команды и основной алгоритм работы процессора (программирование кода)
§ 19. Составные части ЭВМ и взаимодействие их через магистраль
§ 20. Работа ЭВМ в целом
Лекция 10
§ 21. "Информационные модели", или по-другому — "Кодирование информации величинами алгоритмического языка"
§ 22. Информационные модели исполнителей, или Исполнители в алгоритмическом языке
Лекция 11. Применения ЭВМ
§ 23. Информационные системы
§ 24. Обработка текстовой информации
§ 25. Научные расчеты на ЭВМ
§ 26. Моделирование и вычислительный эксперимент на ЭВМ
§ 27. Компьютерное проектирование и производство
§ 28. Заключение
Лекция 12
D. Заключение
D1. Методики преподавания курса
D2. Место курса в "большой информатике"
D3. Место курса в школе
D4. О программном обеспечении курса
E. Послесловие (разные замечания, отступления, рекомендации и пр.)
E1. Рекомендуемая литература
E2. Как возник Робот
E3. Как возник школьный алгоритмический язык
E4. История возникновения системы КуМир
E5. КуМир – внешние исполнители
E6. КуМир – реализация учебной системы с нуля
E7. КуМир – система "Функции и графики"
E8. КуМир – система "КуМир-гипертекст"
E9. КуМир – система "Планимир"
E10. Алгоритмы и программы. Алгоритмизация и программирование
Литература

а мы должны так "обработать" новую клетку, чтобы после обработки в "ведре" хранился максимум для новой пройденной части (т.е. для $A + x$). Другими словами, нам надо понять, как меняется M при переходе от A к $A + x$.

Рассмотрим ***все возможные*** "куски" внутри $A + x$. Эти куски можно разбить на две группы (рис. 6):

— "куски", не содержащие x, т.е. лежащие внутри A;

— "куски", содержащие x, т.е. начинающиеся где-то, а заканчивающиеся на x (допуская некоторую вольность, мы их будем обозначать S_{ix}).

Соответственно, новый максимум — это максимум среди всех S_{ij}, как не содержащих, так и содержащих x. Максимум среди всех S_{ij}, не содержащих x, — это максимум среди всех

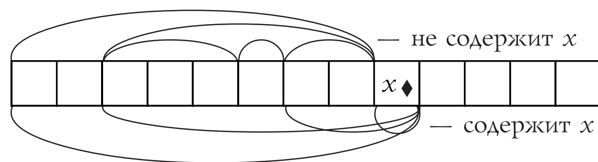


Рис. 6

S_{ij} в A, т.е. старое значение M, то, которое мы храним в "ведре" к началу обработки x. Максимум среди всех S_{ix}, содержащих x, — это величина новая, давайте, мы ее как-нибудь обозначим, например, N. Тогда

$$M_{новое} = \max(M_{старое}, N)$$

Теперь наша задача — узнать, как вычислить N, ***используя какие-нибудь характеристики, какую-нибудь информацию об A***, которую мы сможем добавить в "ведро" и хранить, подобно тому как в задаче про число максимумов мы "добавили" в ведро сам максимум.

Итак (***внимание!— это ключевое место в применении метода***), нам надо максимум среди S_{ix} (т.е. N) выразить через x и какие-то величины, зависящие только от A (от чего-то, что мы будем хранить в "ведре"). Для этого можно сами S_{ix} разбить на x и на оставшуюся часть — S_{iw}, где w — последний элемент в A, т.е. элемент перед x (*рис.* 7):

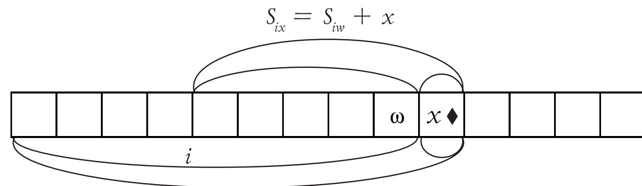


Рис. 7

Точнее говоря, таким образом представляются все S_{ix}, за исключением S_{xx} (при $i = x$). S_{xx} просто равно x и никакой "предшествующей" части не содержит. Таким образом,

$N = \max(S_{ix})$ для всех $i = \max(S_{xx}, S_{iw} + x)$ для всех $i =$

$= \max(x, S_{iw} + x)$ для всех $i = \max(x, NNx)$

где $NNx = \max(S_{iw} + x)$ для всех $i =$
$= x + \max(S_{iw})$ для всех i

Уследили, почему я выделил элемент x? Мы должны все выражать через то, что было раньше, должны придумать, что будем хранить в "ведре". Поэтому я отделил новый элемент x, чтобы выделить остаток, зависящий только от старой "пройденной части", от A.

И смотрите, что получилось: $\max(S_{iw})$ для всех i — это максимум среди всех S_{ij}, оканчивающихся на w, т.е. на последний элемент A. Но это просто "старое" (т.е. для A без x) значение N!

$$N_{старое} = \max(S_{iw}) \text{ для всех } i.$$

Тогда выведенные нами выше формулы перепишутся в виде:

$$M_{новое} = \max(M_{старое}, N_{новое})$$
$$N_{новое} = \max(x, NNx),$$

где $NNx = x + N_{старое}$

Или, если убрать NNx и поставить вычисление $N_{нового}$ до его использования:

$$N_{новое} = \max(x, x + N_{старое})$$
$$M_{новое} = \max(M_{старое}, N_{новое})$$

И — мы с удивлением можем это констатировать — теперь пара ($N_{новое}$, $M_{новое}$) выражается через пару ($N_{старое}$, $M_{старое}$) и очередной элемент x. Таким образом, если мы будем "хранить в ведре" ***всего два числа*** — N и M, то мы без труда за один проход вычислим искомую величину (M для всего коридора).

Итак, в "ведре" надо хранить M (это та величина, которую требовалось найти в задаче) и N — максимум из всех сумм S_{ij}, заканчивающихся на правом краю (на последнем элементе) соответствующей части коридора.

Самый существенный прием в этом методе состоит в том, что мы не пытаемся сразу подсчитать конечную величину для всего коридора целиком. Вместо этого мы рассматриваем процесс "прохода", перебора элементов и "шаг перехода" при обработке очередного элемента. Другими словами, рассматриваем нашу задачу сначала для коридора из одной клетки, потом для коридора из двух начальных клеток и т.д., пока, последовательно "удлиняя" пройденную часть, не получим искомое значение для всего коридора целиком. Поэтому наше внимание было сосредоточено не на том, как подсчитать столь непросто заданную величину, а на том, как она ***меняется***, когда мы "удлиняем" пройденную часть на очередной элемент x. Именно за счет этой смены акцентов нам и удалось так удивительно решить эту задачу.

Конечно, нам еще надо найти, чему равны M и N в первой клетке коридора (когда только одна клетка "пройдена"), но это уже совсем легко — для такого коридора, состоящего из одной клетки x, прямо по определению $M = x$ и $N = x$. Поэтому алгоритм в целом запишется так:

```
алг вещ максимальная подсумма
дано |Робот в начале коридора (рис. 4)
надо |Робот вышел из коридора в клетку B
     |(рис. 4)
     | знач = максимум из сумм температур
     | по всем "кускам" коридора от i до j,
     | для всех допустимых i и j
```

вому или правому краю (над А и над Б). Здесь аналогом "пробела" является "сверху свободно", а аналогом "не пробела" — "сверху стена". Это та же самая задача, сформулированная в терминах Робота.

Естественно, ее можно переформулировать и для числовых таблиц, например, считать аналогом "слова" группу подряд идущих не-нулей — тогда задача переформулируется в виде "подсчитать, сколько в данной таблице встречается групп подряд идущих не-нулей".

Важно, что как бы мы ни формулировали задачу — в символьных терминах, через стены на поле Робота или в терминах чисел в таблице, — это одна и та же задача, и проблемы у учеников возникают практически одни и те же.

Я эту задачу решать не буду, поскольку ее решение приведено в учебнике. Я лишь только еще раз обращаю ваше внимание, что если решать ее по наитию, без применения метода однопроходных алгоритмов, без анализа, что у нас за "ведро", что должно быть в нем в начале, как будет меняться его содержимое, то, как правило, решение получится с ошибками.

Трудная задача на составление однопроходного алгоритма.

Я же изложу задачу, которой в учебнике нет. Она мне очень нравится: это одна из самых красивых задач, хотя и достаточно сложная. Я показываю ее для вас (ибо вы должны знать больше и глубже, чем ваши ученики), а вы можете использовать ее при работе с сильными учениками — лидерами классов. Я ее сформулирую в терминах Робота, а вы сами переформулируете ее для таблиц, если захотите.

Итак, пусть у нас имеется горизонтальный коридор, в каждой клетке которого есть какая-то температура. Робот должен пройти по коридору (***конечно же, один раз!***) и подсчитать следующую интересную величину. Обозначим температуру в клетках коридора: t_1 — в первой клетке, t_2 — во второй и т.д. Рассмотрим ***все возможные*** "куски коридора" — от i-й клетки до j-й ($i < j$), и для каждого такого "куска" подсчитаем сумму температур S_{ij} в его клетках:

$$S_{ij} = t_i + t_{i+1} + ... + t_j.$$

Если всего клеток в коридоре n, то S_{1n} — сумма температур во всех клетках коридора. S_{33} — "сумма" температур в одной 3-й клетке, т.е. просто температура в 3-й клетке. S_{25} — сумма температур в клетках со 2-й по 5-ю включительно. Понятно? Так вот, рассмотрим все такие S_{ij} для всех возможных i и j и среди S_{ij} выберем максимальное число — это и есть та величина, которую нам надо найти.

Итак, еще раз, наша задача — составить алгоритм, вычисляющий максимум из всех возможных S_{ij}, где S_{ij} — сумма температур в "куске" коридора от i до j включительно. Другими словами, если мы рассмотрим ***все возможные*** "куски" коридора, в каждом "куске" просуммируем температуру всех клеток, а потом среди всех полученных сумм найдем максимальную, то это и будет то, что мы хотим найти. Если температура во всех клетках больше либо равна нулю, то ответ очевиден — это просто сумма всех температур во всех клетках. Но температура может быть и меньше нуля — и поэтому так просто задачу не решить.

Задача, на мой взгляд, замечательна тем, что по ее формулировке ни за что не догадаешься, как же, собственно, эту задачу решать, кроме как полным перебором всех случаев — прямо по определению. Но это значит, что надо писать алгоритм, который будет рассматривать все диапазоны, запоминать где-то вычисленные суммы, потом все эти суммы перебирать и искать максимум. Такой громоздкий и неэффективный алгоритм еще можно составить. Но запретите школьникам использовать в алгоритме таблицы — и они вообще не смогут эту задачу решить!!! А я вам сейчас покажу, как ее решить за один проход и очень просто! Как обычно, когда применяются методы алгоритмизации, пройдем один раз по коридору, пособираем что-то такое в "ведре" — и получим ответ!

> ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я специально вам привожу такую формулировку, чтобы показать огромную мощь применения метода. Это касается всех излагаемых в учебнике методов — овладение ими действительно дает огромную мощь. На простых задачах (вроде поиска максимального элемента) это не так заметно — там требуемый алгоритм можно придумать и "так", без знания и целенаправленного применения методов алгоритмизации. Но есть задачи, в которых просто так, без применения метода, догадаться, какой будет ответ, по-моему, нельзя. По виду последней задачи ни за что не скажешь, что ответ можно получить за один "проход", просмотрев клетки коридора только один раз. Именно поэтому она так хороша для демонстрации действенности метода однопроходных алгоритмов даже в его школьной ("ведерно-рыбной") формулировке. Эта задача показывает, что жизнь отнюдь не сводится к умению писать циклы и условия.

Решим эту задачу. Вспомним, в чем состоит метод однопроходных алгоритмов. Прежде всего мы должны посмотреть на все это, как на последовательный процесс. Пусть мы уже прошли какую-то часть коридора (A) и обрабатываем очередной элемент — температуру x в клетке, где сейчас стоит Робот. Саму эту клетку мы будем обозначать также буквой x (см. *рис.* 5).

x
Б
пройденная часть (A)

Рис. 5

Итак, часть коридора (A) мы прошли и стоим в очередной клетке x с температурой x. В соответствии с методом мы должны постараться выразить ответ для новой пройденной части ($A + x$), т.е. понять, каков максимум S_{ij} среди всех "кусков" от i до j, если рассматривать ***только*** новую пройденную часть ($A + x$).

Обозначим этот максимум M и будем считать, что он хранится в "ведре". Это значит, что, когда мы переходим в новую клетку (x), в "ведре" (т. е. в M) хранится максимум S_{ij} для старой пройденной части последовательности (для A),

Лекция 6

Ну а теперь мы приступаем ко второй части главы 1 "Алгоритмический язык". Вся первая часть была связана с действиями, с записью действий на алгоритмическом языке, с конструкциями алгоритмического языка, позволяющими структурировать действия (вспомогательные алгоритмы) либо задавать порядок выполнения действий (управляющие конструкции — циклы и ветвления). Теперь мы переходим ко второй части, ко второму фундаментальному понятию информатики — понятию объекта, называемому в школьном курсе "величиной".

Термин величина, который используется в школьном учебнике и в алгоритмическом языке, ввел Андрей Петрович Ершов, заимствовав его из математики и физики и считая, что так школьникам будет проще для понимания. Обычно в программировании используется термин переменная, который, впрочем, сейчас все чаще вытесняется термином объект. Термин переменная, в свое время тоже заимствованный из математики, уже начал уходить в прошлое. Сейчас чаще говорят про объект и состояние объекта, а также — более широко — про объектно-ориентированный подход в программировании. Но в школьном курсе вместо слов "объект" и "состояние" объекта используются термины величина и значение величины. Дальше я буду пользоваться терминологией школьного учебника.

§ 11. Величины в алгоритмическом языке. Команда присваивания

Методика введения величин — проблемный подход.

Как всегда при проблемном подходе, все начинается с постановки задачи, для решения которой нам понадобятся величины. Задача, рассматриваемая здесь, такова: Робот расположен вплотную над горизонтальной стеной неизвестных размеров, расстояние от Робота до краев стены неизвестно, надо провести Робота "сквозь стену" на одну клетку вниз. Поскольку Робот сквозь стены ходить не умеет, т.е. просто скомандовать **вниз** нельзя, то надо заставить Робота эту стену как-то обойти.

И, как всегда, основным методическим приемом на старте выступает противопоставление схемы программного управления непосредственному управлению Роботом "вручную". Если поставить школьникам задачу "вручную" прокомандовать Роботом (при условии, что размеры стены им неизвестны, т.е. учитель им поле и стену не показывает), то довольно быстро они сообразят, что нужно идти к одному из краев стены, например, вправо, по дороге все время спрашивая **снизу стена?** и считая число сделанных шагов. А когда стена кончится, скомандовать **вниз** и сделать **влево** столько шагов, сколько перед этим сделано шагов **вправо**. Если над стеной Роботу 7 раз было скомандовано **вправо**, то после шага вниз ему надо 7 раз скомандовать **влево**, чтобы он попал в нужную клетку.

Когда школьники все это сообразят и будут в состоянии решить задачу при управлении Роботом "вручную", следует обратить их внимание, что при этом была задействована их — школьников — голова: они запоминали число сделанных **вправо** шагов. Если теперь от управления Роботом "вручную" перейти к написанию алгоритма, который потом будет выполнять ЭВМ (т.е. перейти к схеме программного управления), то нам необходимо както записать в алгоритме, что ЭВМ ***должна*** запоминать число выполненных Роботом команд **вправо** и использовать эту информацию при обратном движении влево. Это и есть тот новый материал, которому посвящен параграф.

После того, как проблема запоминания информации в памяти ЭВМ понята и осознана, вы можете в зависимости от вашего настроения либо просто повторить модель памяти ЭВМ и ввести понятие величины, либо (прежде чем излагать материал параграфа) дать возможность ученикам придумать свои формы записи для задания величин.

Вообще говоря, "придумывание" таких форм записи можно начинать и раньше. Вернемся на минутку к вспомогательным алгоритмам "квадрат 4", "квадрат 6" и т.д. Уже тогда можно было просить школьников придумать что-нибудь для задания алгоритма "квадрат" с произвольной длиной стороны. Поскольку какая-то математическая культура у них есть, они обычно длину стороны как-то обозначают, после чего совершенно естественно возникает запись **<u>арг</u> <u>вещ</u> а**.

Здесь все то же самое, только сначала лучше напомнить ученикам про модель памяти ЭВМ в виде классной доски: когда начинает выполняться какой-то алгоритм, в памяти под него отводится место. Тогда дальше, поскольку форма описания аргументов алгоритмов уже была, ученики вполне могут придумать не только смысл величин (отведение памяти внутри прямоугольника алгоритма), но и даже более или менее правильную форму их описания.

Соответственно, именно здесь появится понятие промежуточной величины — величины, которая нам нужна "на время" для подсчета числа сделанных шагов. Здесь же можно детально изложить, как ЭВМ отводит место в памяти для промежуточных величин.

Проблемный подход и команда присваивания.

А вот дальше, для перехода к команде присваивания, очень полезно попросить учеников снова решить исходную задачу, но уже с использованием нарисованного на доске прямоугольника для величины **<u>цел</u> n**, в которой должно подсчитываться число сделанных Роботом шагов. Это будет такой переходный вариант, когда ученикам придется выполнять работу ЭВМ и вместо подсчета числа шагов в уме они вынуждены будут вписывать, стирать и снова вписывать значение числа шагов внутрь прямоугольника **<u>цел</u> n**. Первый шаг вправо сделали — записали в прямоугольник на доске "1", сделали второй шаг — "1" стерли и написали "2" и т.д.

После этого уже можно говорить, что, как мы видели, для работы с памятью ЭВМ надо уметь не только выделять кусок памяти, но и узнавать (читать из памяти) информацию, которая в ней хранится, а также стирать старое значение и записывать новое.

Получается всего три операции: отведение места в памяти, чтение из памяти, запись новой информации в память.

Первые две из этих трех операций почти ничем не отличаются от работы с аргументами алгоритмов и де-факто уже знакомы школьникам. А для записи информации в память в алгоритмическом языке (как и во всех процедурных языках программирования) есть специальная команда — команда присваивания (пп. 11.6—11.7).

Поскольку собственно с величинами (переменными) и командой присваивания на примере того или иного языка программирования все вы прекрасно знакомы, я лишь расставлю некоторые акценты и поясню то, что специфично для нашего курса.

Использование модели памяти ЭВМ.

Все, что касается величин, можно и нужно излагать и пояснять на модели памяти ЭВМ. Как я уже говорил, "доска" — чрезвычайно точная модель, которая в деталях соответствует тому, как ЭВМ работает со своей памятью.

В частности, обязательность ***описания*** величины связывается с необходимостью в какой-то момент нарисовать прямоугольник этой величины. Говорится, что компьютер, начав выполнять алгоритм "сквозь стену", отведет место для этого алгоритма ("нарисует большой прямоугольник"), после чего, встретив строку **нач цел n**, внутри большого прямоугольника алгоритма "сквозь стену" отведет место ("нарисует маленький прямоугольник") для целочисленной величины **n**. Именно поэтому величину необходимо описывать в строке **нач**.

Прямоугольник величины рисуется внутри прямоугольника алгоритма, откуда следует, что в разных алгоритмах можно использовать одно и то же имя, но промежуточные величины ***будут разными.*** Это свойство промежуточных величин в "большой" информатике называется ***локализацией***: промежуточные величины ***локализованы*** в алгоритме по тексту (то же имя в другом алгоритме обозначает другую величину) и по времени существования (прямоугольник величины рисуется в начале выполнения алгоритма и стирается по завершении выполнения алгоритма).

Использование значения величины в терминах модели памяти объясняется так же, как и использование аргументов: встретив имя величины, ЭВМ вместо имени возьмет и подставит значение из прямоугольника величины.

Изложение команды присваивания также происходит на базе модели памяти: сначала ЭВМ вычисляет значение выражения в правой части команды присваивания, подставляя при необходимости вместо имен величин их значения, а затем старое значение величины стирается и внутрь прямоугольника записывается новое. (Это, пожалуй, единственный недостаток модели — в реальности внутри ЭВМ эта одна операция: при записи нового значения "внутрь прямоугольника" величины старое значение автоматически пропадает. В нашей же модели мы сначала "стираем", чего в реальности не делается, а лишь потом записываем.)

Понятия, связанные с понятием величины.

Очень важным является п. 11.2, где вводятся такие характеристики величины, как ***имя, значение, тип*** (целочисленная, вещественная и пр.). При желании вы можете прямо здесь рассказать еще и про ***вид*** величины (аргумент, результат, промежуточная величина алгоритма, общая величина исполнителя). Все ***типы*** и ***виды*** величин алгоритмического языка перечислены на третьей странице форзаца. Четверка ***имя, значение, тип, вид*** — это ***полный*** набор понятий, связанных с величинами. И ученики должны ***мгновенно*** отвечать на вопросы типа: "каково ***имя*** этой величины?", "какого ***типа*** эта величина?", "какого ***вида*** эта величина?". Заметьте также, что ***имя, тип*** и ***вид*** величины фиксированы и определяются по тексту алгоритма, а ***значение*** величины может меняться в ходе выполнения алгоритма. Это все простые понятия, но важные для дальнейшего. И обратите внимание, что мы не случайно в нашей модели памяти ЭВМ пишем ***имя, тип*** и ***вид*** величины над прямоугольником величины (внутри которого пишется ***значение*** величины).

Разделение труда между ЭВМ и исполнителями.

По-прежнему мы считаем существенным и продолжаем подчеркивать разделение труда между ЭВМ и исполнителями. Важно понимать, что понятие промежуточной величины, выполнение команд присваивания, вообще все действия с величинами и их значениями осуществляет ЭВМ. Исполнители по-прежнему не умеют выполнять ничего, кроме своих команд. Чтобы прояснить это, на с. 90 учебника приведена полная последовательность конкретных команд, которые ЭВМ выдаст Чертежнику в процессе выполнения алгоритма рисования параболы А45.

С другой стороны, такое разделение, на наш взгляд, способствует усвоению собственно конструкций алгоритмического языка, их смысла, пониманию того, что и как делает ЭВМ в ходе их выполнения.

Фактически этим все и исчерпывается. Параграф, кроме того, содержит несколько примеров алгоритмов, работающих с величинами. Помимо очень простых алгоритмов "вниз сквозь стену" и "закрасить радиоактивные клетки", здесь рассматривается и вполне содержательная задача рисования графика функции на примере параболы.

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Мы сделали все, чтобы этот переход от алгоритмов без величин к алгоритмам с величинами прошел незаметно, как простая вещь. Модель памяти была введена в § 6 при описании аргументов алгоритмов. Поэтому здесь она должна казаться знакомой: просто эта модель применяется для описания нового явления — работы ЭВМ с промежуточными величинами. Использование имени величины в выражениях ничем не отличается от использования имен аргументов, что тоже уже было, причем с тем же самым смыслом. Единственное, что появилось действительно нового, — это команда

максимумов в новой пройденной части коридора (включая x). Первое, что нам нужно, — это старый максимум, максимум в пройденной части коридора (без x). Если x меньше этого максимума, то число максимумов не меняется. Если x равен этому максимуму, то число максимумов надо увеличить на единицу. Если x больше этого максимума, то x становится новым максимумом и он единственный, т.е. число максимумов равно 1. Обозначим k — число максимумов, а m — сам максимум. Тогда переход от старой пройденной части (без x) к новой пройденной части (включая x) запишется так:

```
выбор
  | при x < m : (ничего не меняется)
  | при x = m : k : = k + 1
  | при x > m : m: = x, k : = 1
все
```

Таким образом, для этой задачи надо хранить в "ведре" не одно, а два числа: старый максимум и количество этих максимумов, m и k. Встретив очередной элемент x и сравнив его со старым максимумом m, мы можем изменить при необходимости и сам максимум m, и количество максимумов k. И к тому моменту, когда мы выйдем из коридора, в "ведре" будет ответ k. Причем за один проход!

Еще раз подчеркну, в чем же, собственно, состоит метод. Метод состоит в том, чтобы при вычислении какой-то величины, например числа максимумов в таблице или в коридоре, обрабатывать элемент за элементом последовательно и при этом всякий раз при обработке очередного элемента анализировать и менять какую-то не очень большую информацию, которую мы храним и обновляем с самого начала ("в ведре").

Ключевой вопрос метода — что такое это "ведро", какую информацию мы должны хранить в нем. Надо рассмотреть процесс обработки очередного элемента x, процесс "удлинения" "просмотренной части" на элемент x, и понять, какая информация нужна для получения ответа на вопрос. Это и есть метод однопроходных алгоритмов. После того как вопрос с "ведром" решен, сам алгоритм, как правило, записывается очень легко. Для числа максимумов в таблице алгоритм А85 приведен на с. 131 учебника.

В однопроходном алгоритме цикл всегда один — элементы перебираются один раз. Если мы ведем Робота по коридору, то получается один цикл **пока**, если перебираем элементы таблицы, то один цикл **для**, в котором перебираются все подряд элементы таблицы.

Однопроходные алгоритмы для работы со строками.

Я еще раз подчеркну, что не важно, как именно мы получаем элементы. Слово "однопроходный" означает лишь, что перебор осуществляется один раз. Например, можно перебирать одну за другой буквы в литерной строке. В учебнике разбирается один подобный алгоритм — для подсчета количества слов в строке. Строка состоит из символов, и перебирать мы будем символы. Слово "однопроходный" означает, что мы должны составить алгоритм, перебирающий символы один раз. Рассмотрим, например, строку "Я (два пробела) завтра (четыре пробела) пойду (пробел) в (два пробела) кино (три пробела)!":

Я_ _завтра_ _ _ _пойду _ в_ _кино_ _ _!

В этой строке 6 слов. (Заметьте, что восклицательный знак "!" — отдельное слово. "Словом" мы здесь называем *любую* последовательность символов (не пробелов) между пробелами или между пробелами и границей строки.)

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я уже говорил, что до изложения и демонстрации применения метода полезно дать задачку, чтобы школьники "помучились", решая ее "как-нибудь". Задача про число слов — очень хороший пример.
>
> Если вы дадите такую задачу, то большинство школьников какой-то однопроходный алгоритм напишет. При этом, однако, решения почти у всех будут с ошибками. Вот типичные ошибки:
> — алгоритм "пропускает" слово из одного символа в начале строки (в нашем примере — слово "Я") или
> — алгоритм "пропускает" слово из одного символа в конце строки (в нашем примере — слово "!"). Причем если ученик допустил первую ошибку и вы просите его ее исправить, то, как правило, он устраняет первую ошибку и делает вторую. И наоборот. Я это знаю по опыту.
>
> Обычно школьники либо ловят перепад "символ—пробел" и считают, что количество таких перепадов и есть количество слов (и тогда они допускают ошибку второго рода — "пропускают" слово в конце), либо ловят перепад "пробел—символ" — и, соответственно, пропускают слово в начале. Поняв наконец, в чем ошибка, они обычно пытаются дописать "если" в начале или в конце алгоритма, чтобы учесть и крайние слова. Но после этого обычно алгоритм перестает работать, если строка начинается или кончается группой пробелов. В конце концов в попытках исправить ошибки школьники добавляют такое количество разных "если" и "циклов", что алгоритм становится совершенно запутанным и ни они сами, ни кто-то другой уже не в состоянии понять, как он работает. А главное — на некоторых строках алгоритм все равно дает неверный ответ (т.е. по-прежнему работает с ошибками).
>
> После этих мучений изучение метода однопроходных алгоритмов и решение задачи с его помощью кажется облегчением и обычно проходит на ура.

Эту же задачу можно переформулировать для Робота. Робот расположен в левой клетке некоторого горизонтального ряда клеток, ограниченного стеной справа и слева, а над этим рядом расположено какое-то количество горизонтальных стенок (*рис.* 4):

А Б

Рис. 4

Надо перевести Робота из А в Б (слева направо) и посчитать число стенок над Роботом. Обычно школьники "пропускают" одноклеточные стенки, прилегающие к ле-

выловит самую большую рыбу. Простой алгоритм: выловили новую рыбу — сравниваем с той, которая у нас в ведре. Если новая рыба больше, то старую выпускаем, а новую кладем в ведро. Если новая рыба меньше — мы ее просто выпускаем и т.д.

Вообще все это описание нужно для того, чтобы ввести слова-аналогии, например, “ведро” — для обозначения места, где мы что-то храним (набора величин алгоритмического языка).

Рассмотренный выше пример-аналогия вошел в учебник с целью пропаганды бережного отношения к живой природе: чтобы рыба по ходу соревнования не пропадала, каждому участнику выдается ведро, в котором он хранит самую большую из пойманных рыб, а все остальные (“ненужные”) рыбы в целости и сохранности выпускаются обратно в реку.

Я приведу сейчас еще один — менее “экологичный” — пример-аналогию, который в свое время А.Г. Кушниренко услышал от своего учителя (К.В. Ким), — пример с собиранием грибов.

Итак, та же задача на нахождение максимума (в примере — поиск самого большого гриба). Аналогия звучит так. Человек пошел в лес за грибами. Идет с корзинкой по лесу, по тропинке, и ищет грибы. Найдя новый гриб, сравнивает его с тем, который лежит в корзинке. Если новый гриб больше, то старый (меньший) из корзинки выбрасывается, а новый (больший) кладется в корзинку. Привлекательность этой аналогии в том, что человек идет, “совершает один проход” по лесной тропинке.

В учебнике, повторю, чтобы не разбрасывать бессмысленно вполне годные к употреблению грибы по лесу, мы заменили это собирание грибов на ловлю рыбы, хотя при ловле рыбы мы никуда не идем, а сидим на берегу. Но мы решили, что природу беречь в данном случае важнее, чем идти.

Ключевое свойство однопроходного алгоритма.

В нашей аналогии (в учебнике) в каждый момент происходит сравнение вновь выловленной рыбы с той, что хранится в ведре. Меньшая отпускается, а большая помещается в ведро. Понятно, что при этом к моменту, когда соревнования закончатся, у нас в ведре будет самая большая из пойманных рыб. Алгоритм поиска максимума заключается в том, что, вылавливая очередную рыбу, или получая очередное число от Робота, или взяв очередной элемент таблицы, мы сравниваем его с тем максимумом, который уже нашли до этого момента (с тем, что “хранится в ведре”). И аналогия в основном придумана именно для того, чтобы ввести понятие ведра — т.е. места, где мы что-то храним.

Итак, первое, на что нужно обратить внимание, — процесс, в котором к нам поступают очередные элементы: или Робот идет по коридору и узнает очередные значения радиации; или мы перебираем элементы таблицы; или вылавливаем рыбу за рыбой.

Во всех случаях есть процесс (“проход”), когда мы получаем элемент за элементом, число за числом. Мы сидим с ведром на берегу и вытаскиваем очередную рыбу, или идем с корзинкой по лесу и находим очередной гриб, или читаем очередной элемент таблицы — с точки зрения информатики это просто получение новой порции информации.

Кроме процесса, есть нечто, что в аналогии учебника можно назвать “ведром”, — место, где мы что-то храним. Получив новую порцию информации (рыбу, гриб), мы смотрим на нее и на содержимое “ведра” и что-то делаем с содержимым ведра, с нашим хранилищем. Скажем, меняем рыбу в ведре на большую. Не обязательно, чтобы в “ведре”, в хранилище, было только одно число — там может быть и десяток чисел. Важно другое: всякий раз, получив очередной элемент (“рыбу”), мы что-то проделываем с содержимым нашего “ведра”.

Здесь важны обе сущности, с которыми мы работаем, — и “рыба”, и “ведро”. Мы получаем очередное число из нашей последовательности, вылавливаем очередную рыбу, узнаем очередное значение радиации у Робота или очередной элемент таблицы. Это одна часть. Далее мы сравниваем этот очередной элемент с чем-то, что мы храним, — с содержимым ведра или корзины, быть может, меняем то, что мы храним, и после этого переходим к следующему элементу.

Очень важно заметить, что при получении очередного элемента мы рассматриваем только сам этот элемент (“рыбу”) и содержимое нашего хранилища (“ведро”), ***но никогда не рассматриваем предыдущие элементы***. Выловив очередную рыбу, мы смотрим только в “ведро”, а каких мы ловили рыб до того, нам уже не интересно, мы это можем забыть. Это самая важная составляющая, ключевое свойство всякого однопроходного алгоритма.

Соответственно, самое существенное в составлении однопроходного алгоритма состоит в придумывании “ведра” — того, что мы будем хранить.

Для задачи поиска максимума (а это простейшая из возможных задач) ответ очевиден: если мы хотим найти максимальное значение, то именно его и надо хранить в “ведре”. В алгоритмическом языке информация хранится в величинах. Поэтому с алгоритмической точки зрения “ведро” — это какой-то набор величин алгоритмического языка. Для хранения максимума, например, можно завести величину **вещ v**. Максимум найти, таким образом, легко (алгоритм А80). Прошли один раз, нашли максимум.

А как найти число максимумов? Здесь — ключевой момент, ключевой пример для понимания однопроходных алгоритмов. Давайте порассуждаем в терминах “ведра”. Итак, у нас будет некоторое “ведро” (и пока мы не знаем, что в нем будет храниться). Пусть мы уже “прошли” какую-то часть коридора и получили значение радиации в очередной клетке, получили очередное значение *x*. Нам нужно сказать, сколько максимумов в пройденной части коридора, включая x (*рис.* 3).

x♦

пройденная часть (без *x*)

Рис. 3

Давайте разберемся, что нам для этого нужно помнить и знать, какую информацию о пройденной части коридора (без x) нам надо хранить, чтобы ответить на вопрос, сколько

присваивания, которая позволяет устанавливать и изменять значение величины. Вот, собственно, и все. Замечу еще, что термин присваивание — пожалуй, единственный, который при всей своей странности перекочевал в школьный курс — в первые учебники А.П. Ершова — в неизменном виде (что, видимо, связано с полным отсутствием какого-либо аналога для этого термина в математике и физике).

Упражнения

Естественно, что после введения величин класс задач, которые мы можем решать, существенно расширяется.

Упражнения к параграфу можно разделить на две части. Первая часть (упр. 1—4) — просто отработка команды присваивания, т.е. упражнения типа “до команды присваивания значения величины были такие-то, скажите, какими они будут после выполнения команды” или, наоборот, “после команды стали равны тому-то, скажите, какими они были до”. Очень хорошее упражнение здесь — это упражнение 4, где из четырех вариантов обмена значений величин *x* и *y* надо указать, где в результате значения действительно будут обменены, а где — нет. Эти упражнения очень просты и нацелены просто на лучшее усвоение команды присваивания и понятия величины.

Упражнения 5 и 6 носят промежуточный характер — алгоритм уже задан, надо понять, как он будет работать в заданной ситуации (упр. 5) или что он вообще делает (упр. 6). В упражнении 5 основное, что должны сделать ученики, — сказать, чему будет равно значение величины **n** в момент окончания работы алгоритма (или отказа).

Вторая часть (упр. 7—9) — нормальные задачи на составление алгоритмов, в которых для решения задачи надо использовать величины и команду присваивания. Решения всех этих задач вы можете найти в методическом пособии [Авербух]*.

Повторю, что в целом этот параграф очень важный — он вводит второе фундаментальное понятие информатики из той четверки, о которой я вам говорил. Это второе фундаментальное понятие, понятие объекта (величины), будет потом более глубоко раскрыто в § 14 (табличные величины). Но в силу проведенной подготовительной работы (модель памяти ЭВМ, работа с аргументами) сам материал параграфа оказывается достаточно простым, во всяком случае, намного проще, чем циклы с **если** внутри, которыми закончился § 10.

§ 12. Алгоритмы с результатами и алгоритмы-функции

Этот параграф идет вслед за введением величин, но посвящен совсем другому — углублению третьего фундаментального понятия информатики, понятия вспомогательного алгоритма (подпрограммы с параметрами).

* Начиная с этого номера для ссылок мы будем использовать условные названия книг.

В § 6 были введены понятия вспомогательного алгоритма и вспомогательного алгоритма с аргументами. Здесь вводятся понятие ***результатов*** вспомогательного алгоритма и понятие ***алгоритмов-функций***. И то и другое позволяет осуществить аналог “обратной связи” — передать в основной алгоритм информацию, полученную в результате выполнения вспомогательного алгоритма.

МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Давайте, я еще раз бегло пройду по этой четверке фундаментальных понятий информатики.

Первое понятие — команды, и прежде всего циклы. Это § 4 (команда) и 8—9 (команды циклов). Все остальное (команда вызова, команда присваивания, команды ветвления и пр.) лишь разнообразит наше представление о том, какие бывают команды, но практически уже ничего не добавляет к фундаментальной сущности первого понятия информатики. Итак, это понятие можно считать полностью пройденным.

Второе понятие — объекты (величины), и прежде всего таблицы. Это § 11 (понятие величины) и 14 (табличные величины).

Третье понятие — вспомогательный алгоритм (подпрограмма) с параметрами. Это § 6 и § 12.

Наконец, четвертое понятие — понятие исполнителя — появится только в § 22.

ЕЩЕ ОДНО МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. То, что материал этого параграфа идет сразу вслед за величинами, отнюдь не случайно. С одной стороны, как я уже говорил, мы постарались все связанное с вспомогательными алгоритмами максимально приблизить к началу, чтобы это понятие по возможности глубже “впечаталось” в память учеников. С другой стороны, изложить понятие результатов алгоритма до величин невозможно, поскольку невозможно ни записать, чему должен быть равен результат, ни объяснить, куда он попадет. Эти два условия (максимально рано, но после величин) и определяют место темы “результаты алгоритмов” в нашем курсе.

Параграф в целом можно разбить на две части:

1) ***виды*** величин и алгоритмы с результатами (пп. 12.1—12.7), включая общие правила выполнения команды вызова вспомогательного алгоритма (п. 12.4);

2) алгоритмы-функции (пп. 12.8—12.11).

Между этими частями имеется достаточно сильное неравенство. Если первая часть носит фундаментальный, базовый характер, то вторая — про алгоритмы-функции — никаких фундаментальных сущностей не содержит и вообще может быть опущена без особого ущерба для алгоритмической культуры учащихся. Но — давайте по порядку.

Алгоритмы с результатами.

Ввести и объяснить алгоритмы с результатами более или менее просто, и ситуация здесь полностью повторяет ситуацию с введением команд обратной связи: если раньше с помощью аргументов мы передавали информа-

цию из основного алгоритма в вспомогательный, то теперь, используя результаты, можем передать в основной алгоритм информацию из вспомогательного.

Дальше, как всегда, вводится форма записи результатов (которая ввиду ее полной аналогичности записи аргументов никаких вопросов вызывать не должна) и объясняется, как будет работать ЭВМ (на модели памяти ЭВМ с рисованием соответствующих прямоугольников для результатов и пр.). Поскольку объяснять все это надо на каком-то примере, то используется алгоритм А47, который по значениям катетов прямоугольного треугольника вычисляет длину гипотенузы. Соответственно, объясняется, что результаты вспомогательного алгоритма копируются внутрь величин, указанных в вызове, перед завершением работы вспомогательного алгоритма и стиранием его прямоугольника из памяти ЭВМ.

С учетом этого уточнения мы уже получаем полную, завершенную схему выполнения вспомогательного алгоритма. Если нам нужно передать значения от основного алгоритма к вспомогательному, то используются аргументы, если от вспомогательного к основному — то результаты. Пользуясь аргументами и результатами, мы можем устроить любой обмен информацией между основным и вспомогательным алгоритмами.

Эта полная схема и просуммирована в п. 12.4, где изложено все, что мы прошли про вспомогательные алгоритмы. Кроме того, здесь — единственное место в учебнике, где написано, что на время выполнения вспомогательного алгоритма выполнение основного приостанавливается. Мы это неявно подразумевали, но нигде раньше даже не обсуждали. Здесь же, в п. 12.4, собраны воедино все правила, описывающие выполнение вспомогательного алгоритма и его связь с основным.

В этот момент, после п. 12.4, все уже пройдено. Теперь мы можем решить любую задачу, при необходимости передав информацию и от основного алгоритма вспомогательному, и обратно.

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я обращаю ваше внимание, что в алгоритме "гипотенуза" уже не используются ни Робот, ни Чертежник. И таких алгоритмов, начиная с § 12, будет понемногу становиться все больше и больше. Помните график роли понятия исполнитель в нашем курсе (с. 47), где к § 12 исполнители "сходят на нет"? Мы действительно прошли уже все необходимое для работы без исполнителей, для записи чисто информационных алгоритмов, которые выполняет только ЭВМ.
>
> Методическая роль исполнителей, позволивших плавно вводить конструкции алгоритмического языка и решать задачи разного уровня, полностью исчерпана.
>
> Позже мы вернемся к исполнителям, но это будут уже "совсем другие" исполнители, точнее, они будут играть совсем другую роль.

С задачами по этой теме ситуация следующая. Все задачи после параграфа формулируются в духе "написать алгоритм с такими-то аргументами и такими-то результатами". К сожалению, объем школьного курса не позволяет дойти до задач, в ходе решения которых эти алгоритмы нужно будет самим придумывать и вводить. В идеале школьники должны были бы овладеть не только понятием вспомогательного алгоритма с результатами, но и научиться самостоятельно выделять эти вспомогательные алгоритмы в задачах, формулировать их аргументы, результаты, **дано** и **надо**. Но для этого необходимо значительно увеличить сложность решаемых задач так, чтобы каждую пришлось разбивать на подзадачи, для которых и будут составляться вспомогательные алгоритмы. Поэтому в упражнениях после параграфа мы ограничились задачами на составление алгоритмов с уже заданными аргументами и результатами.

Алгоритмы-функции.

С алгоритмами-функциями ситуация осложняется тем, что невозможно сформулировать задачи, которые было бы нельзя решить, не применяя алгоритмы-функции. Любой алгоритм, использующий вспомогательный алгоритм-функцию, всегда можно записать и без алгоритмов-функций, заменив их вспомогательными алгоритмами с результатами.

Вместо команды присваивания **a:= f(x)**, где **f** — алгоритм-функция, мы всегда можем написать вызов **F(x,a)**, где **x** — аргумент, а **a** — результат. Всюду, где применяются алгоритмы-функции, можно заменить их на обычные алгоритмы с результатами.

Конечно, функции в ряде случаев удобнее. Сравните, например, эти две записи:

С использованием алгоритмов-функций	С использованием алгоритмов с результатами

`a:= f(x1)+f(x2)`	`F(x1, a1)`
	`F(x2, a2)`
	`a:= a1 + a2`

Вы можете смотреть на этот пример как на обоснование необходимости алгоритмов-функций. Алгоритмы-функции удобны тем, что мы можем сразу записать их в выражение и за счет этого упростить запись.

Но поскольку речь идет всего лишь об упрощении записи, то никакой фундаментальной сущности в алгоритмах-функциях нет. Это исключительно и только особая форма записи вспомогательного алгоритма с результатом. По существу же при использовании и алгоритмов-функций, и алгоритмов с результатами происходит одно и то же: из основного алгоритма вспомогательному передаются аргументы, обратно передается результат. Фундаментальная сущность у алгоритмов с результатами и алгоритмов-функций совпадает. Все отличия в форме записи и в особенности в форме использования (в форме вызова).

Поэтому подход у нас к алгоритмам-функциям такой же, как и к команде выбора, т.е. мы это понятие излагаем, считаем, что оно полезно и школьники должны уметь им пользоваться. В ряде случаев это намного удобнее, чем применение алгоритмов с результатами, но не более того.

Технически изложение формы записи и порядка работы ЭВМ при вызове и выполнении алгоритма-функции, как обычно, опирается на модель памяти ЭВМ. Для значения функции, которое после выполнения алгоритма-

— однопроходные — один раз что-то перебрали и получили ответ;
— двухпроходные, когда элементы таблицы (или иные объекты) для получения ответа перебираются и анализируются дважды;
— трехпроходные и т.д.

Описанный выше алгоритм нахождения числа максимумов (сначала проход по коридору и определение максимума, потом еще один проход и определение, в скольких клетках радиация совпадает с максимумом) является двухпроходным. Либо Робот у нас дважды проходит по коридору, либо элементы в таблице перебираются дважды.

Повторю, что термины однопроходный, двухпроходный, трехпроходный являются общепринятыми и не зависят ни от какого Робота. Например, компилятор называется однопроходным, если он анализирует текст программы один раз.

Почему однопроходные алгоритмы так важны.

Часто бывает, что два прохода сделать либо просто невозможно, либо — по каким-то причинам — крайне нежелательно. Например, обрабатываемые "элементы" могут поступать в компьютер через антенну со спутника. Летает спутник, собирает с датчиков какую-то информацию и, скажем, 50 раз в секунду передает ее на компьютер в центр обработки. А компьютер обрабатывает эти данные по мере поступления и в любой момент должен быть готов ответить на вопрос, каков был, например, максимальный или средний уровень излучения (если измеряется излучение) на сегодняшний день или за прошедший месяц. И все это должно работать непрерывно. Тут, во-первых, никакой памяти не хватит, чтобы запомнить всю поступающую информацию. А во-вторых, на запрос надо ответить быстро — если в этот момент начать анализировать всю информацию (а параллельно продолжать принимать новую), то все это затянется надолго. Кроме того, 50 раз в секунду — это еще не очень много, но бывают ситуации, в которых числа поступают очень быстро, и их можно обработать только раз и тут же забыть (иначе не успеешь обработать следующие).

Итак, бывает необходимо составить однопроходный алгоритм, который каждое число обрабатывает только один раз. Вернемся к задаче про число максимумов, с которой мы начали. Хотя естественное решение, которое первым приходит в голову, является двухпроходным, такая задача вполне может возникнуть в ситуации, когда два раза по коридору пройти нельзя. Например, уровень радиации таков, что Робот выдержит только один проход по коридору, а коридор такой длинный, что запомнить все числа в памяти бортовой ЭВМ Робота невозможно.

В реальном программировании, впрочем, более распространена ситуация, когда необходимость однопроходного алгоритма объясняется временными факторами. Дело в том, что получение и перебор анализируемых элементов, если этих элементов достаточно много, долгий процесс (поскольку обычно эти элементы лежат во внешней памяти ЭВМ — на диске или еще где-то и все вместе в обычную память не помещаются). В нашей учебной среде аналогичная проблема может возникнуть, если перевод Робота из одной клетки в другую требует длительного времени, является долгим. В таких ситуациях время выполнения алгоритма начинает определяться тем, сколько "проходов" мы делаем, а временем собственно обработки элементов можно пренебречь. Тогда "второй проход" вдвое увеличивает время выполнения алгоритма.

Однопроходные алгоритмы являются, как правило, самыми быстрыми. Поэтому если в рекуррентных соотношениях мы "убирали" индексы с целью экономии памяти (говоря, что ее может не хватить), то в однопроходных алгоритмах на первом месте скорость выполнения. Однопроходный алгоритм — это алгоритм, который обычно "проходит" быстрее других.

Пример составления однопроходного алгоритма.

Итак, давайте вернемся к начальной задаче — найти число клеток с максимальным уровнем радиации. И уточним — задачу надо решить за один проход, т.е. алгоритм должен быть однопроходным.

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Попробуйте, не читая учебника, решить задачу самостоятельно — и вы увидите, что это не так просто. При изучении любого из методов алгоритмизации чрезвычайно полезно — особенно для сильных учеников — задать задачу из соответствующего класса ***до изучения метода***. Как правило, это повышает мотивацию и способствует более глубокому усвоению.
>
> ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Вам, быть может, полезно будет знать следующее. Имеется строгое математическое доказательство того, что для любой задачи подобного рода
>
> а) существует однопроходный алгоритм;
> б) среди всех однопроходных алгоритмов решения есть минимальный в смысле размера памяти, требуемого для его выполнения, и
> в) этот минимальный однопроходный алгоритм — единственный (с точностью до замены обозначений, или, как говорят математики, с точностью до изоморфизма). Доказательство этого, а также более глубокое изложение метода однопроходных алгоритмов вы можете найти в [ПдМ], в разделе "Индуктивное вычисление функций на пространстве последовательностей".

Итак, однопроходный алгоритм — самый быстрый. Минимальный однопроходный алгоритм, кроме того, использует меньше всего памяти. Таким образом, минимальный однопроходный алгоритм — это наилучший алгоритм и по быстродействию, и по памяти. Теория утверждает, что для любой задачи такой алгоритм существует и в каком-то смысле он единственный. Конечно, для преподавания в школе вам все это не понадобится. Однако полезно знать, что это очень широкий класс задач и для любой из них можно составить наилучший алгоритм — однопроходный и с минимальными затратами памяти.

На школьном уровне для нас гораздо важнее разобраться с тем, как такие задачи решать. И, поскольку уровень школьный, это разбирательство базируется не на строгих теориях, а на простых аналогиях. В учебнике приведена аналогия с ловлей рыбы в простейшей задаче нахождения максимума. Проводим соревнование, кто

Что значит научиться методу рекуррентных соотношений.

Итак, давайте подведем итоги. После всех “углублений” мы теперь можем сказать, что метод рекуррентных соотношений при решении конкретной задачи состоит в следующем. Школьник должен научиться:

а) “видеть”, распознавать, что эта задача может быть решена методом рекуррентных соотношений;

б) применять метод и получать сами рекуррентные соотношения;

в) “продолжать последовательность влево” так, чтобы все ее элементы, начиная с первого, вычислялись по общей формуле;

г) записывать соответствующий алгоритм без использования таблиц и индексов.

При этом части **а** и **б** являются самыми творческими, самыми сложными в обучении. Части же **в** и **г** достаточно рутинны и просты. Если уж рекуррентное соотношение выписано, то дальше все делается известным образом и просто. Никакой смекалки, никаких “озарений” здесь уже не требуется. Явно выписанное соотношение всегда легко и быстро преобразуется в алгоритм.

Несколько слов об упражнениях (упражнения 1—12, с. 136—138 учебника).

Упражнения 1 и 11 по сути являются еще одним “углублением” метода — в этих упражнениях впервые встречаются рекуррентные соотношения с ***двумя*** последовательностями, в которых ***пара*** новых элементов последовательностей выражается через их предыдущие элементы. Сами рекуррентные соотношения приведены в указании к упр. 1 и в тексте упр. 11. Естественно, бывают и тройки, четверки и т.д. последовательностей, но эти случаи в учебнике не затронуты.

Упражнения *3а* и 4 практически повторяют задачу про сопротивление гирлянды. Они не должны вызвать никаких затруднений.

А вот упр. *3б* и *3в* достаточно сложны и интересны — в том числе и для сильных учеников (именно поэтому они помечены звездочкой “*”). Самое сложное — догадаться рассмотреть две последовательности одновременно: при решении упр. *3б* надо дополнить соответствующую последовательность аналогичной последовательностью из упр. *3в*, и наоборот. И в обоих случаях выражать ***пару*** новых элементов этих последовательностей через ***пару*** предыдущих. Именно это, хотя и в более завуалированной форме, написано в указании к упражнениям.

Повторю, что самое сложное — догадаться применить метод для решения задачи, в которой никаких рекуррентных соотношений нет. В упражнениях эта часть работы проделана авторами учебника — ученикам предлагается составить алгоритм, когда рекуррентное соотношение либо известно, либо выводится из понятных соображений. Тем не менее я хочу обратить ваше внимание, что упр. 6 и 7 можно сформулировать просто как составление алгоритма нахождения квадратного или кубического корня из заданного числа. Конечно, чтобы от этой формулировки перейти пусть даже не к рекуррентным соотношениям, а только к уравнениям, приведенным в учебнике, необходимы дополнительные математические знания. Но обратите внимание, что все же в итоге мы применяем метод рекуррентных соотношений для решения задач, внешне к рекуррентным соотношениям никакого отношения не имеющим.

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Конечно, это уже выглядит как шарлатанство. Если для гирлянды еще как-то можно было понять, почему надо применить метод рекуррентных соотношений, то в задаче “составьте алгоритм нахождения квадратного корня из заданного числа” применение метода рекуррентных соотношений кажется как минимум странным.
>
> Тут, пожалуй, мне пора признаться и открыть вам маленький секрет. Честно говоря, метод рекуррентных соотношений при желании можно применить почти к любой задаче, в которой что-то вычисляется последовательным повторением некоторых действий (в цикле). Для этого достаточно рассмотреть новые значения величин (после выполнения тела цикла) и выразить их через предыдущие значения этих величин — это и будут рекуррентные соотношения.
>
> Но содержание метода, конечно, не в этом. Если вы уже составили алгоритм, то никакие методы вам, видимо, не понадобятся. Но существует большой класс задач, в которых ***проще*** “выводить” алгоритм, применяя метод рекуррентных соотношений, чем придумывать, “угадывать” его “сам по себе” — без использования методов алгоритмизации.

16.2. Метод 2 — “однопроходные алгоритмы”

Какие алгоритмы называются однопроходными.

Сначала — объяснение слова “однопроходный”. Представьте себе, что Робот находится в горизонтальном коридоре, в клетках которого имеются какие-то уровни радиации, например, 1, 3, 7, 5, 7, 3, 1, 7. Или (если вы объясняете без Робота) задана линейная таблица из *n* элементов. И нас интересует число клеток (число элементов таблицы) с максимальной радиацией. В моем примере максимальный уровень равен 7, а число клеток с максимальным уровнем равно трем.

Итак, наша задача — подсчитать, сколько клеток имеет максимальную радиацию. Естественное решение, мгновенно приходящее в голову, выглядит так: сначала пройдем по всему коридору и определим максимум, потом пройдем обратно и подсчитаем, сколько клеток совпадает с максимумом. Для таблицы это значит, что сначала надо перебрать все ее элементы и найти максимум, а потом перебрать их еще раз и подсчитать, сколько элементов совпадает с максимальным.

Каждый перебор всех элементов таблицы (так же как каждый проход Робота по коридору) в программировании, как это ни удивительно, называется “проходом”, даже если никакого Робота и в помине нет. И алгоритмы обработки информации делятся на

функции будет передано в основной алгоритм, используется величина со специальным служебным именем **знач**. Заметьте, что **знач** — это и специальное имя, и отдельный ***вид*** величин алгоритмического языка. Внутри алгоритма-функции с величиной **знач** можно работать как с любой другой, обычной величиной. По окончании выполнения алгоритма-функции значение величины **знач** подставляется в выражение вместо вызова алгоритма-функции.

Слово **знач** является сокращением от слов “значение функции” и, как и вся остальная терминология школьного курса информатики и школьного алгоритмического языка (алгоритм, аргумент, результат, величина, значение и пр.), было введено Андреем Петровичем Ершовым путем заимствования из математики.

Итогом всего этого параграфа, итогом овладения понятиями вспомогательного алгоритма, аргументов, величин, результатов, алгоритмов-функций и т.д. является составление алгоритма построения графика произвольной функции **f**, изложенного в п. 12.11. Точнее, здесь отдельно имеется алгоритм, который рисует график произвольной функции, а отдельно алгоритм-функция (в смысле информатики), который эту функцию (в смысле математики — увы, иначе и не скажешь) задает.

На этом изложение третьего фундаментального понятия информатики — понятия вспомогательного алгоритма — заканчивается. Тем самым из четырех понятий, про которые я неустанно твержу с самого начала, два (циклы и вспомогательные алгоритмы) пройдены полностью. Величины затронуты, но пока не пройдены таблицы. А четвертое понятие — информационные модели исполнителей — даже еще и не упомянуто.

§ 13. Команды ввода/вывода информации. Цикл <u>**для**</u>

Этот параграф, как и § 7, носит характер своего рода “перерыва”: после содержательного § 12 про алгоритмы с результатами и алгоритмы-функции это очень простой параграф с очень простым материалом, который можно рассматривать как смену деятельности, как некоторый отдых. Параллельно ученики могут “подтянуть” уже пройденный материал, если кто-то что-то не успел.

В соответствии со своим названием параграф содержит две части:

1) команды ввода/вывода информации;

2) цикл **для**.

Эти две части практически друг с другом не связаны, но мы их пересекли за счет подбора задач, в которых используется и то, и другое.

Сейчас, когда курс информатики в школах преподается уже несколько лет, я думаю, нет необходимости рассказывать вам, что это за команды и как они работают.

Команды ввода/вывода.

При объяснении этих команд вы можете задействовать ранее пройденный материал. Если, например, выполняя команду **ввод n**, человек наберет на клавиатуре число **25**, то ЭВМ произведет ***в точности*** те же действия, что и при выполнении команды присваивания **n:= 25**. Таким образом, число **25** станет значением величины **n** (будет записано внутрь “прямоугольника” величины **n**).

Обратите внимание, что в отличие от Бейсика и Паскаля команда **вывод** в школьном алгоритмическом языке сама по себе не приводит к переходу на новую строку. Если мы хотим указать ЭВМ, что следующая порция информации должна выводиться с новой строки, то надо явно написать служебное слово **нс** (сокращение от “новая строка”) в команде вывода. Если же мы слово **нс** не пишем, то никакого неявного перехода на новую строку не осуществляется и разные команды вывода будут выводить информацию друг за дружкой в одну строку. Это, пожалуй, единственный нюанс школьного алгоритмического языка.

С другой стороны, ввод информации человек обычно заканчивает нажатием на клавишу [Enter] (т.е. переходом на новую строку). Поэтому при чередовании команды вывода с командами ввода, как правило, никаких **нс** писать не надо. Другими словами, в обычных простых диалогах можно обойтись и без **нс**.

В качестве примера я вам рекомендую алгоритм А57 со с. 104 учебника, вычисляющий размер вклада. Если вы этот алгоритм будете демонстрировать на компьютере, то я советую вам “поиграть” с процентами и понаблюдать, как растет вклад при разных процентных ставках.

> ПРОГРАММНОЕ И МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Мы смогли отодвинуть команды ввода/вывода так далеко от начала курса по следующим причинам.
>
> 1) Никакой фундаментальной сущности, никаких фундаментальных для развития алгоритмической культуры, алгоритмического мышления учащихся понятий за этими командами не скрывается (поэтому они и столь просты для изучения). Соответственно, в безмашинном курсе они и вовсе не нужны.
>
> 2) В случае машинного курса использование системы КуМир, в которой значения аргументов основного алгоритма ***автоматически*** запрашиваются у человека, а значения результатов автоматически выводятся на экран, вместо команд ввода/вывода можно оформлять соответствующие величины как аргументы и результаты алгоритмов (что соответствует сути дела). А до того, как введены понятия аргументов и результатов, можно просто менять числа внутри алгоритма и тут же запускать его на выполнение, наблюдая на “полях” алгоритма результаты его работы.
>
> Соответственно, если у вас курс машинный, но адекватного программного обеспечения (системы КуМир) нет, то скорее всего вам придется перенести изучение команд ввода/вывода гораздо ближе к началу курса.

Диалоговые системы и схема программного управления.

Я также хочу обратить ваше внимание на небольшое замечание про диалоговые системы в п. 13.5. Это замечание вносит существенное изменение в схему про-

граммного управления, в которой раньше у нас человек не участвовал. Точнее, я должен признаться, что в схеме программного управления слово "человек" означало ***программиста*** того, кто составляет алгоритм.

В ходе выполнения алгоритма ЭВМ может выводить какую-то информацию для "человека" и вводить какую-то информацию от него. Здесь под словом "человек" понимается ***пользователь***, т.е. тот, кто использует ЭВМ вместе с алгоритмом для решения каких-то своих задач, например, указывая ЭВМ, какую именно деталь надо изготовить (вспомните про информационную индустрию — см.: Лекция 3, № 5/99, с. 8).

Таким образом, наша схема "программного управления" существенно усложнилась. Во-первых, мы уже знаем, что исполнителей может и не быть — алгоритм бывает "чисто информационным", вычислительным. Во-вторых, как мы только что выяснили, с помощью команд ввода/вывода ЭВМ может взаимодействовать с человеком в ходе выполнения алгоритма.

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Впрочем, схема программного управления как картинка нужна на старте, чтобы было понятно, о чем идет речь. И ее лучше не усложнять, потому что если с самого начала оговаривать все возможные варианты и случаи, то вместо схемы в голове у учеников будет "каша". А теперь, когда схема уже устоялась и свою роль сыграла, мы можем посмотреть на нее и критически. Наше знание шире и глубже, чем эта схема.
>
> Я также, забегая вперед, скажу, что мы пока не изображаем на схеме таких исполнителей, как "клавиатура" и "экран" (они будут введены в § 19). В этом месте экран и клавиатура не отделяются от ЭВМ.

Цикл **для**.

Как и команда **выбор**, цикл **для** является избыточной конструкцией в том смысле, что мы абсолютно любой алгоритм можем записать и без этого цикла, заменив его циклом **пока**:

```
i:=i1
нц пока i≤i2
 | тело цикла
 |  i:=i+1
кц
```

Но это не очень удобно. Кроме того, можно забыть написать присваивание начального значения перед циклом или увеличение величины **i** внутри цикла. А при работе с табличными величинами циклы такого вида встречаются очень часто. Поэтому цикл **для** вводится здесь просто как удобная и компактная форма записи.

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Цикл **для** помещен именно в это место учебника по двум причинам:
>
> 1) он очень прост и попадает в разряд "отдыха", а § 13, повторю, имеет именно такой статус;
>
> 2) содержательно цикл **для** будет использован нами при работе с табличными величинами, а это следующий §14.
>
> И еще одно замечание. Первоначально (в издании 1988 года) цикл **для** излагался в самом начале, еще до цикла **пока**. Но, по отзывам учителей, это было неудачно, так как объяснять цикл **для** без использования понятия "величины" (для **i**) и оператора присваивания оказалось сложным. Именно поэтому цикл **для** переехал в § 13.

Упражнения после параграфа, естественно, задействуют введенные в нем команды ввода/вывода и цикл **для**, но в остальном являются обычными задачами на составление (упр. 1, 4 — 6), изменение (упр. 2) и анализ (упр. 3) алгоритмов.

§ 14. Табличные величины

Этот параграф посвящен второму фундаментальному понятию информатики — "величины, и прежде всего табличные величины". Соотношение между "табличными величинами" и просто "величинами" примерно такое же, как между "циклами" и просто "командами". Если циклы позволяют нам коротко описывать огромные последовательности действий для ЭВМ, то табличные величины дают возможность коротко описывать огромные массивы информации, которые должны быть обработаны.

Я повторю, что во всех языках программирования (честно говоря, я не знаю ни одного исключения) вместо термина таблица используется термин массив (массив данных, массив информации). Соответственно "линейные таблицы" называются "одномерными массивами", "прямоугольные таблицы" — "двумерными массивами", а также существуют не затронутые в учебнике трех-, четырех-, пяти- и пр. мерные массивы.

Линейные таблицы.

Изложение этого материала базируется на модели памяти ЭВМ. С учетом всего ранее пройденного для объяснения понятия линейной таблицы достаточно нарисовать картинку типа той, что приведена в п. 14.2 учебника, сказать, что табличная величина состоит из "элементов", ввести понятие индекса и форму записи элемента таблицы **k[i]**. Этого будет вполне достаточно для практического решения задач. Другими словами, материал здесь достаточно прост.

Но я хочу обратить ваше внимание на следующее. Табличная величина имеет одно общее для всех ее элементов имя, а элементы таблицы отдельных имен не имеют. Именно этим табличная величина отличается от просто набора из нескольких величин. За счет этого мы можем:

1) компактно описать большое количество "элементарных" величин (например, запись **цел таб k=[1 : 1000]** заставляет ЭВМ отвести память для тысячи целых чисел);

2) используя имя таблицы **k** и *вычисленное в алгоритме* значение величины **i**, получить или изменить **i**-й элемент таблицы, написав **k[i]** вместо отсутствующего имени элемента.

И здесь происходит внешне не слишком заметный, но очень важный, качественный скачок. Поскольку значение величины **i** может меняться при выполнении ал-

кие-то промежуточные величины, запомнить в них старые значения, а лишь потом присваивать новые.

Запоминание всех "старых" значений.

Впрочем, есть простой путь, который позволяет и в этом случае писать алгоритм, "не задумываясь" о порядке присваиваний и прочем. Надо ввести "второй комплект", продублировать величины. Если мы использовали a и b, то надо ввести "a старое" (as) и "b старое" (bs). После чего переход к следующему элементу последовательности записывается в виде:

```
as := a; bs := b
a := ... (формула от as и bs)...
b := ... (формула от as и bs)...
```

Для последовательности Фибоначчи можно написать:

```
as := a; bs := b
a := as + bs
b := as
```

Если бы величин было пять, то надо было бы ввести еще пять величин для "старых" значений и в формулах перехода использовать величины с этими "старыми" значениями. Такой простой подход срабатывает абсолютно всегда и позволяет не думать о порядке вычислений (но требует удвоить число величин).

Вы можете видеть, что простая и ясная связь между рекуррентными соотношениями и текстом алгоритма здесь несколько теряется. Эффективность вычислений достигается за счет частичной потери простоты и ясности. Но зато мы не храним все элементы последовательности. Это также "исчезновение индексов", но в чуть более сложной ситуации.

Как избавиться от "нерегулярностей" в начале последовательности.

Последнее "углубление" метода рекуррентных соотношений, рассматриваемое в учебнике (п. 16.6), связано с начальными "нерегулярностями". Уже для последовательности Фибоначчи значение n-го элемента надо отдельно вычислять для $n = 1$ и для $n = 2$. Общая формула применима только начиная с $n = 3$. Соответственно, в алгоритме появляются дополнительные конструкции ветвления, что, конечно, усложняет алгоритм (в алгоритме А81 это конструкция **если**, в которой основное содержание алгоритма записано после **иначе**).

Если рекуррентные соотношения окажутся еще сложнее, например очередной элемент будет выражаться через пять предыдущих, то первые пять значений надо будет задавать отдельно и нам придется написать что-то вроде:

```
выбор
 | при n=1 : знач : =...
 | при n=2 : знач : =...
 | при n=3 : знач : =...
 | при n=4 : знач : =...
 | при n=5 : знач : =...
 | иначе
 | ......................
 | ......................
все
```

От таких "начальных нерегулярностей" и, соответственно, дополнительных ветвлений в алгоритме, как правило, можно избавиться, если "продолжить последовательность влево" с сохранением рекуррентных соотношений (в математике это называется "доопределением" функции).

Это значит, что мы можем попробовать придумать, подобрать такие a_0, a_{-1} и т.д. (подобрать элементы "влево" от обычных), чтобы все обычные (не придуманные) элементы последовательности, начиная с a_1, вычислялись через общее рекуррентное соотношение для данной последовательности.

Как это сделать? Рассмотрим пример из учебника — последовательность Фибоначчи — еще раз. Основное рекуррентное соотношение последовательности Фибоначчи — это

$$a_n = a_{n-1} + a_{n-2}$$

Следовательно,

$$a_{n-2} = a_n - a_{n-1}$$

Используя эту формулу, легко начать вычислять элементы "влево":

при $n = 2 : a_0 = a_2 - a_1 = 1 - 1 = 0$
при $n = 1 : a_{-1} = a_1 - a_0 = 1 - 0 = 1$
при $n = 0 : a_{-2} = a_0 - a_{-1} = 0 - 1 = -1$
и т.д.

Прием называется "продолжение последовательности влево", потому что мы вычисляем недостающие слева элементы a_0, a_{-1} и пр. через рекуррентное соотношение и известные нам элементы последовательности (расположенные "справа" от вычисляемых).

Реально надо довычислить столько "левых" элементов, через сколько предыдущих элементов выражается очередной элемент последовательности, с тем, чтобы общее рекуррентное соотношение оказалось применимым для вычисления a_1. Например, для последовательности Фибоначчи достаточно продолжить последовательность влево на два элемента: a_0 и a_{-1}. После этого мы можем переписать рекуррентное соотношение в виде:

$$a_n = a_{n-1} + a_{n-2}; \quad a_{-1} = 1; \quad a_0 = 0$$

Чрезвычайно важно, что основное рекуррентное соотношение не изменилось. Изменились только начальные условия. Зато теперь ***любые*** "обычные" элементы последовательности, начиная с a_1, можно вычислять по общей формуле. Соответственно, нам более не нужны развилки — и алгоритм А81 превращается в алгоритм А82.

И смотрите — опять тот же эффект от применения метода: наши рассуждения несколько усложнились и удлинились, зато алгоритм стал еще проще! Дальше надо порешать задачи, "понабивать руку", довести эти навыки до некоторого автоматизма — и вы с удивлением увидите, как целый класс задач станет для вас простым и рутинным, таким же, как нахождение корней квадратного уравнения.

Конечно, иногда бывают случаи, когда такое "продолжение влево" оказывается невозможным — в формуле возникает какое-нибудь деление на 0 или еще что-то. Но если последовательность удается продолжить влево, то это позволяет обойтись без **если**, без отдельных частных случаев и т.п., дает возможность намного упростить алгоритм, оставить только вычисления по общей формуле.

После того как рекуррентное соотношение составлено (т.е. математические формулы получены), мы можем — уже не задумываясь — записать алгоритм (см. алгоритм А80 учебника).

Посмотрите на этот алгоритм — он чрезвычайно прост! Один цикл, внутри которого всего один оператор присваивания. И это для подсчета сопротивления такой схемы! А ведь на старте задача казалась весьма сложной.

Дело в том, что все сложности этой задачи в результате оказались "скрытыми" в методе ее решения. Это общее свойство: применяя какой-то метод, мы всегда используем ранее накопленное знание. В этот момент мы не разбираемся в деталях метода, не анализируем, "почему это так", — в освоенном методе мы "знаем, что это так", и используем все вместе как единый и простой элемент знания. Вывод формулы для корней квадратного уравнения — весьма содержательная задача. Но если мы формулу уже вывели и запомнили, то при решении конкретных задач мы просто ее используем, не задумываясь, откуда она взялась и почему верна. И решение конкретных квадратных уравнений становится делом достаточно простым.

Так же и у нас: получив рекуррентные соотношения для гирлянды из лампочек, мы, используя старое знание (например, про "исчезновение индексов"), можем, "не задумываясь", просто записать алгоритм. И алгоритмы, как правило, получаются очень простые, как А80. Они намного проще, чем те, с которыми мы уже возились (цикл, внутри него еще цикл, внутри **если** или еще что-нибудь). Но, поскольку эти алгоритмы сразу "не видны", не являются очевидными и сами по себе в голову не приходят, их надо выводить, используя какие-то методы алгоритмизации. Поэтому научиться составлять такие алгоритмы сложнее, хотя, повторю, сами алгоритмы могут оказаться много проще уже нами изученных. Ведь нам теперь надо овладеть не просто алгоритмическими обозначениями, но и ***методами алгоритмизации***, а это, конечно, сложнее.

Таким образом, метод рекуррентных соотношений состоит в том, что в задаче надо (1) увидеть какую-то последовательность величин (в задаче выше — увидеть последовательность гирлянд для растущего n), (2) выразить искомые величины рекуррентными соотношениями и (3) записать соответствующий алгоритм.

Рекуррентные соотношения с несколькими "предыдущими" элементами.

Заканчивается раздел про метод рекуррентных соотношений еще одним "углублением". Дело в том, что до сих пор рассматривались только соотношения, в которых следующий элемент выражался через ***один*** предыдущий элемент. Существуют ситуации, когда очередной элемент выражается не через один предыдущий, а через два, три или более. Например, в последовательности Фибоначчи очередной элемент выражается через два предыдущих. Тогда описанное выше простое "выбрасывание индексов" не срабатывает — ведь нам надо хранить не одно, а несколько предыдущих значений (предыдущих элементов последовательности), поэтому и величин в алгоритме понадобится несколько. Если в рекуррентной формуле справа фигурируют разные a_{i-1}, a_{i-2}, а мы просто выкинем индексы, то получится ерунда — одна буква будет ***в одно и то же время*** обозначать разные элементы последовательности. Поэтому следующая часть параграфа посвящена вопросу "что делать?", если очередной элемент последовательности выражается через ***несколько*** предыдущих.

Рассмотрим это на примере последовательности Фибоначчи, когда для вычисления очередного элемента последовательности нужно знать два предыдущих. Естественно, мы легко можем записать алгоритм с использованием таблицы для запоминания элементов последовательности (как мы это делали в алгоритме А78). Но для записи алгоритма без индексов нам понадобятся уже две величины — для хранения двух предыдущих значений. Назовем их, например, `a` и `b`. Пусть в какой-то момент времени значением величины `a` является a_{i-1}, а значением `b` — a_{i-2}. Тогда, если я от этой пары хочу перейти к паре a_i, a_{i-1}, т.е. в математической схеме увеличить i на единицу, то мне нужно сделать следующее. Поскольку величина `b` должна стать равной a_{i-1} (это старое значение `a`), следует выполнить команду присваивания `b := a`. Новое значение `a` должно стать равным $a_i = a_{i-1} + a_{i-2}$, т.е. новое значение `a` можно получить как сумму старых значений `a` и `b`, т.е. командой присваивания `a := a + b`.

Необходимость запоминания некоторых "старых" значений.

Беда в том, что эти два действия в моих рассуждениях — `b := a` и `a := a + b` — должны выполниться как бы параллельно. Если мы сначала выполним первое действие, то "потеряем" старое значение `b` и не сможем его использовать во второй формуле. Поэтому нам придется совершить некоторые дополнительные действия, например, запомнить старое значение `b` в какой-нибудь промежуточной величине `c` и после изменения `b` в формуле для нового значения `a` использовать это запомненное значение `c`:

```
c := b;    b := a;  a := a + c.
```

Это один из возможных вариантов, при котором значением `c` является a_{i-2}, а сама величина `c` нужна только на время перехода от $a = a_{i-1}$, $b = a_{i-2}$ к $a = a_i$, $b = a_{i-1}$.

Вы видите, что если очередной элемент последовательности выражается через несколько предыдущих, алгоритм становится сложнее: надо запоминать эти предыдущие значения и следить за порядком действий, чтобы эти "предыдущие значения" случайно не испортить. Сложность составления алгоритма довольно резко вырастает, хотя содержание нашего "исчезновения индексов" почти не меняется. Содержание остается тем же — увидев, что очередной элемент выражается через предыдущие, мы должны понять, сколько предыдущих элементов нам необходимо, и ввести соответствующее количество величин для хранения их значений. Этих величин будет ровно столько, сколько чисел нам надо было бы помнить, если бы мы считали вручную.

Дополнительная сложность возникает позже, при записи в величинах формул перехода к следующему элементу последовательности, поскольку в этот момент следует позаботиться, чтобы последовательность присваиваний (которые будут выполняться одно за другим) не испортила нужных нам значений. Возможно, при этом придется ввести алгоритма, мы получаем возможность компактно записать обработку большого количества информации. Например, написав

```
нц для i от 1 до 1000
| k[i]:=0
кц
```

мы можем заставить ЭВМ всем этим 1000 элементам присвоить значение `0`. Такой цикл заставляет ЭВМ не только выполнить массу действий (это мы уже проходили), но и изменить при этом массу информации — 1000 чисел. А ведь в приведенном фрагменте всего три строчки!

Таким образом, использование табличных величин позволяет составлять компактные алгоритмы, обрабатывающие огромное количество информации, задействовать не только быстродействие, но и объем памяти ЭВМ.

Использование Робота при изложении таблиц.

Для лучшего усвоения понятия таблицы можно использовать аналогии с соответствующими структурами на поле Робота. Скажем, горизонтальный коридор на поле Робота с заданной в каждой клетке коридора радиацией — полный аналог линейной таблицы с вещественными элементами. Чтобы это подчеркнуть, а также обеспечить плавный переход от Робота к таблицам и слегка окрасить скучные математические формулировки типа "подсчитать сумму элементов таблицы", в учебнике приведен алгоритм А62, копирующий информацию о радиации в коридоре (**n** вещественных чисел) в таблицу **вещ таб a[1 : n]**. Хотя в этот момент, как правило, никакой "мотивации" уже не требуется, продолжая "игры" с Роботом, можно объяснить постановку задачи: у нас "одноразовый" Робот — уровень радиации в коридоре таков, что Робот может пройти коридор только один раз. Второго раза даже его железный организм не выдержит.

После алгоритма А62 все наши стандартные задачи для Робота в коридоре (найти максимальный уровень радиации, подсчитать число клеток с максимальным уровнем и пр.) переформулируются в задачи обработки линейных таблиц.

За счет указанной постановки получается такой плавный переход от Робота и алгоритмов его управления к таблицам. Поскольку после запоминания информации об уровнях радиации в коридоре мы начинаем работать с таблицами, появляется некоторое обоснование, осмысление, какая информация и откуда может в этих таблицах появиться. Соответственно, последующие задачи про таблицы приобретают какую-то "жизненную" окраску.

Кроме линейных таблиц, в конце параграфа вводятся прямоугольные таблицы, т.е. массивы с двумя индексами, которые изображаются на "доске" (в модели памяти ЭВМ) в виде прямоугольных таблиц (откуда и название). Насколько мне известно, этот материал никогда ни у кого никаких сложностей не вызывал.

Упражнения

Конечно, ключевой материал этого параграфа — введение самих таблиц и работа с ними. Как всегда в нашем курсе ("черепаха" курса — "понимание через делание"), имеется достаточное количество задач, которые должны быть разобраны в классе, и почти бесконечное количество упражнений, которые можно задать для самостоятельного выполнения.

Именно в упражнения после этого параграфа попадает, я бы сказал, "джентльменский набор" простейших задач и упражнений по программированию: индекс максимального или минимального элемента таблицы, среднее арифметическое элементов таблицы, упорядочивание элементов таблицы по возрастанию и пр.

§ 15. Логические, символьные и литерные величины

Этот параграф завершает изложение второго фундаментального понятия информатики — понятия величины. Здесь суммируется все, что было пройдено про величины ранее, а также вводятся величины трех новых ***типов:*** логические, символьные и литерные. Таким образом, понятию величины всего в нашем курсе посвящены три параграфа: 11 (введение понятия величины), 14 (табличные величины) и 15 (тип величины, логические и прочие величины).

Тип величины.

Всякая величина — это четверка (***имя, значение, тип, вид***). С понятиями имени и значения, как правило, никаких проблем у школьников не возникает. (Для тех, кто знаком с другими языками программирования, отмечу только, что имя в алгоритмическом языке может состоять из нескольких слов, например, "число горячих клеток", и при этом их можно обычным образом писать через пробел, не используя никаких специальных символов типа подчеркивания **_**).

Говоря, что с понятием значение проблем не возникает, я имею в виду само это понятие и обычные числовые (целые и вещественные) значения. В этом параграфе будут введены значения новых типов — логические (где значением является **да** или **нет**), а также символьные (где значением является символ, например, **a**, **+**, **;**, **1**, **!** и пр.). Но связанные с этим вопросы будут отнесены к понятию типа величины.

Понятие вида величины содержательно понадобится нам только при изложении общих величин исполнителей, а до той поры может никак специальным образом не вводиться вообще (что и сделано в учебнике).

А вот тип величины — понятие и достаточно сложное, и очень важное. Поэтому я на нем остановлюсь гораздо более подробно, чем в учебнике.

Прежде всего формальное определение. ***Тип*** — это характеристика величины, задающая

а) множество значений, которые может принимать величина, и

б) множество действий, операций, которые можно совершить с величиной (т.е. со значением) данного типа.

Например, величина целочисленного типа может принимать только целые значения. Другими словами, значением такой величины может быть только целое число. Величины такого типа (а точнее, значения такого типа, т.е. целые числа) можно складывать, вычитать, умножать

и делить — при этом будут получаться целые числа (при делении получится частное, а остаток (дробная часть) будет потерян). Целые числа также можно сравнивать: **<, ≤, =, ≠, >** и пр.

Впрочем, в алгоритмическом языке для величин и значений ***любого*** простого типа допустимы операции сравнения **=** и **≠**, а также команда присваивания **`величина := значение`**.

Поэтому при описании нового типа величин необходимо указать:

а) какие значения может принимать величина этого типа;

б) какие действия и операции (кроме **=, ≠, :=**) над ней допустимы.

Например, когда мы вводим понятие величины логического типа, мы должны сказать:

а) что величина этого типа может принимать только одно из двух значений: либо **да**, либо **нет** (полезно напомнить ученикам о битах и отметить, что для хранения значения логической величины достаточно одного бита);

б) что над величинами и значениями данного типа допустимы все логические (откуда и название типа) операции, как-то: **и**, **или**, **не**.

Кстати, для таблицы значением является набор чисел, а количество элементов таблицы и их индексы ***входят*** в ***тип*** таблицы. Другими словами, **`цел таб a[1 : 10]`** и **`цел таб b[0 : 9]`** — это величины двух ***разных***, хотя и "родственных" типов. Для первой величины допустима операция получения элемента **`a[10]`**, которая недопустима для второй величины — для **`b`**.

Итак, тип указывает, какие значения может принимать величина и что этими значениями можно делать. Те типы, которые мы уже изучали (**цел**, **вещ**, **цел таб** ... и пр.), как и те, что изложены в этом параграфе (**лог**, **сим**, **лит**), являются так называемыми встроенными, предопределенными типами алгоритмического языка. Других (не предопределенных) типов в алгоритмическом языке нет. Доопределить, построить "свой" тип нельзя.

Я уже говорил и еще скажу в заключение, что в этом месте алгоритмический язык отличается от всех современных языков программирования и от языка Паскаль, в которых есть средства конструирования новых типов данных. Здесь я лишь хочу бегло обратить на этот факт ваше внимание, а подробно мы остановимся на нем в заключение, когда будем обсуждать место курса в "большой" информатике.

Логические величины.

Для объяснения, что такое величина логического типа, необходимо указать, какие значения может принимать такая величина, а также где и как ее можно использовать. Я это практически уже проделал.

Добавлю только для полноты картины, что значением логической величины может быть либо **да**, либо **нет**, либо ***значение величины может быть неопределено.*** Последнее, впрочем, относится к величинам любых типов.

Кроме того, величины логического типа (логические значения) можно использовать в условиях в алгоритмическом языке. Простые примеры имеются в учебнике.

Я же хочу обратить ваше внимание на задачу и алгоритм в п. 15.5. Задача такая. От горизонтального коридора на поле Робота кое-где вверх отходят тупики разной высоты. Надо закрасить клетки коридора напротив тех тупиков, высота которых больше трех клеток (мы их назвали "длинными тупиками").

Если мы начнем решать эту задачу, т.е. составлять алгоритм, у нас, конечно, будет цикл, потому что неизвестно, сколько клеток в коридоре. Условие окончания цикла (вспомните п. 9.11) — условие выхода из коридора, т.е. **`снизу свободно`**. Внутри цикла надо идти по коридору вправо, и если сверху обнаружится "длинный" тупик, то соответствующую клетку коридора закрасить.

И вот тут очень важное место. Я бы сказал — интерференция, взаимное усиление всех пройденных нами понятий. Теперь, после введения величин логического типа, мы в соответствии с методом последовательного уточнения эту изложенную выше *идею* решения можем записать прямо на алгоритмическом языке:

```
если сверху длинный тупик
| то закрасить
все
```

Здесь условие "сверху длинный тупик" — вызов вспомогательного алгоритма-функции, значением которого является ***логическая*** величина.

> ЛИРИЧЕСКОЕ ОТСТУПЛЕНИЕ. Собственно говоря, многие конструкции в алгоритмическом языке вводятся именно для того, чтобы, произнеся нормальную человеческую фразу типа "если длинный тупик, то закрасить клетку коридора", мы могли бы ее примерно в таком же нормальном виде записать в алгоритм. Но уже строго и недвусмысленно — в виде фрагмента алгоритма на алгоритмическом языке.

Конечно, после того как мы написали этот фрагмент, мы в соответствии с методом последовательного уточнения должны еще написать вспомогательный алгоритм-функцию, который будет анализировать, есть ли сверху тупик и является ли он "длинным". Это алгоритм А73 на с. 119 учебника.

Теперь я готов сказать, что уровень "письма и счета" достигнут. Если школьники в состоянии так записать алгоритм, понимая, что это — не неформальная запись, а абсолютно строгая, представляя себе, как ЭВМ будет выполнять такой алгоритм, что будет происходить в памяти ЭВМ и т.д., то это и значит, что базовые понятия и навыки алгоритмизации, а с ними и алгоритмический язык как инструмент записи алгоритмов освоены. С этого момента можно переходить к изучению применений ЭВМ, а также к использованию полученных знаний для понимания устройства окружающего нас мира.

Имеется гирлянда из проводов с лампочками (*рис.* 65 учебника) из достаточно большого числа (n) параллельно включенных лампочек, сопротивлением $R2$ каждая, и при этом сопротивлением соединительных проводов ($R1$) нельзя пренебречь. Требуется посчитать общее сопротивление гирлянды между клеммами A и B. Такая вот задача.

> МЕТОДИЧЕСКОЕ ОТСТУПЛЕНИЕ. Я уже говорил, что мы старались все задачи в учебнике подобрать таким образом, чтобы продемонстрировать полезность информатики, показать, что можно решать задачи, которые в рамках обычной математики или физики не решаются.
>
> На наш взгляд, попытка, скажем, представить алгоритм вычисления корней квадратного уравнения как нечто содержательное с точки зрения информатики — это просто дискредитация информатики. Трудозатраты на написание, ввод и выполнение такого алгоритма гораздо выше, чем на непосредственное нахождение корней по формуле. Смысл этой деятельности остается непонятным школьнику независимо от того, какие аргументы приводит учитель. Конечно, переписывание формулы в виде алгоритма требует отчетливого понимания того, что корней может и не быть, а также знания алгоритмических обозначений (соответствующего языка программирования), но практически не дает никакого вклада в развитие алгоритмического мышления (а для нас, повторю, развитие алгоритмической составляющей мышления — главная цель курса). Поэтому, на наш взгляд, алгоритмы типа "нахождения корней квадратного уравнения" могут быть составлены только "мимоходом", обсуждаться, как фрагменты решения какой-то более интересной и понятной задачи.
>
> **Базовые задачи информатики должны быть задачами с явно выраженной алгоритмической составляющей, задачами, которые школьники без компьютера, без информатики решить не могут.**
>
> Задача с гирляндой лампочек — одна из них (во всяком случае, *формула* сопротивления в такой цепи школьникам неизвестна и, думаю, они не в состоянии ее вывести). Написать для этой задачи ответ так, как это принято в физике или в математике (в виде какой-то формулы), школьники не могут, т.е. задача не решается обычными физическими и математическими методами. И именно этим она хороша для нас, поскольку демонстрирует собственную область применения информатики, ее несводимость и невторичность по отношению к математике и физике.
>
> И ЕЩЕ ОДНО ОТСТУПЛЕНИЕ. Единственная разумная формула, которую могут придумать школьники, — это формула для сопротивления ***бесконечной*** гирлянды. Сопротивление x такой ***бесконечной*** гирлянды может быть найдено, как корень квадратного уравнения, которое получится, если в рекуррентном соотношении вместо a_i и a_{i-1} подставить x.

Итак, наша задача — найти сопротивление между клеммами A и B. Помните, я говорил, что первое, что необходимо уметь, — это видеть. В данном случае увидеть, что это — задача на рекуррентные соотношения, хотя в ее постановке нет ни слова ни о каких рекуррентных соотношениях. Такое "видение" дает только практика, только опыт. Но обычно, если в постановке задачи есть многоточие, n каких-то элементов и т.п., то это вполне может оказаться задача на рекуррентные соотношения.

Что значит "задача на рекуррентные соотношения"? Это значит, что мы можем выразить сопротивление гирлянды из n лампочек через сопротивление такой же гирлянды из $n - 1$ лампочки. Т.е., получив задачу, школьник должен это просто "увидеть" — увидеть, что можно рассмотреть последовательность таких гирлянд, начиная с $n = 1$ и постепенно увеличивая n. И что сопротивление для следующей гирлянды можно выразить через сопротивление для предыдущей. Если он это заметил, то цель достигнута — он распознал эту задачу как задачу на рекуррентные соотношения, дальше можно начать писать формулы.

Действительно, для $n = 1$ получится гирлянда, изображенная на *рис. 66а* учебника. Сопротивление для такой простой гирлянды (электрической цепи) школьники легко могут найти, используя формулы из курса физики. При $n = 1$ общее сопротивление гирлянды будет равно $a_1 = 2 \cdot R1 + R2$.

Теперь давайте попробуем выразить сопротивление для гирлянды из n лампочек (обозначим его a_n) через сопротивление a_{n-1} гирлянды из $n - 1$ лампочки. Для этого гирлянду из n лампочек надо нарисовать как гирлянду из $n - 1$ лампочки, к которой добавлена еще одна лампочка (*рис. 66б* учебника). Получается опять довольно простая цепь, сопротивление которой может быть выражено по формулам физики:

$$a_n = 2 \cdot R1 + \cfrac{1}{\cfrac{1}{R2} + \cfrac{1}{a_{n-1}}} = 2 \cdot R1 + \frac{a_{n-1} \cdot R2}{a_{n-1} + R2}.$$

Таким образом, мы получили формулу, выражающую a_n через a_{n-1}. Вместе с формулой $a_1 = 2 \cdot R1 + R2$ это уже полное рекуррентное соотношение, задающее последовательность.

Я еще раз обращаю ваше внимание, что в постановке задачи никакого рекуррентного соотношения не было. Надо было просто вычислить сопротивление некоторой цепи. Когда я говорю, что овладение методом возможно только через работу ("понимание через делание"), то это означает, что в процессе решения такого рода задач школьники должны научиться распознавать рекуррентные соотношения там, где их в явном виде "нет", и научиться переформулировать задачу в терминах нахождения конкретного элемента какой-то последовательности, видеть эту последовательность и уметь выразить следующий его элемент через предыдущий или предыдущие. Все точно так же, как с квадратным уравнением: надо его "видеть", даже если оно внешне выглядит по-другому, например, $c \cdot (x - d)^6 + a \cdot (x - d)^3 + b = 0$.

Самая большая сложность — это научить школьников в подобного рода задачах, где никаких рекуррентных соотношений нет, эти соотношения "видеть", распознавать. Т.е. использовать метод рекуррентных соотношений как средство решения задач, когда в постановке задачи не сказано, что это задача именно на этот метод.

вычисления следующего элемента (2 * 2 – 1 = 3), и величине а будет присвоено новое значение 3. С течением времени значение величины а меняется.

Итак, в алгоритме А79 используется ***одна*** величина а, меняется только ее ***значение*** по мере выполнения алгоритма. Математические обозначения a_i и a_{i-1} относятся при этом к разным ***значениям*** одной и той же "алгоритмической" величины а в разные моменты времени, т.е. отражают, как ***меняется*** величина а.

Сравнение двух подходов к вычислению рекуррентной последовательности.

Это чрезвычайно важное место, можно сказать, сердце метода рекуррентных соотношений (хотя до самого метода мы еще не дошли). Давайте еще раз сравним алгоритмы А78 и А79. В алгоритме А78 запоминаются все элементы последовательности — каждый в отдельном элементе таблицы. Это просто, алгоритм записывается очевидным образом, но требует много памяти, расточительно и не всегда возможно. В алгоритме А79 элементы последовательности a_i и a_{i-1} обозначают разные значения величины а. Величина а меняется в ходе вычислений — это и есть переход от одного i к другому. Я называю этот переход от математических индексов в описании последовательности к обычным величинам в алгоритмах и их значениям в разные моменты времени "***исчезновением индексов***".

Самым важным является тот факт, что, когда мы вычисляем по рекуррентным соотношениям, таблицы и индексы, вообще говоря, не нужны. Для больших n это может оказаться очень существенным. Чтобы вычислить $a_{1000000}$ (миллионный элемент последовательности), в алгоритме А78 придется завести таблицу для хранения миллиона чисел. В алгоритме же А79 будет храниться только одно число, а вычислять можно и миллионный, и миллиардный, и любой другой элемент последовательности. Таким образом, с чисто прагматической точки зрения это еще и экономия памяти ЭВМ.

ПРОГРАММНОЕ ОТСТУПЛЕНИЕ. В большинстве языков программирования, будь то Паскаль, Бейсик или школьный алгоритмический язык, величина целого типа может принимать не слишком большие значения, не больше нескольких десятков двоичных цифр. Для вычисления по алгоритму А78 или А79 миллионного члена последовательности необходимо, чтобы целое число могло содержать до миллиона двоичных цифр. Если на запоминание каждой двоичной цифры тратить один бит, то для выполнения алгоритма А79 хватит памяти размером около 1 Мбит. А вот для выполнения алгоритма А78 понадобится (по минимуму, если усложнить алгоритм и на каждый элемент последовательности отводить ровно столько бит, сколько надо):

1 бит на первый элемент последовательности;
2 бита на второй элемент;
....
999 999 бит на предпоследний элемент и
1 000 000 бит на последний элемент.

Итого 1 + 2 + 3 + ... + 999 999 + 1 000 000 бит, или около 500 Гбит, т.е. в 500 000 раз больше памяти.

С содержательной же точки зрения этот эффект "исчезновения индексов" означает, что мы начинаем лучше учитывать особенности алгоритмической (динамической) составляющей. Говоря очень грубо, разница между А78 и А79 — это разница между математическим и алгоритмическим стилем мышления. Последовательность ***значений*** величины а в алгоритме А79 и есть последовательность a_i в математическом описании последовательности. Вот самое первое и самое важное, в чем вы должны разобраться.

Больше, собственно, мне на эту тему сказать нечего. Вы должны просто потренироваться, посопоставлять такие алгоритмы, "набить руку" и привыкнуть. Дело в том, что наша математическая культура и предыдущий школьный опыт "сбивают" человека — как правило, если специально не поработать, человек интуитивно пишет элемент последовательности как $a[i]$, как элемент таблицы — и получается алгоритм А78. Нужно некоторое время посоставлять алгоритмы "без индексов" (как А79), чтобы от статическо-математического подхода сместиться в сторону алгоритмического, чтобы привыкнуть к тому, что математические a_i "скрываются" здесь за одной и той же величиной а — это ее начальное значение, следующее, следующее и т.д.

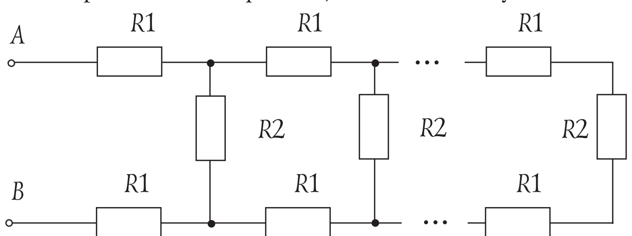
Алгоритм А78 можно понимать статически — это как бы то же самое математическое задание последовательности, только в алгоритмических обозначениях. Для понимания алгоритма А79 уже нужно знать *и* ***представлять*** себе, как он будет выполняться и что при этом будет происходить. Алгоритмическая составляющая мышления в алгоритме А79 задействована существенно глубже. Именно в этом и состоит некоторая сложность в составлении и понимании таких алгоритмов. С другой стороны, именно поэтому такие алгоритмы обычно намного эффективнее. Овладение методом рекуррентных соотношений позволяет преодолеть эти сложности и легко писать алгоритмы без "лишних" индексов.

Итак, первый важный вывод состоит в том, что если надо вычислить значение элемента последовательности, заданной рекуррентными соотношениями, то соответствующий алгоритм можно записать без использования таблиц, "без индексов".

Как применять метод на практике.

Ну а теперь, после этого продолжительного вступления, мы перейдем уже к самому методу рекуррентных соотношений.

Метод рекуррентных соотношений начинается отнюдь не с рекуррентных соотношений. Вообще в жизни очень мало задач, которые прямо так и формулируются: "вычислить n-й элемент последовательности, заданной таким-то рекуррентным соотношением". Подобные формулировки характерны для тренировочных задач в школе. Жизненные задачи формулируются совсем по-иному. В качестве такой "жизненной" задачи в учебнике рассмотрена задача по физике (требующая, соответственно, элементарного знания физики) — см. с. 127 учебника.



Символьные и литерные величины.

Раздел про символьные величины необычайно прост. Достаточно сказать, что значением символьной величины может быть любой символ, а в качестве операций можно использовать операции сравнения.

А вот литерная величина — это совершенно новое понятие. Литерную величину удобно представлять себе в виде линейной таблицы, элементами которой являются символьные величины. Но в отличие от числовых линейных таблиц ***длина*** литерной строки (т.е. количество "элементов таблицы") может меняться в ходе выполнения алгоритма. Кроме того, для литерных величин определены новые операции: "вырезка" куска строки, добавление одной строки в конец другой и пр.

Этот материал уж очень "технический" или "программистский" (в обычном, приниженном смысле этого слова). Вы можете весь материал про литерные строки опустить без особого ущерба для алгоритмической культуры учащихся (с соответствующими изъятиями из дальнейшего материала учебника). Если же вы его проходите, то я рекомендую поработать в гипертексте "литерные величины" в системе КуМир, ответить на контрольные вопросы и т.д.

Мы поместили этот материал по двум причинам.

Во-первых, его можно рассматривать как новый материал, на котором еще раз проверяются, отрабатываются и углубляются все ранее полученные школьниками знания и навыки. К тому же опыт показывает, что формулировки задач со строками и составление соответствующих алгоритмов вызывают интерес. Я чуть ниже на этом остановлюсь.

Во-вторых, мы немного задействуем этот материал в дальнейшем, при изложении понятий компиляции и интерпретации, построении информационных моделей алгоритмов и при задании названий городов в информационной модели транспортной сети.

Если величины литерного типа изучаются, то здесь можно сформулировать достаточно много разных задач, в которых аргументами и/или результатами являются строки. Эта область восходит к детским играм с шифровкой и дешифровкой, она не имеет отношения ни к Роботу, ни к Чертежнику, ни даже — в каком-то смысле — к вычислениям. Это совсем другая область, и здесь вполне могут проявиться те ученики, кому работа со словами нравится больше, чем, скажем, геометрия.

Хочу напомнить вам следующий методический прием: здесь очень хорошо использовать для усиления мотивации и лучшего понимания задач игру в "черные ящики" (см.: Лекция 3, № 5/99, с. 4). Этот методический прием состоит в том, что, прежде чем сформулировать ученикам задачу на составление алгоритма, учитель "задумывает" соответствующий алгоритм как алгоритм работы "черного ящика" и начинает играть с учениками в этот "черный ящик", выписывая на доске аргументы и результаты работы "ящика". После того как школьники отгадают правило работы "ящика" (напомню, что такое отгадывание весьма увлекательно), учитель говорит: "Молодцы, алгоритм работы "черного ящика" вы угадали. А теперь запишите этот алгоритм на алгоритмическом языке".

Еще одно методическое замечание, которое я хочу тут сделать, состоит в том, что если вы будете преподавать наш курс в более младших классах или если у вас просто будет достаточно времени, то еще на ранних стадиях курса можно устроить "пропедевтику" работы с символьными и литерными величинами с помощью соответствующих исполнителей. В учебнике таких исполнителей нет, но можно либо использовать таких исполнителей, как "Редактор слова" из нашего вузовского учебника [ПдМ], "Муравей" Г.А. Звенигородского [Звенигородский], либо просто расширенного Робота, который дополнительно умеет читать и записывать символы (буквы, цифры и пр.) в клетках поля. Тогда значительную часть задач и упражнений этого параграфа можно будет переформулировать в терминах соответствующего исполнителя (подобно тому, как задачи обработки табличной информации мы формулировали в терминах обработки радиации в коридоре на поле Робота). Опыт показывает, что задачи такого словесного характера пользуются большой популярностью. Бывают просто жемчужины, например, когда в ходе решения задачи "получить путем перестановки букв из данного слова максимальное количество других осмысленных слов" школьники придумывают, как из слова "вертикаль" сделать "кильватер".

Резюме главы 1 "Алгоритмический язык".

С содержательной, логической точки зрения мы на этом главу 1 "Алгоритмический язык" закончили. Конечно, остался еще большой и самый сложный в курсе § 16, но он в каком-то смысле лежит за пределами первой главы. В предварительных вариантах учебника это была отдельная глава, которая называлась "Методы алгоритмизации". Затем мы эту главу упростили, сжали до одного параграфа и в итоге поместили в конец главы 1. Но § 16 — методам алгоритмизации — я посвящу две следующих лекции.

А тему "Алгоритмический язык" мы почти прошли. Повторю, что цель этой главы — научить школьников владеть алгоритмическим языком не хуже, чем они владеют письмом и счетом. Владение алгоритмическим языком должно стать базовым навыком, инструментом для решения задач. У нас еще будет некоторое дополнение к языку, когда мы будем проходить четвертое фундаментальное понятие информатики в § 21—22 — понятие информационной модели исполнителя. Но в целом алгоритмический язык на этом можно считать изученным.

Лекция 7—8

§ 16. Методы алгоритмизации

Этот параграф учебника является, по-видимому, самым сложным и, соответственно, одним из самых интересных. Если по учебнику преподавать в 7—8-х классах, то его, на мой взгляд, следует пропустить. Также его можно пропустить, если у вас слабый класс.

Параграф этот — повышенной сложности, а кроме того, в нем существенно задействован математический (логический) стиль мышления, т.е. от учащихся требуется умение рассуждать и мыслить логически, необходимо наличие у них минимальной математической культуры.

Но в то же время всякая наука становится наукой, когда в ней появляются ее собственные методы решения задач. Если опять использовать аналогию с математикой, то существует огромная разница между

— умением угадывать корни квадратных уравнений или находить их подбором и

— знанием общей формулы для нахождения корней квадратного уравнения и умением ее применять.

И хотя угадывание или подбор корней квадратных уравнений — несомненно, деятельность, как-то связанная с математикой, назвать такое "угадывание" наукой (математикой), конечно, язык не повернется.

Материал § 16 в нашем учебнике как раз и демонстрирует, что информатика достигла определенной зрелости. Обращаю ваше внимание, что до сих пор мы именно "угадывали" алгоритмы. Конечно, мы изучили много новых понятий — алгоритмы, управляющие конструкции, величины и пр. (подобно тому, как для угадывания корней квадратного уравнения надо предварительно изучить, что такое "квадратное уравнение" и что такое "корень"). Но алгоритмы мы придумывали по наитию. Мы их "подбирали" или "угадывали". Конечно, мы при этом о чем-то думали, как-то рассуждали и хотя бы интуитивно использовали какие-то общие соображения типа "без цикла тут не обойтись". Но ***мы не использовали явно никаких методов создания алгоритмов*** — мы их "придумывали", а не "выводили" по общим правилам и формулам.

Лишь мельком я говорил про какие-то методы последовательного уточнения (см.: Лекция 4, № 6/99, с. 5). Но в целом все задачи решались, образно говоря, "озарением" — в каждой задаче решение просто придумывалось, а объяснение решения часто сводилось к слову "Смотри!" (легенда гласит, что именно так древние греки объясняли решения геометрических задач).

Конечно, и задачи у нас были не самые сложные, поэтому, зная формулировку задачи и конструкции языка (циклы, **если** и пр.), мы эти простые задачи как-то решали. Решение часто было "видно", и, честно говоря, мы не столько учились решать задачи, сколько записывать решения, которые "видно" и так. (В науке предпочитают говорить "решение очевидно".) Повторю, что при этом никакого аналога "формулы для корней квадратного уравнения" у нас не было. Мы придумывали алгоритмы, а не "выводили" их по каким-то формулам.

В § 16 впервые излагаются простейшие методы решения алгоритмических задач.

Освоение любого метода состоит в овладении двумя разными "навыками и умениями". Ученик должен

1) узнавать задачи из соответствующего класса задач, т.е., увидев задачу, он должен понять, что эта задача попадает в соответствующий класс, подходит для решения данным методом. Аналогично тому, как в математике, увидев уравнение, он должен понять, квадратное это уравнение или какое-то другое, т.е. уметь узнавать квадратные уравнения среди множества других. Итак, первое — уметь узнавать задачи из соответствующего класса;

2) уметь быстро и без ошибок применить метод к решению данной конкретной задачи (для формулы корней квадратного уравнения — уметь определить, чему равны в конкретном уравнении *a*, *b* и *c*, подставить их значения в формулу, провести вычисления и проверить правильность решения подстановкой в уравнение или по формуле Виета).

В каком-то смысле п. 1 — узнавание задач, решаемых данным методом — это самая сложная часть в овладении методом. И лично я уверен, что нет никакого иного пути научиться узнавать задачи, определять, какой метод можно применить, кроме как "решать, решать и еще раз решать" задачи на соответствующий метод, смотреть, как решает такие задачи учитель, как их решают другие, решать самому (на первых порах с помощью учителя) и т.д.

В § 16 последовательно изложено несколько методов алгоритмизации. Сами методы более или менее независимы, и мы их будем проходить также последовательно — метод за методом.

16.1. Метод 1 — "рекуррентные соотношения"

В учебнике этот метод излагается в несколько этапов. Я остановлюсь на всех этапах, и притом достаточно подробно, поскольку, как я уже говорил, этот параграф — самый сложный в курсе.

Что такое метод рекуррентных соотношений.

Сначала вступление. Очень часто встречаются задачи, в которых значения каких-то величин надо вычислять шаг за шагом по каким-то формулам, используя ранее вычисленные значения. Вспомните, например, последовательность Фибоначчи из самого первого параграфа (1, 1, 2, 3, 5, 8, 13, ... — каждый следующий элемент получается как сумма двух предыдущих). Для этой последовательности *n*-й элемент получается как сумма *n*−1 и *n*−2-го:

$$a_n = a_{n-1} + a_{n-2}$$

а первые два элемента (чтобы было с чего стартовать) задаются явно:

$$a_1 = 1,\ a_2 = 1.$$

Приведенные выше формулы, задающие последовательность, называются рекуррентным соотношением. Слово "рекуррентный" означает, что очередной элемент последовательности выражается через предыдущий или несколько предыдущих элементов.

Другой пример рекуррентной последовательности из рассматривавшихся в § 1 — последовательность степеней двойки:

$$a_n = 2 \cdot a_{n-1},\ \ a_1 = 2.$$

Обычно задача состоит в нахождении какого-то конкретного элемента такой последовательности, например, "найти 15-е число Фибоначчи" или "найти 20-ю степень двойки".

Итак, общая постановка задачи выглядит следующим образом: задана последовательность, в которой очередной элемент выражается через один или несколько предыдущих, и надо вычислить какой-то конкретный элемент этой последовательности. В таком случае для составления алгоритма можно воспользоваться одним из самых простейших методов алгоритмизации, который называется так же, как и соотношения, задающие последовательность, — метод рекуррентных соотношений.

Практически мы несколько раз пройдем по этому методу, постепенно углубляя наши знания.

Простейший пример рекуррентного соотношения.

Рассмотрим какую-нибудь рекуррентную последовательность, например, из учебника

$$a_i = 2 \cdot a_{i-1} - 1,\ a_1 = 2.$$

Если начать ее вычислять, то мы получим последовательность 2, 3, 5, 9, ... Если нам надо написать алгоритм для вычисления *n*-го члена этой последовательности, то мы легко можем написать алгоритм А78, приведенный в учебнике на с. 126:

```
алг цел элемент (арг цел n) (А78)
дано n>0
надо | знач=n-й элемент последовательности
нач цел i, цел таб a[1:n]
  | a[1]:=2
  | нц для i от 2 до n
  | |   a[i]:=2*a[i-1]-1
  | кц
  | знач:=a[n]
кон
```

Здесь мы еще до самого метода не дошли. Пока у нас просто имеется какая-то последовательность, заданная рекуррентным соотношением, и мы — один к одному — переписали ее в виде алгоритма. Для запоминания элементов последовательности ввели табличную величину, в первом элементе которой запомнили значение первого элемента последовательности, во втором — второго и т.д.

Это такой нулевой уровень — если есть рекуррентное соотношение, то можно записать алгоритм, который, буква в букву, соответствует формуле. Поэтому здесь пока никаких сложностей (а значит, и никаких методов и даже потребности в них) не возникает.

Основная идея метода рекуррентных соотношений — "исчезновение" индексов.

А вот дальше начинается собственно содержание. Прежде всего заметим, что если мы начнем вычислять *n*-й элемент такой последовательности сами — "вручную", то не станем запоминать ***все*** ранее вычисленные ее элементы. В этом нет никакой необходимости — ведь каждый следующий элемент вычисляется через предыдущий. При вычислении $a_5 = 2 \cdot a_4 - 1$, мы подставим $a_4 = 9$ и по формуле получим, что следующий элемент равен $2 \cdot 9 - 1 = 17$. Чтобы получить следующий элемент (a_6), в формулу надо подставлять полученное значение — $17(a_5)$. Про предыдущие значения (a_1, a_2, a_3, a_4) уже можно забыть — при дальнейших вычислениях они нам больше не понадобятся.

При вычислении нового элемента используется только один предыдущий элемент, и, следовательно, хранить все предыдущие элементы последовательности в памяти ЭВМ совершенно незачем — для вычислений они не нужны. Чтобы подсчитать следующий элемент, нужно знать только одно число — значение предыдущего элемента последовательности.

На этой основе алгоритм можно попытаться переписать без использования таблиц — ведь хранить надо только одно число. Этот алгоритм приведен в учебнике следующим (А79, стр. 126).

```
алг цел элемент (арг цел n) (А79)
дано n>0
надо | знач=n-й элемент последовательности
нач цел i, a
  | a:=2
  | | запоминание 1-го члена посл-ти
  | нц для i от 2 до n
  | |   | a:=2*a-1
  | |   | вычисление i-го члена посл-ти
  | кц
  | знач:=a
кон
```

Фактически в этом новом алгоритме всюду, где ранее использовалась таблица, написана одна и та же обычная величина a. Если сравнить эти два алгоритма — А78 и А79, то видно, что a[i] и a[i - 1] заменены на одну и ту же букву a без каких-либо индексов.

Это уже некоторое содержание изучаемого метода. Математические индексы, использовавшиеся при математическом описании (задании) последовательности и ***служащие в математике для обозначения разных элементов последовательности***, совершенно необязательно отображать в индексы таблиц. В алгоритме А78 математические обозначения a_i и a_{i-1} (т.е. разные элементы последовательности) соответствовали a[i] и a[i - 1] — т.е. разным элементам таблицы. А в алгоритме А79 они соответствуют ***одной и той же величине*** a, при этом математические обозначения a_i и a_{i-1} (разные элементы последовательности) соответствуют не самой этой величине, ***а ее значениям в разные моменты времени***, в разные моменты вычислений.

Ведь в отличие от статического (неизменного) математического задания последовательности рекуррентным соотношением в ходе выполнения алгоритма значения величин ***меняются***. В начале исполнения алгоритма А79, после выполнения команды a:= 2 значением величины a станет первый элемент последовательности — 2. Но уже после первого выполнения в теле цикла команды a:= 2 * a - 1 старое значение a послужит для

Кривые Гильберта и Серпинского, или Снова рекурсия

Окончание. См. с. 1, 2

```
d:=detect;
initgraph(d, r, PATH);
{Переход в графический режим}
Hscr:=GetMaxY+1;{Высота экрана}
Wscr:=GetMaxX+1;{Ширина экрана}
S:=round(PrS/100*Hscr); {Сторона квадрата}
h:=round(S/(Power2(n)-1)); {длина связок}
{Находим координаты начальной точки кривой.
Для ориентации: вверх и вправо начальная
точка - левая нижняя точка квадрата;
для ориентации: вниз и влево - правая
верхняя точка квадрата}
Case orient of
 1,3:{ориентация: вверх или вправо}
 begin
  x0:=Wscr div 2 - S div 2;
  y0:=Hscr div 2 + S div 2;
 end;
 2,4:{ориентация: вниз или влево}
 begin
  x0:=Wscr div 2 + S div 2;
  y0:=Hscr div 2 - S div 2;
 end;
end;{Case orient}
{Графический курсор устанавливаем в
начальную точку}
moveto(x0, y0);
{Рисуем соответствующий вариант кривой
Гильберта}
case orient of
  1: GU(n); 2: GD(n); 3: GR(n); 4: GL(n);
end {case orient};
readln; {Выход - нажатием клавиши Enter}
closegraph  {Переходим в текстовый режим}
END.
```

Язык Бейсик (вариант QuickBasic [5])

```
DECLARE SUB Delay (del&) 'Задержка
'Процедуры рисования четырех разновидностей
'кривых Гильберта
DECLARE SUB GL (i AS INTEGER)
DECLARE SUB GR (i AS INTEGER)
DECLARE SUB GU (i AS INTEGER)
DECLARE SUB GD (i AS INTEGER)
'Процедуры рисования связок
DECLARE SUB LineDown ()
DECLARE SUB LineUp ()
DECLARE SUB LineLeft ()
DECLARE SUB LineRight ()
'Описания переменных
DIM PrS AS SINGLE, S AS SINGLE, x0 AS
SINGLE, y0 AS SINGLE
DIM n AS INTEGER, orient AS INTEGER
DIM SHARED h AS SINGLE
'Константы
DIM SHARED del&: del& = 200000
'параметр задержки
Hscr! = 480: Wscr! = 640
'Высота и ширина экрана
'Основной алгоритм
CLS 'Чистка экрана
'Вводим исходные данные для построения
'кривой Гильберта
DO
PRINT "Введите длину стороны опорного
      квадрата";
INPUT " в % от высоты экрана ", PrS
LOOP UNTIL PrS < 100
INPUT "Введите порядок кривой ", n
DO
PRINT "Введите ориентацию кривой: ";
INPUT "вверх - 1, вниз - 2, вправо - 3,
      влево - 4 ", orient
LOOP UNTIL orient >= 1 AND orient <= 4
S = PrS / 100! * Hscr! 'Сторона квадрата
h = S / (2 ^ n - 1)  'длина связок
'Находим координаты начальной точки кривой.
'Для ориентации: вверх и вправо начальная
'точка - левая нижняя точка квадрата;
'для ориентации: вниз и влево -
'правая верхняя точка квадрата
IF orient = 1 OR orient = 3 THEN
 x0 = Wscr! / 2 - S / 2
 y0 = Hscr! / 2 + S / 2
ELSE
 x0 = Wscr! / 2 + S / 2
 y0 = Hscr! / 2 - S / 2
END IF
'Переход в графический режим для монитора
'VGA. Экран 640*480
SCREEN (12)
'Графический курсор устанавливаем в
'начальную точку
PSET (x0, y0)
'Рисуем соответствующий вариант кривой
'Гильберта
SELECT CASE orient
 CASE 1: CALL GU(n)
 CASE 2: CALL GD(n)
 CASE 3: CALL GR(n)
 CASE 4: CALL GL(n)
END SELECT
DO: LOOP UNTIL INKEY$ = CHR$(13)
'Выход - нажатием клавиши Enter
SCREEN 0     'Переходим в текстовый режим
END

SUB Delay (del&)'Процедура задержки
DIM i&
FOR i& = 1 TO del&: NEXT i&
END SUB

SUB GD (i AS INTEGER)
IF i > 0 THEN
    CALL GL(i - 1): CALL LineDown
    CALL GD(i - 1): CALL LineLeft
    CALL GD(i - 1): CALL LineUp
    CALL GR(i - 1): CALL Delay(del&)
END IF
END SUB

SUB GL (i AS INTEGER)
IF i > 0 THEN
      CALL GD(i - 1): CALL LineLeft
      CALL GL(i - 1): CALL LineDown
      CALL GL(i - 1): CALL LineRight
      CALL GU(i - 1): CALL Delay(del&)
END IF
END SUB

SUB GR (i AS INTEGER)
IF i > 0 THEN
      CALL GU(i - 1): CALL LineRight
      CALL GR(i - 1): CALL LineUp
      CALL GR(i - 1): CALL LineLeft
      CALL GD(i - 1): CALL Delay(del&)
END IF
END SUB

SUB GU (i AS INTEGER)
IF i > 0 THEN
    CALL GR(i - 1): CALL LineUp
    CALL GU(i - 1): CALL LineRight
    CALL GU(i - 1): CALL LineDown
    CALL GL(i - 1): CALL Delay(del&)
END IF
END SUB

SUB LineDown
 LINE -STEP(0, h)
END SUB

SUB LineLeft
 LINE -STEP(-h, 0)
END SUB

SUB LineRight
 LINE -STEP(h, 0)
END SUB

SUB LineUp
 LINE -STEP(0, -h)
END SUB
```

Язык Си

В языке Си для функций, обращение к которым предшествует их определению, надо указать прототип [7]. Такими функциями являются GD и GU.

```
#include <stdio.h>
#include <graphics.h>
#include <math.h>
#include <conio.h>
#include <dos.h>
#include<stdlib.h>
#define PATH ""
/* файлы *.BGI находятся в рабочем каталоге */
#define del 5000 /* Время задержки */
int h;
/* Прототипы функций GD и GU, вызываемых
    до своего определения                 */
void GD(int); void GU(int);
/*  Округление вещественного числа до
ближайшего целого */
int round(float a)
{return (int)floor(a+0.5);}
/*Функции рисования связок.  От последней
точки (на нее указывает графический курсор)
проводится вниз, вверх, влево, вправо отрезок
длиной h пикселей. Напомним, что ось Y
графического экрана направлена сверху вниз */
void LineDown() {linerel(0, h);}
void LineUp() {linerel(0, -h);}
void LineLeft() {linerel(-h,0);}
void LineRight() {linerel(h,0);}
/*Функции рисования четырех разновидностей
кривых Гильберта*/
void GL(int i)
{if (i>0)
 {GD(i-1); LineLeft();
  GL(i-1); LineDown();
  GL(i-1); LineRight();
  GU(i-1); delay(del);
 }
}
void GR(int i)
{if (i>0)
 {GU(i-1); LineRight();
  GR(i-1); LineUp();
  GR(i-1); LineLeft();
  GD(i-1); delay(del);
 }
}
void GU(int i)
 {
 if (i>0)
 {GR(i-1); LineUp();
  GU(i-1); LineRight();
  GU(i-1); LineDown();
  GL(i-1); delay(del);
 }
}
void GD(int i)
{if (i>0)
 {GL(i-1); LineDown();
  GD(i-1); LineLeft();
  GD(i-1); LineUp();
  GR(i-1); delay(del);
 }
}
void main()
{int d=DETECT,r,n,orient,x0, y0,
S,Hscr,Wscr;
 float PrS;
 clrscr();/*Чистка экрана*/
 /*Вводим исходные данные для построения
кривой Гильберта*/
 do
  {printf("\nВведите длину стороны опорного
квадрата");
   printf(" в % от высоты экрана ");
   scanf("%f",&PrS);
  }
 while (PrS>=100);
 printf("\nВведите порядок кривой ");
scanf("%d",&n);
 do
  {printf("\nВведите ориентацию кривой ");
   printf("вверх - 1, вниз - 2, вправо - 3,
          влево - 4 ");
   scanf("%d",&orient);
  }
 while (orient<1 || orient>4);
 initgraph(&d, &r, "");
/* Переход в графический режим */
 Hscr=getmaxy()+1;/*Высота экрана*/
 Wscr=getmaxx()+1;/*Ширина экрана*/
 S=round(PrS/100*Hscr); /*Сторона квадрата*/
 h=round(S/(pow(2,n)-1));/*длина связок*/
 /*Находим координаты начальной точки
кривой. Для ориентации: вверх и вправо
начальная точка - левая нижняя точка
квадрата; для ориентации: вниз и влево -
правая верхняя точка квадрата */
 if (orient == 1 || orient == 3)
  {x0=Wscr/2 - S/2; y0=Hscr/2 + S/2;}
 else {x0=Wscr/2 + S/2; y0=Hscr/2 - S/2;}
 /*Графический курсор устанавливаем в
начальную точку*/
 moveto(x0, y0);
 /*Рисуем соответствующий вариант кривой
Гильберта*/
 switch (orient)
  {case 1: GU(n); break;
   case 2: GD(n); break;
   case 3: GR(n); break;
   case 4: GL(n); break;
  }
 getch(); /*Выход - нажатием любой клавиши*/
 closegraph();
/*Переходим в текстовый режим*/
}
```

ЛИТЕРАТУРА

1. *Златопольский Д.М.* Рекурсия. Информатика, 1996, № 7, 8.
2. *Островский С.Л., Гольдшлаг О.Я.* Фрактальные кривые. Информатика, 1995, № 23.
3. *Кушниренко А.Г., Лебедев Г.В., Сворень Р.А.* Основы информатики и вычислительной техники. М.: Просвещение, 1990.
4. *Эпиктетов М.Г.* Почему школьный алгоритмический? Информатика, 1995, № 24.
5. *Фаронов В.В.* Программирование на персональных ЭВМ в среде ТУРБО-ПАСКАЛЬ. М.: Изд-во МГТУ, 1992.
6. *Зельднер Г.А.* QuickBasic 4.5. М.: ABF, 1994.
7. *Александров П.С.* Введение в общую теорию множеств и функций. М.: Гостехиздат, 1948.
8. *Романовская Л.М. и др.* Программирование в среде Си. М.: Финансы и статистика, 1992.

Окончание в следующем номере

ИНФОРМАТИКА ПОСЛЕ УРОКОВ

Вопросы для проведения школьных конкурсов “Что? Где? Когда?”, “Брейн-ринг”, викторин

Подготовил Д.М. ЗЛАТОПОЛЬСКИЙ

1. Какая марка компьютеров является полудрагоценной?

Ответ. “Агат”.

2. Какая поговорка описывает момент, когда закончится выполнение следующего цикла:

Школьный алгоритмический язык

```
нц
 | вывод нс, "Здравствуйте!"
кц при 2=1
```

Ответ. “Когда рак на горе свистнет”.

Паскаль

```
repeat writeln('Здравствуйте!');
until 2=1;
```

Ответ. “После дождичка в четверг”.

Бейсик (вариант QuickBasic)

```
DO PRINT "Здравствуйте!"
LOOP UNTIL 2=1
```

Ответ. Программист на QuickBasic ответил бы: “Когда LOOPнет мое терпение!”.

3. Программист попал в армию. Какой вопрос он задаст офицеру, давшему команду “По порядку номеров — рассчитайся!”?

Ответ. “А в какой системе счисления считать?”

4. В языке программирования, использованном в приведенном ниже фрагменте программы, допускается записывать зарезервированные слова, имена величин и константы на нескольких строчках. Какая рекурсивная функция описана в этом фрагменте? (Звездочками отмечены некоторые символы.)

```
F
U
NC
TIO
N ***
(******)
:************
BEGIN IF *** THEN ***
ELSE *********************** END;
```

Ответ. Функция для вычисления числа Фибоначчи.

5. О какой компьютерной программе идет речь в песне:

Он мне дорог с давних лет
И его милее нет —
Этих окон негасимый свет.

Ответ. Речь идет об операционной системе Windows, хотя некоторые слова песни изменены.

6. Какая связь между городом в Англии, ружьем калибра 30×30 и одним из элементов компьютера?

Ответ. Они все связаны со словом “винчестер”.

7. Перед вами стихотворение, написанное в 60-х годах программистом С.А. Маркиным:

Начало светлое весны...
Лесов зеленые массивы
Цветут. И липы, и осины,
И ели помыслы ясны.

Себе присвоил этот май
Права одеть листвою ветки,
И целый месяц в душах метки
Он расставляет невзначай...

И пишется легко строка,
И на этюдник рвутся кисти,
Уходит ложь в обличье истин,
И говорю я ей: пока!

Сколько слов, связанных с синтаксисом некоторого языка программирования, имеется в стихотворении (это могут быть так называемые зарезервированные слова этого языка, названия операторов, типов величин и т.п.)?

Ответ. 15 слов:
Начало
массивы
и, и
И
присвоил
И, метки
И, строка
И
ложь, истин
И, пока

8. Когда появился манипулятор типа “мышь”, то для него в русском языке некоторое время использовалось название по имени персонажа известной русской сказки. Назовите имя этого персонажа.

Ответ. “Колобок” (источник — Математический энциклопедический словарь, 1988 г.)

9. Почему на компьютерном жаргоне процессор называется камнем?

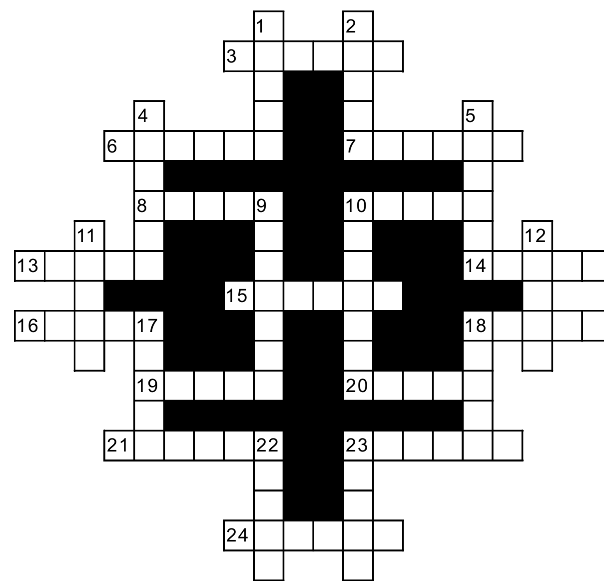
Ответ. Потому что основой микросхемы процессора является кристалл кремния высокой степени чистоты.

ВНЕКЛАССНАЯ РАБОТА ПО ИНФОРМАТИКЕ

Кроссворд

По горизонтали: 3. Упорядоченный набор данных одного типа. **6.** Структура или внешний вид файла. **7.** Язык программирования. **8.** Номер ячейки памяти. **10.** Алгоритмический язык. **13.** Пластина, на которой размещаются микропроцессор и другие микросхемы. **14.** Специально написанная программа, умеющая выполнять различные нежелательные для пользователя действия на компьютере. **15.** Американский ученый, давший математическое обоснование принципов устройства ЭВМ. **16.** Копия. **18.** Порция информации. **19.** Число символов в строке литерной величины. **20.** Отечественная система автоматизации программирования, созданная для ЭВМ М-220. **21.** Адресуемый элемент памяти. **23.** Специальная программа, используемая для измерения интервалов времени. **24.** Наука о способах доказательств и опровержений.

По вертикали: 1. Совокупность программ, заранее введенных со своими данными во внешнюю память. **2.** Американский математик, основоположник кибернетики. **4.** Перемещение перфокарт в перфораторе. **5.** Русский академик, кораблестроитель, построивший в 1911 году вычислительную машину для решения дифференциальных уравнений. **9.** Отрезок прямой, указывающий направление. **10.** Указание исполнителю. **11.** Мини-компьютер армянского производства. **12.** Трехэлектродная лампа, используемая в ЭВМ первого и второго поколений. **17.** Проблема, которую необходимо решить. **18.** Устройство для считывания графической или текстовой информации в компьютер. **22.** Алгоритмический язык. **23.** Знак, используемый для отделения целой части числа от дробной.



ОТВЕТЫ НА КРОССВОРД

По горизонтали: 3. Массив. 6. Формат. 7. Рапира. 8. Адрес. 10. Кобол. 13. Плата. 14. Вирус. 15. Нейман. 16. Образ. 18. Слово. 19. Длина. 20. Альфа. 21. Ячейка. 23. Таймер. 24. Логика.

По вертикали: 1. Пакет. 2. Винер. 4. Подача. 5. Крылов. 9. Стрелка. 10. Команда. 11. Наири. 12. Триод. 17. Задача. 18. Сканер. 22. Алгол. 23. Точка.

Кроссворд подготовил В.Г. Федоринов

ОБЪЕДИНЕНИЕ ПЕДАГОГИЧЕСКИХ ИЗДАНИЙ “ПЕРВОЕ СЕНТЯБРЯ”

Первое сентября — А.С. Соловейчик, индекс подписки — 32024
Английский язык — Е.В. Громушкина, индекс подписки — 32025
Биология — Н.Г. Иванова, индекс подписки — 32026
Воскресная школа — монах Киприан (Ященко), индекс подписки — 32742
География — О.Н. Коротова, индекс подписки — 32027
Здоровье детей — А.У. Лекманов, индекс подписки — 32033
Информатика — Е.Б. Докшицкая, индекс подписки — 32291
Искусство — Н.Х. Исмаилова, индекс подписки — 32584
История — А.Ю. Головатенко, индекс подписки — 32028
Литература — Г.Г. Красухин, индекс подписки — 32029
Математика — И.Л. Соловейчик, индекс подписки — 32030
Начальная школа — М.В. Соловейчик, индекс подписки — 32031
Немецкий язык — Gerolf Demmel, индекс подписки — 32292
Русский язык — Л.А. Гончар, индекс подписки — 32383
Спорт в школе — Н.В. Школьникова, индекс подписки — 32384
Управление школой — Н.А. Широкова, индекс подписки — 32652
Физика — Н.Д. Козлова, индекс подписки — 32032
Химия — О.Г. Блохина, индекс подписки — 32034
Школьный психолог — М.Н. Сартан, индекс подписки — 32898

Гл. редактор
Е.Б.Докшицкая
Зам. гл. редактора
С.Л.Островский
Редакция:
Л.Н.Картвелишвили, Ю.А.Соколинский, Н.Л.Беленькая, Н.П.Медведева
Дизайн и компьютерная верстка:
Н.И.Пронская
Корректоры:
Е.Л.Володина, С.М.Подберезина

©ИНФОРМАТИКА 1999

выходит четыре раза в месяц
При перепечатке ссылка на ИНФОРМАТИКУ обязательна, рукописи не возвращаются.
Регистрационный номер 012868

121165, Москва, Киевская, 24
тел. 249 4896
Отдел рекламы
тел. 240 1041

ИНДЕКС ПОДПИСКИ
для индивидуальных подписчиков 32291
для предприятий и организаций 32591
комплекта приложений 32744

Internet: infosef@glasnet.ru
Fidonet: 2:5020/69.32
WWW: http://www.1september.ru

Отпечатано с готовых диапозитивов редакции в типографии “ПРЕССА”, 125865, Москва, ул. Правды, 24

Тираж 7000 экз.
Заказ №

Start тел./факс (095)249 3138, факс (095)249 3184, тел. 249 3386

10.02