Дорошенко В. М.
Никифоров А. А.

Разработка методов, алгоритмов и программного обеспечения для телекоммуникационных, измерительных и управляющих систем

Учебное пособие

Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Саратовский государственный технический университет
имени Ю. А. Гагарина»

В. М. Дорошенко, А. А. Никифоров

РАЗРАБОТКА МЕТОДОВ, АЛГОРИТМОВ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ ТЕЛЕКОММУНИКАЦИОННЫХ, ИЗМЕРИТЕЛЬНЫХ И УПРАВЛЯЮЩИХ СИСТЕМ

Учебное пособие

Электронное издание
локального распространения

Санкт-Петербург
Наукоемкие технологии
2025

ISBN 978-5-907946-48-4

УДК 621.391:004(075.8)
ББК 32.97я73
Д69

Рецензент:
Дмитрий Александрович Зимняков, доктор физико-математических наук, профессор кафедры медицинской физики Саратовского национального исследовательского государственного университета имени Н. Г. Чернышевского

Д69 Дорошенко В. М., Никифоров А. А. Разработка методов, алгоритмов и программного обеспечения для телекоммуникационных, измерительных и управляющих систем [Электронный ресурс]: учебное пособие / В. М. Дорошенко, А. А. Никифоров. – Электрон, текстовые дан. (1,5 Мб). – СПб.: Наукоемкие технологии, 2025. – 82 с. – 1 электрон., опт. диск (CD-ROM).

ISBN 978-5-907946-48-4

Учебное пособие предназначено для студентов, обучающихся по направлениям подготовки 11.03.02 Инфокоммуникационные технологии и системы связи» по дисциплинам «Общая теория связи», «Информационные технологии в системах мобильной связи», «Вычислительная техника и информационные технологии», «Синтез технических систем», «Основы проектирования и САПР».

В учебном пособии приведены методы создания алгоритмов и программного обеспечения для различных сфер применения. Даны базовые понятия и методы работы с алгоритмами, массивами данных и структурами. Приведенные в пособии сведения направлены на улучшения навыков и компетенций по разработке ПО в телекоммуникационных, измерительных и управляющих системах.

Текстовое электронное издание

Минимальные системные требования:

- процессор: Intel x86, x64, AMD x86, x64 не менее 1 ГГц;
- оперативная память RAM ОЗУ: не менее 512 МБайт;
- свободное место на жестком диске (HDD): не менее 120 МБайт;
- операционная система: Windows XP и выше;
- Adobe Acrobat Reader;
- дисковод CD-ROM;
- мышь.

УДК 621.391:004(075.8)
ББК 32.97я73

ISBN 978-5-907946-48-4

Учебное издание

Дорошенко Валентина Михайловна
Никифоров Александр Анатольевич

Разработка методов, алгоритмов и программного обеспечения для телекоммуникационных, измерительных и управляющих систем

Учебное пособие

Электронное издание
локального распространения

Издательство «Наукоемкие технологии»
ООО «Корпорация «Интел Групп»
https://publishing.intelgr.com
E-mail: publishing@intelgr.com
Тел.: +7 (812) 945-50-63
Интернет-магазин издательства
https://shop.intelgr.com/

Подписано к использованию 26.03.2025 г.
Объем издания – 1,5 Мб.
Комплектация издания – 1 CD.
Тираж 500 CD.

ISBN 978-5-907946-48-4

ОГЛАВЛЕНИЕ

1 СИСТЕМЫ СЧИСЛЕНИЯ, ИСПОЛЬЗУЕМЫЕ В ЭВМ

1.1 Позиционные системы счисления

Система счисления – принятый способ записи чисел и сопоставления этим записям реальных значений. Все системы счисления можно разделить на три класса: **позиционные, непозиционные и смешанного типа.** Для записи чисел в различных системах счисления используется некоторое количество отличных друг от друга знаков. Число таких знаков в позиционной системе счисления называется **основанием системы счисления.** Ниже приведена табл. 1.1, содержащая наименования некоторых позиционных систем счисления и алфавита (перечня знаков/цифр), из которых образуются в них числа.

Таблица 1.1

Некоторые системы счисления

Основание	Система счисления	Алфавит (знаки/цифры)
2	Двоичная	0,1
3	Троичная	0,1,2
8	Восьмеричная	0,1,2,3,4,5,6,7
10	Десятичная	0,1,2,3,4,5,6,7,8,9
16	Шестнадцатеричная	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

В позиционной системе счисления число может быть представлено в виде суммы произведений коэффициентов Q_i на степени основания системы счисления B:

$$Q_nQ_{n-1}Q_{n-2} \ldots Q_1Q_0,Q_{-1},Q_{-2} \ldots =$$

$$= Q_n \times B^n + Q_{n-1} \times B^{n-1} + \ldots + Q_1 \times B^1 + Q_0 \times B^0 + Q_{-1} \times B^{-1} + Q_{-2} \times B^{-2} + \ldots$$

(знак «запятая» отделяет целую часть числа от дробной). Таким образом, значение каждого знака в числе зависит от позиции, которую занимает знак в записи числа. Именно поэтому такие системы счисления называют позиционными. Примеры (десятичный индекс в скобках внизу указывает основание системы счисления):

$$32,63_{(10)} = 3\times10^1 + 2\times10^0 + 6\times10^{-1} + 3\times10^{-2}$$

(в данном примере цифра 3 в одном случае означает число десятков, а в другом — число сотых долей единицы);

$$926_{(10)} = 9\times10^2 + 2\times10^1 + 6.$$

(«Девятьсот двадцать шесть» с формальной точки зрения представляется в виде «девять умножить на десять в степени два, плюс два умножить на десять в степени один, плюс шесть»).

$$1011_{(2)} = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0;$$

$$102_{(3)} = 1\times3^2 + 0\times3^1 + 2\times3^0;$$

$$647,3_{(8)} = 6\times8^2 + 4\times8^1 + 7\times^{80} + 3\times8^{-1};$$

$$D1E,C_{(16)} = D\times16^2 + 1\times16^1 + E\times16^0 + C\times16^{-1}.$$

В современных персональных компьютерах (ПК) параллельно используются несколько позиционных систем счисления (как правило, двоичная, восьмеричная, десятичная и шестнадцатеричная), поэтому большое практическое значение имеют процедуры перевода чисел из одной системы счисления в другую. Заметим, что во всех приведенных выше примерах результат является десятичным числом, и, таким образом, способ перевода в десятичную систему чисел из любой позиционной системы счисления уже продемонстрирован.

Чтобы перевести целую часть числа из десятичной системы в систему с основанием *B,* необходимо разделить ее на *B*. Остаток даст младший разряд числа. Полученное при этом частное необходимо вновь разделить на *B* — остаток даст следующий разряд числа и т.д. Для перевода дробной части ее необходимо умножить на *B*. Целая часть полученного произведения будет первым (после запятой, отделяющей целую часть от дробной) знаком. Дробную же часть произведения необходимо вновь умножить на *B*. Целая часть полученного числа будет следующим знаком и т.д.

Необходимо упомянуть, что кроме рассмотренных выше позиционных систем счисления существуют такие, в которых значение знака не зависит от того места, которое он занимает в числе. Такие системы счисления называются не-

позиционными. Наиболее известным примером непозиционной системы является так называемая «римская». В этой системе используется 7 знаков (I, V, X, L, C, D, M), которые соответствуют следующим величинам:

I(1) V(5) X(10) L(50) C(100) D(500) M(1000)

Примеры: III (три), LIX (пятьдесят девять), DLV (пятьсот пятьдесят пять).

Строго говоря, римская система счисления не является непозиционной в чистом виде. Действительно, в ней значения некоторых чисел, состоящих из одинаковых знаков, зависят от порядка их следования. Например, записи в римской системе IX и XI обозначают разные числа, соответственно 9 и 11. Другой пример XLIX и LXXI обозначают соответственно числа 49 и 71.

Кроме очевидного неудобства записи чисел, основным недостатком непозиционных систем, из-за которых они представляют лишь исторический интерес, является отсутствие формальных правил записи чисел и, соответственно, арифметических действий над ними. Однако традиционно римскими числами часто пользуются при нумерации глав в книгах, веков в истории и др.

1.2 Двоичная система счисления

Особая значимость двоичной системы счисления в информатике определяется тем, что внутреннее представление любой информации в компьютере является двоичным, т.е. описываемым наборами только из двух знаков (0 и 1).

Конкретизируем описанный выше способ в случае перевода чисел из десятичной системы в двоичную. Целая и дробная части переводятся порознь. Для перевода целой части (или просто целого) числа необходимо разделить ее на основание системы счисления и продолжать делить частные от деления до тех пор, пока частное не станет равным 0. Значения получившихся остатков, взятые в обратной последовательности, образуют искомое двоичное число. Например, переведем из десятичной системы число 29,12 в двоичную систему счисления. Следуя указанному выше правилу, переводим сначала целую часть числа:

Остаток

29:2=14 (1),

14:2 = 7 (0),

7:2 = 3 (1),

3:2=1 (1),

1:2 = 0 (1).

Таким образом

$$29_{(10)} = 11101_{(2)}.$$

Для перевода дробной части (или числа, у которого «0» целых) надо умножить ее на 2. Целая часть произведения будет первой цифрой числа в двоичной системе. Затем, отбрасывая целую часть у результата, вновь умножаем на 2 и т.д. Заметим, что конечная десятичная дробь при этом вполне может стать бесконечной (периодической) двоичной. Переводим дробную часть 0,12:

0,12 × 2 = 0,24 (целая часть 0),

0,24 × 2 = 0,48 (целая часть 0),

0,48 × 2 = 0,96 (целая часть 0),

0,96 × 2 = 1,92 (целая часть 1),

0,92 × 2 = 1,84 (целая часть 1),

0,84 × 2 = 1,68 (целая часть 1), и т.д.

В итоге

$$0{,}12_{(10)} = 0{,}0001111..._{(2)}$$

Над числами, записанными в любой системе счисления, можно производить различные арифметические операции. Так, для сложения и умножения двоичных чисел необходимо использовать табл. 1.2.

Таблица 1.2

Таблицы сложения и умножения в двоичной системе

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

Заметим, что при двоичном сложении 1 + 1 возникает перенос единицы в старший разряд — точь-в-точь как в десятичной арифметике:

$$\begin{array}{r} 1001 \\ +\ \ 11 \\ \hline = 1100 \end{array}$$

или

$$\begin{array}{r} 1001 \\ \times\ \ \ 11 \\ \hline 1001 \\ +\ 1001\ \ \ \\ \hline = 11011 \end{array}$$

1.3 Восьмеричная и шестнадцатеричная системы счисления

С точки зрения изучения принципов представления и обработки информации в компьютере, обсуждаемые в этом пункте системы представляют большой интерес. Хотя компьютер «знает» только двоичную систему счисления, часто с целью уменьшения количества записываемых знаков бывает удобнее пользоваться восьмеричными или шестнадцатеричными числами (например, адреса ячеек памяти в современных ПК записываются в шестнадцатеричной системе). Более того, далее будет показано, что процедура взаимного перевода чисел из каждой из этих систем в двоичную очень проста — гораздо проще переводов между любой из этих трех систем и десятичной.

Перевод чисел из десятичной системы счисления в восьмеричную производится (по аналогии с двоичной системой счисления) с помощью делений и умножений на 8. Например, переведем число $68{,}34_{(10)}$:

$68 : 8 = 8$ (4 в остатке),

$8 : 8 = 1$ (0 в остатке),

$1 : 8 = 0$ (1 в остатке)

$0{,}34 \times 8 = 2{,}72$, (целая часть 2)

$0{,}72 \times 8 = 5{,}76$, (целая часть 5)

$0{,}76 \times 8 = 6{,}08$ ...

Таким образом, $68{,}34_{(10)} = 104{,}256..._{(8)}$ (из конечной дроби в одной системе может получиться бесконечная дробь в другой).

Перевод чисел из десятичной системы счисления в шестнадцатеричную производится аналогично.

С практической точки зрения представляет интерес процедура взаимного преобразования двоичных, восьмеричных и шестнадцатеричных чисел. Для этого воспользуемся табл. 1.3 чисел от 0 до 15 (в десятичной системе счисления), представленных в других системах счисления.

Для перевода целого двоичного числа в восьмеричное необходимо разбить его справа налево на группы по 3 цифры (самая левая группа может содержать менее трех двоичных цифр), а затем каждой группе поставить в соответствие ее восьмеричный эквивалент из табл. 1.3. Например:

$10011011001 = 10\ 011\ 011\ 001$, т.е. $10011011001_{(2)} = 2331_{(8)}$.

Заметим, что группу из трех двоичных цифр часто называют «двоичной триадой». Перевод целого двоичного числа в шестнадцатеричное производится путем разбиения данного числа на группы по 4 цифры — «двоичные тетрады»:

$10111100011011001 = 1\ 0111\ 1000\ 1101\ 1001,$

т.е. $10111100011011001_{(2)} = 178D9_{(16)}$.

Для перевода дробных частей двоичных чисел в восьмеричную или шестнадцатеричную системы аналогичное разбиение на триады или тетрады производится от запятой вправо (с дополнением недостающих последних цифр нулями):

$$0{,}1100011101011_{(2)} = 0{,}110\ 001\ 110\ 101\ 100 = 0{,}61654_{(8)},$$

$$0{,}1100011101_{(2)} = 0{,}1100\ 0111\ 0100 = 0{,}C74_{(16)}.$$

Таблица 1.3

Соответствие чисел в различных системах счисления

Десятичная	**Шестнадцатеричная**	**Восьмеричная**	**Двоичная**
0	0	0	0
1	1	1	1
3	3	3	11
4	4	4	100
5	5	5	101
7	7	7	111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	T	16	1110
15	F	17	1111

Перевод восьмеричных (шестнадцатеричных) чисел в двоичные производится обратным путем — сопоставлением каждому знаку числа соответствующей тройки (четверки) двоичных цифр.

Преобразования чисел из двоичной в восьмеричную и шестнадцатеричную системы и наоборот столь просты (по сравнению с операциями между этими тремя системами и привычной нам десятичной) потому, что числа 8 и 16 являются целыми степенями числа 2. Этой простотой и объясняется популярность восьмеричной и шестнадцатеричной систем в вычислительной технике и программировании.

Арифметические действия с числами в восьмеричной и шестнадцатеричной системах счисления выполняются по аналогии с двоичной и десятичной системами. Для этого необходимо воспользоваться соответствующими таблицами. Для примера табл. 1.4 иллюстрирует сложение и умножение восьмеричных чисел.

Рассмотрим еще один возможный способ перевода чисел из одной позиционной системы счисления в другую – *метод вычитания степеней.* В этом случае из числа последовательно вычитается максимально допустимая степень требуемого основания, умноженная на максимально возможный коэффициент, меньший основания; этот коэффициент и является значащей цифрой числа в новой системе. Например, число $114_{(10)}$:

$$114–2^6=114–64 = 50,$$

$$50–2^5 = 50–32=18,$$

$$18–2^4 = 2,$$

$$2–2^1= 0.$$

Таким образом, $114_{(10)} = 1110010_{(2)}$.

$$114–1\times8^2=114–64 = 50,$$

$$50–6\times8^1 = 50–48 = 2,$$

$$2–2\times8^0 = 2–2 = 0.$$

Итак, $114_{(10)} = 162_{(8)}$.

Таблица 1.4

Таблицы сложения и умножения в восьмеричной системе

Сложение

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

Умножение

×	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	10	12	14	16
3	0	3	6	11	14	17	22	25
4	0	4	10	14	20	24	30	34
5	0	5	12	17	24	31	36	43
6	0	6	14	22	30	36	44	52
7	0	7	16	25	34	43	52	61

Попробуйте самостоятельно составить таблицы сложения и умножения чисел в шестнадцатеричной системе счисления.

2 АЛГОРИТМЫ

В повседневной жизни каждый человек сталкивается с необходимостью решения задач самой разной сложности. Некоторые из них трудны и требуют длительных размышлений для поиска решений (а иногда его так и не удается найти), другие же, напротив, столь просты и привычны, что решаются автоматически. При этом выполнение даже самой простой задачи осуществляется в несколько последовательных этапов (шагов). В виде последовательности шагов можно описать процесс решения многих задач, известных из школьного курса математики: приведение дробей к общему знаменателю, решение системы линейных уравнений путем последовательного исключения неизвестных, построение треугольника по трем сторонам с помощью циркуля и линейки и т.д. Такая последовательность шагов в решении задачи называется алгоритмом. Каждое отдельное действие — это шаг алгоритма.

Решение любой научно-технической задачи, в настоящее время, немыслимо без применения вычислительной техники. Как начертал Никлаус Вирт (создатель алгоритмического языка PASCAL): «Программы = алгоритмы + структуры данных». Этим самым он хотел подчеркнуть, что основой программирования является не знание большого количества алгоритмических языков, а способность строить алгоритмы решения задач, а также знание и умение использовать различные структуры данных, применяемые для решения конкретных задач.

Поэтому в данном курсе мы посвятим основное внимание именно этим двум аспектам программирования, а для демонстрации примеров будем использовать наиболее подходящий алгоритмический язык высокого уровня C (C++).

2.1 Начальные сведения об алгоритмах

Процесс решения любого достаточно сложного научно-технического задания включает в себя, как правило, следующие этапы:

1. **Постановка задачи.** Основное требование к постановке задачи — дать достаточное количество информации для ее решения. Очень часто постанов-

ка задачи выполняется не программистом, а «заказчиком», вообще не знающим программирования. Программист является «исполнителем» заказа. От него требуется добиться от заказчика как можно более полной информации о решаемой задаче.

2. **Формализация задачи и моделирование**. Формализация предполагает замену словесной формулировки решаемой задачи ее математическим описанием (разработкой **математической модели** задачи). Моделирование задачи является важнейшим этапом, целью которого является поиск общей концепции решения. Обычно моделирование выполняется путем выдвижения гипотез решения задачи и их проверки любым рациональным способом (прикидочные расчеты, физическое моделирование и т.д.). Результатом каждой проверки является либо принятие гипотезы, либо отказ от нее и разработка новой. При моделировании важно иметь опыт программирования, знать возможности компьютера и языка программирования и выдвигать гипотезы с учетом этих возможностей. Помимо идеи решения задачи, результатами этого этапа должны быть формализованная постановка задачи.

3. **Выбор метода вычислений**. Во многом данный этап определяется предыдущим. Метод вычислений определяется, исходя из возможностей компьютера и математической сложности решаемой задачи.

4. **Разработка алгоритма.** Этот этап представляет собой реализацию идеи решения задачи. К разработке алгоритма следует приступать только после принятия гипотезы решения задачи. Данный этап является основой успешного решения всей задачи.

5. Тестирование алгоритма. Этап предполагает проверку алгоритма «вручную» с использованием подготовленных ранее контрольных примеров. Для сложных задач этот этап может оказаться весьма трудоемким, поэтому опытные программисты пропускают его и тестируют саму программу.

6. **Программирование алгоритма.** Программирование является формальной записью алгоритма средствами языка программирования, доступного на конкретном ПК.

7. **Отладка и тестирование программы**. Тестирование выполняется путем вывода промежуточных результатов работы программы и сравнения их с контрольным примером. Для этого либо используют специальные средства отладки программ, имеющиеся в интегрированной среде языка программирования, либо временно добавляют в программу команды вывода промежуточных значений. Уменьшить трудоемкость поиска ошибок в программе можно более тщательным проектированием алгоритма и планированием процесса тестирования на ранних стадиях разработки программы.

8. **Эксплуатация программы и интерпретация результатов.** В сложных программах может быть недостаточно тестирования для устранения всех ошибок. Очень часто они обнаруживаются на стадии эксплуатации заказчиком.

Успех в разработке программы зависит от двух основных факторов: соблюдения методики разработки алгоритмов, о которой речь пойдет ниже, и от опыта программирования. Не следует игнорировать или недооценивать этапы проектирования алгоритма (1 – 5), выполняемые вне компьютера.

Этапы 4-7 называют собственно программированием.

Рассмотрим четвертый этап более подробно.

2.1.1 Определение алгоритма

Единого «истинного» определения понятия «алгоритм» нет. Приведем для примера определения алгоритма, данные известными учеными и методистами.

«Алгоритм — это конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определенность, ввод, вывод, эффективность». (Д. Э. Кнут — американский преподаватель и идеолог программирования, автор ряда классических книг по программированию и алгоритмам).

«Алгоритм — это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи». (А.Н. Колмогоров, российский и советский математик, академик Российской академии наук).

«Алгоритм — это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату» (А.А. Марков, основоположник советской школы конструктивной математики).

«Алгоритм — строго детерминированная последовательность действий, описывающая процесс преобразования объекта из начального состояния в конечное, записанная с помощью понятных исполнителю команд». (Н.Д. Угринович, известный российский методист в области информатики).

Попробуем сформировать определение алгоритма, наиболее понятное при изучении основ программирования и алгоритмизации, основываясь на вышеприведенных определениях классиков.

Алгоритм — формальное описание порядка операций, сформулированное на понятном исполнителю языке, которое определяет процесс перехода от допустимых исходных данных к некоторому искомому результату и обладает свойствами массовости, конечности, определенности, детерминированности. В программировании алгоритм является фундаментом программы.

Понятие алгоритма в математике является более широким, чем алгоритм в программировании.

2.1.2 Формальные свойства алгоритмов

Различные определения алгоритма в явной или неявной форме содержат следующий ряд общих требований:

- **Дискретность** — алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени, то есть преобразование исходных данных в результат осуществляется во времени дискретно.
- **Детерминированность** (определенность). В каждый момент времени следующий шаг работы однозначно определяется состоянием системы. Таким образом, алгоритм выдает один и тот же результат (ответ) для одних и тех же исходных данных.

- **Понятность** — алгоритм для исполнителя должен включать только те команды и операции, которые ему (исполнителю) доступны («элементарные операции») и входят в его систему команд и операций.
- **Завершаемость** (конечность) — при корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов.
- **Массовость** (универсальность). Алгоритм должен быть применим к разным наборам исходных данных. Алгоритм служит, как правило, для решения не одной конкретной задачи, а некоторого класса задач. Так, алгоритм сложения применим к любой паре натуральных чисел. В этом выражается его свойство массовости, то есть возможности применять многократно один и тот же алгоритм для любой задачи одного класса.
- **Результативность** — завершение алгоритма определенными результатами.
- Алгоритм содержит ошибки, если приводит к получению неправильных результатов либо не дает результатов вовсе.
- Алгоритм не содержит ошибок, если он дает правильные результаты для любых допустимых исходных данных.

Элементарная операция — последовательность действий, не требующих дальнейших пояснений.

Для ПК это, например, записать число в оперативную память регистра с номером 6, сложить это число с числом из ячейки памяти 9 регистра, умножить два числа и т.д. Для подготовленного математика вычисление значения $\int_b^a f(x)dx$ можно тоже считать элементарной операцией, поскольку для него не возникает вопросов, как это сделать. Из приведенных примеров видно, что понятие элементарности операций зависит от того, кому предназначено описание алгоритма.

Как уже говорилось, действия в схеме алгоритма должны быть элементарными. Договоримся теперь, что мы будем понимать под элементарными дей-

ствиями, и какое их содержание, применимо к программированию на компьютере. Будем вкладывать в понятие элементарности следующее:

1. Всякий раз, когда мы записываем константу, т.е. будем считать, что это число помещается в ячейку памяти ПК.
2. При записи в схеме алгоритма переменной величины, обозначаемой буквой (комбинацией букв или цифр), как правило, латинского алфавита, будем полагать, что ПК отводит под эту переменную ячейку памяти. Все числовые значения, которые принимает эта переменная, будут записываться в эту ячейку памяти. Притом каждое новое числовое значение стирает в ячейке старое.
3. Процесс записи числового значения в ячейку памяти, отведенную для данной переменной, будем записывать при помощи операции присваивания численного значения, обозначаемого = (в данном случае это знак присваивания, а не равенства!).

    1) a=3.14;
    2) b=1.25E8;
    3) c=0.85E-9;
    4) a=c;
    5) b=a;
    6) c=1.2E12;

После выполнения действия 4 значение 3.14 в ячейке a будет стерто, и вместо него будет значение из ячейки с равное $0{,}85 \cdot 10^{-9}$, тоже после выполнения действий 5 и 6.

4. В соответствии с п. 3 возможны следующие вычислительные действия:

    1) x=5;
    2) x=x–2;
    3) x=x*x+x*x*x;

Во втором действии производиться следующая элементарная операция. Из ячейки x вызывается числовое значение, которое согласно действию 1 было равно 5, из этого числа вычитается число 2 и результат (число 3) записывается в ячейку x, стирая старое значение.

В действии 3 содержимое ячейки возводиться в квадрат (9), затем из него вычисляется куб (27), полученные числа складываются (36) и результат записывается в ячейку x, стирая предыдущее значение (3).

Общее правило выполнения операций присваивания заключается в следующем:

1) производиться арифметическое действие, записанные в правой части операции.
2) Результат записывается в ячейку памяти, обозначенную слева от знака =.

В математике обычно действия типа 2 и 3 заменяются при помощи рекурентных формул и индексированных переменных.

$$X_i = X_{i-1} - 2;$$

$$X_i = X_{i-1}^2 + X_{i-1}^3 .$$

5. Элементарными операциями будут считать:

- Арифметические операции: +, –, *, /.
- Операции сравнения: <, >, <=, >=, ==
- Операции элементарных функций: sin(x), cos(x), tg(b), ln(r), log(r), e^x, a^x и т.д.
- Логические операции и (&&), или (||), не (!)
- Операция нахождения модуля числа abs(x)

6. Некоторые операции, которые в смысле программирования не являются не элементарными, и в алгоритме они будут обозначаться предопределенным процессом.

2.1.3 Представление алгоритмов

Алгоритмы можно записывать различными способами. Например, можно записывать алгоритма на родном языке, помечая каждый его шаг соответствующим номером. В этом случае он будет представлен в виде занумерованной последовательности действий, но это не всегда удобно.

Для того чтобы сделать алгоритм понятным и наглядным, прибегают к помощи структурной схемы (блок-схемы) — одному из распространенных способов записи алгоритмов. Блок-схема представляет собой набор элементов (бло-

ков различной геометрической формы), соединенных стрелками (потоками информации), указывающими, какой блок является следующим по выполнению. В этом случае каждый блок — это «шаг» алгоритма.

Блок-схема алгоритма — это чертеж. Каждая операция алгоритма отображается на нем графическими символами — геометрической фигурой, в контуре которой записано содержание операции.

В соответствии с ГОСТ 19.701-90 «Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения» используются обозначения блоков на схеме алгоритма, приведенные в таблице 2.1.

Отметим некоторые важные черты, характеризирующие схему алгоритма.

1. Процесс, описываемый схемой алгоритма, должен быть разбит на отдельные шаги. Каждый шаг должен состоять из элементарных действий.
2. Выполнение алгоритма начинается с действия, которое указанно в схеме первым.
3. В описании любого определенного действия может быть указан его приемник. Если приемник не указан явно, то выполняется действие, расположенное в схеме алгоритма следующим, в данный момент.
4. Алгоритм должен быть составлен так, чтобы от исполнения не требовалось таких качеств, как находчивость, воображение, понимание смысла задачи.

Таблица 2.1

Наименование	Обозначение	Функция
Блок начало-конец (пуск-остановка)		Элемент отображает вход из внешней среды или выход из нее (наиболее частое применение — начало и конец программы). Внутри фигуры записывается соответствующее действие.
Блок вычислений (вычислительный блок)		Выполнение одной или нескольких операций, обработка данных любого вида (изменение значения данных, формы представления). Внутри фигуры записывают непосредственно сами операции, например: $a = 10 \times b + c$.

Продолжение таблицы 2.1

Логический блок (блок условия)		Отображает ветвление алгоритма с одним входом и двумя или более альтернативными выходами, из которых только один может быть выбран после вычисления условий, определенных внутри этого элемента.
Предопределенный процесс		Символ отображает выполнение процесса, состоящего из одной или нескольких операций, который определен в другом месте программы (в подпрограмме, модуле). Внутри символа записывается название процесса и передаваемые в него данные. Например, в программировании — вызов функции.
Данные (ввод-вывод)		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод). Данный символ не определяет носителя данных
Граница цикла		Символ заголовка цикла.
Соединитель		Символ отображает вход в часть схемы и выход из другой части этой схемы. Используется для обрыва линии и продолжения ее в другом месте (если схема состоит из нескольких страниц). Соответствующие соединительные символы должны иметь одинаковое (при том уникальное) обозначение.
Комментарий		Используется для более подробного описания шага, процесса или группы процессов. Описание помещается со стороны квадратной скобки и охватывается ей по всей высоте. Пунктирная линия идет к описываемому элементу, либо группе элементов (при этом группа выделяется замкнутой пунктирной линией.

2.2 Типовые структуры алгоритмов

Алгоритм синтезируется из типовых структур, которые необходимо запомнить. Чем больше типовых структур знает программист, тем производи-

тельней его труд. Известный голландский алгоритмист Э.В. Дейкстра (1930-2002) доказал, что при составлении алгоритмов достаточно использования всего трех разных типовых структур: линейных, ветвящихся и циклических.

2.2.1 Алгоритм линейной структуры

Это алгоритм, все символы которого изображаются на схеме в той последовательности, в какой должны быть выполнены действия. Такой порядок действия называется естественным.

Пример: вычислить

$$Z = (tg(b \cdot x - c) - ctg(-b \cdot x + c)) / (b \cdot x - c)^3 .$$

Возможный вариант алгоритма, реализующий эту задачу, показан на рисунке 2.1.

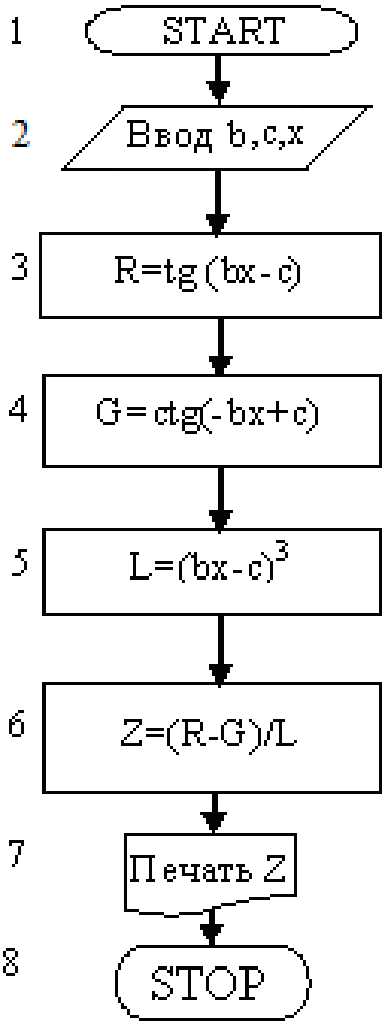


Рисунок 2.1

Очевидно, что количество действий для вычисления записанной схемы нерационально. Несложно заметить, что величину (*b*x–c*) необходимо вычислить один раз, запомнить полученное значение и использовать его в дальнейшем. Тогда схема примет вид, показанный на рисунке 2.2.

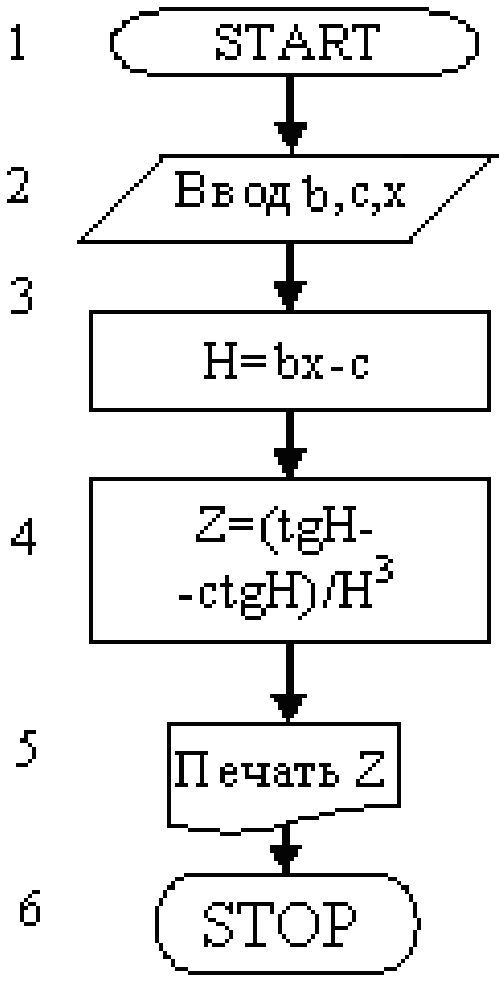


Рисунок 2.2

Схему можно еще упростить, если еще учесть, что *ctg*(*–x*)=*–ctg*(*x*) и *tg*(*x*)=1/*ctg*(*x*) (рисунок 2.3).

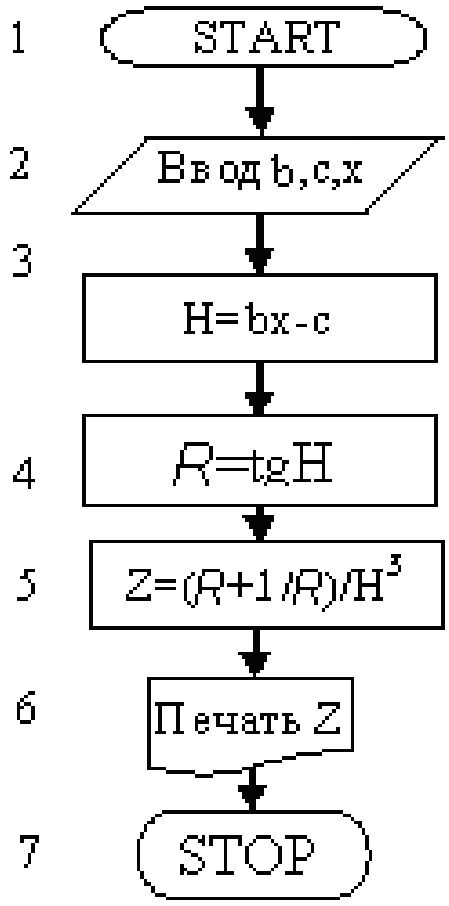


Рисунок 2.3

Заметим, что в последних схемах введены обозначения *tg*(*H*) и переменная *R*, т.е. зарезервировали переменную ячейки памяти. Однако значение *R* для нашей задачи не является конечным результатом. Это значение нас не интересует, оно промежуточное. Поэтому в смысле экономии памяти ПК можно схему еще улучшить. Давайте запишем результат в *tg*(*x*) в ячейку *Z* для хранения, рисунок 2.4.

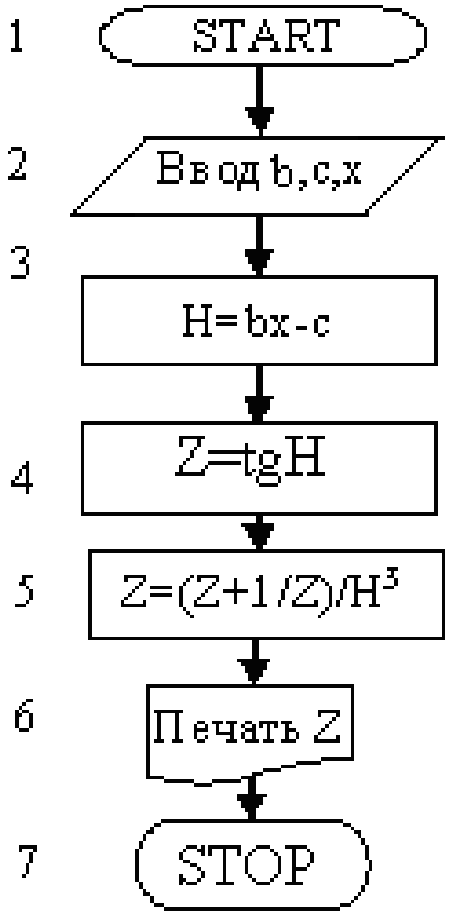


Рисунок 2.4

После составления схемы всегда надо составить контрольный пример и «прорешать» его. Особенно это надо делать начинающим программистам, т.к. они, как правило, подразумевают в схеме некоторые действия, не являющимися элементарными. Хотя основным исполнителем алгоритмов и программ является компьютер, на стадии тестирования исполнителем может быть сам алгоритмист.

Просчет по блок-схеме

1) Вариант данных 1: *B=2; X=0.5; C=3*:

H=2·0,5–3=–2;

Z=tg(–2)=–0,034917 (используется калькулятор);

$Z=(Z+1/Z)/H^3=(-0,034917+1/(-0,034917))/(-2)^3=3,5838731$.

2) Вариант данных 2: *B=4; X=2; C=8*:

H=4·2–8=0;

Z=tg(0)=–0 (используется калькулятор!);

$Z=(Z+1/Z)/H^3=(0+1/(0))/(0)^3$ действие не выполнимо.

Последний пример показывает, что алгоритм может быть невыполнимым.

К ограничениям, не допускающим выполнение алгоритма, относятся:

1. Деление на 0 или на разность близких чисел, которая для ПК считается нулем.
2. ln(0)=(–∞) и ln(–*N*), где *N* — положительное число.
3. arcsin(*x*), arccos(*x*) при *x*>1.
4. tg(*x*), *x*=±π/2 ± *k*π, где *k* — целое число.
5. Исчезновение порядка, когда число меньше, чем $5\cdot10^{-79}$, при подстановке в ячейку памяти ПК становиться отрицательным; чтобы понять это, необходимо знать, как представляются числа в конкретном ПК.

При разборке алгоритма пользователь должен предугадывать такие аварийные ситуации.

2.2.2 Алгоритм развертывания структуры

Это алгоритмы, в которых предусмотрено развертывание выполненной последовательности действий в зависимости от результата проверки некоторого логического условия.

Рассмотрим алгоритм из предыдущего раздела и предусмотрим печать сообщения о его невыполнимости при *H* = 0 и *tgH* = 0. (рисунок 2.5).

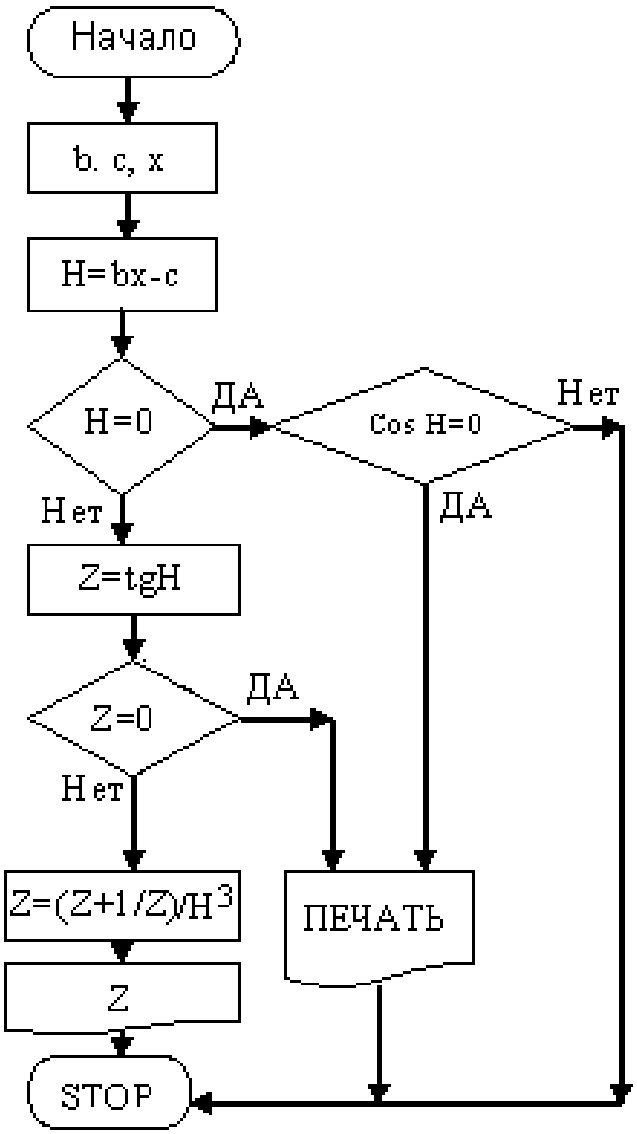


Рисунок 2.5

Значок ветвления решения в виде ромба позволяет записывать развертывания по двум или, что используется гораздо реже, трем ветвям. На рис. 2.6 показаны два таких возможных случая. Проверяется некоторое логическое условие. В первом случае величина *A* сравнивается с нулем. В зависимости от того, больше *A* нуля, равно или меньше его, выбирается соответствующая ветвь алгоритма. Во втором — тоже проверяется логическое условие. Если $A > 0$, алгоритм выполняется по ветви (Да), в противном случае он выполняется по ветви (Нет).

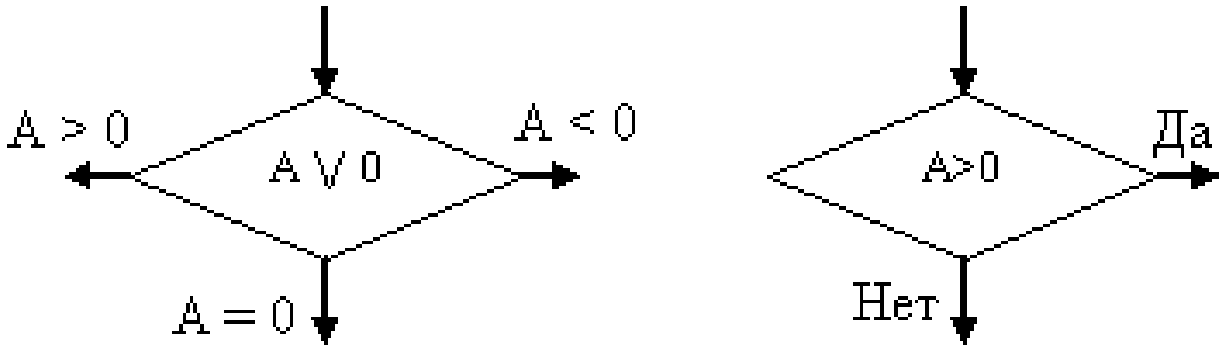


Рисунок 2.6

Пример 1. Вычислить $y = \begin{cases} x^2 + 2, \text{если } A \le x \le B \\ \cos(x) + \sin(x), \text{если } A > x \\ \text{tg}x, \text{если } x > B \end{cases}$

Блок-схема алгоритма, решающего данную задачу, показана на рис. 2.7.

В последнем примере возможно применение такой схемы, если при вычислении cos$x \neq 0$, поэтому правильнее было перед вычислением tg$x$ проверить выполнение этого условия. Предлагается сделать это читателю самостоятельно.

Приведем еще один поучительный пример, показывающий, что при создании алгоритмов необходимо несколько по-другому смотреть на задачу, чем это делается в математике.

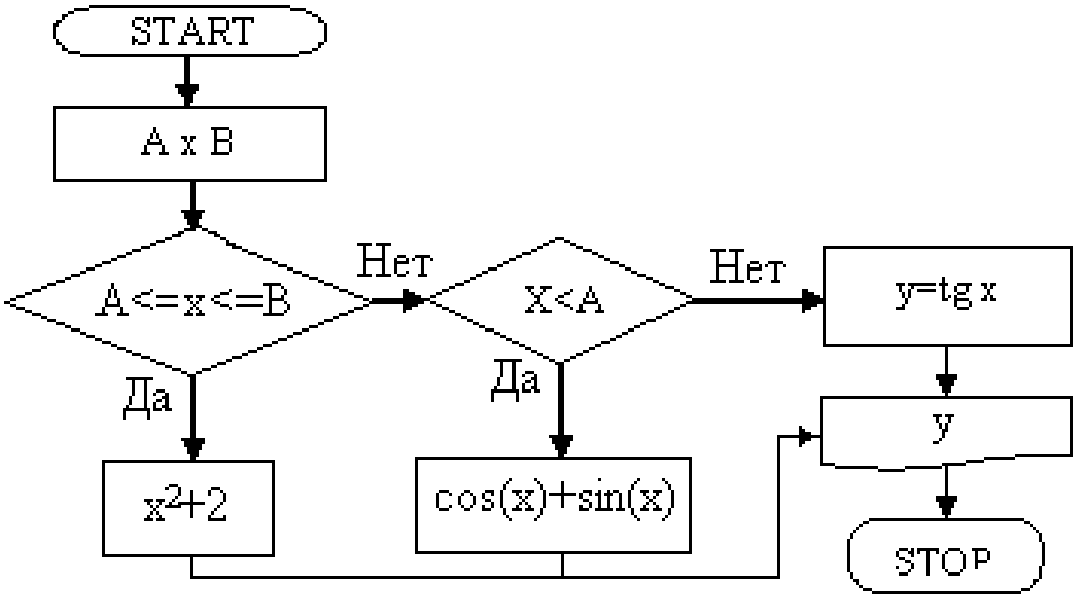


Рисунок 2.7

Пример 2. Составить алгоритм решения квадратного уравнения $ax^2+bx+c=0$, когда заданы три его коэффициента a,b,c. Если мы решаем квадратное уравнение, то, как правило, начинаем с проверки неотрицательности дискриминанта $d = b^2 – 4ac \geq 0$. После чего выносится решение о корнях уравнения. Поэтому алгоритм расчета корней мог бы быть следующим (рисунок 2.8).

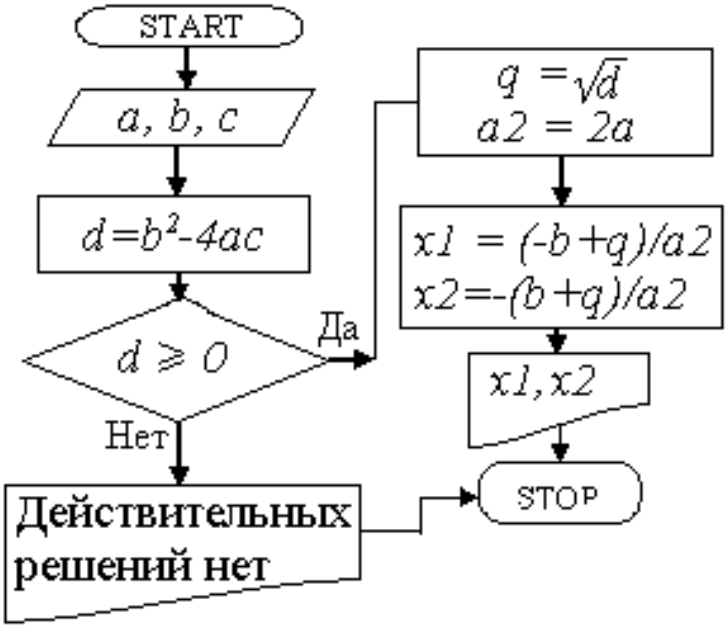


Рисунок 2.8

Теперь, проведем контрольную проверку данного алгоритма. Легко убедиться, что набор входных данных: $a=0$, $b=2$, $c=3$ приведет к неправильной его работе, потому что получается «деление на ноль». Естественно, когда мы решаем квадратное уравнение «вручную», равенство первого коэффициента нулю $a=0$ означает, что наше уравнение не квадратное, а линейное. Но, если этого не учесть в алгоритме, то можно столкнуться с неверной его работой.

Следовательно, надо исправить приведенный алгоритм. Если $a=0$, то уравнение не квадратное, его решение будет $x=–c/b$. Но это справедливо только, когда $b\neq0$. В случае нулевого коэффициента $b=0$, необходимо проверять равенство нулю и свободного члена c, потому что в зависимости от этого у уравнения будет либо бесконечное число корней, либо корней вообще не будет. Поэтому схема алгоритма несколько усложнится (рис. 2.9).

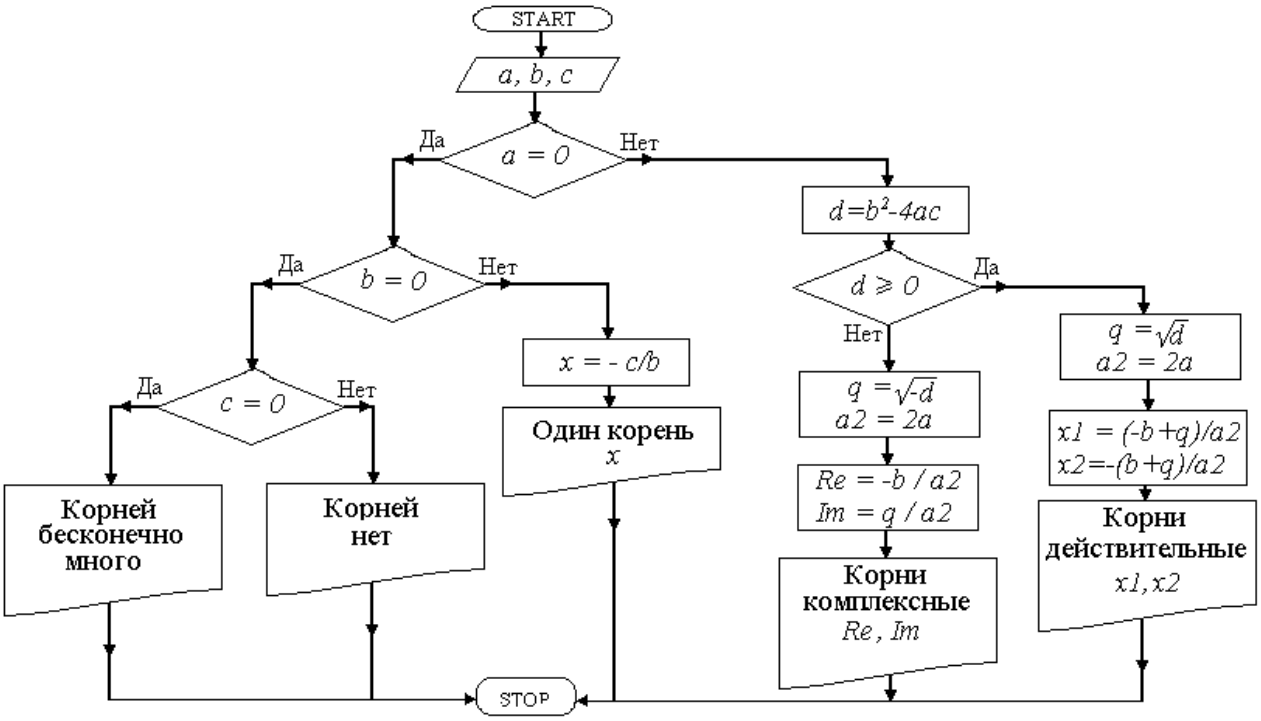


Рисунок 2.9

В случае отрицательности дискриминанта *d* алгоритм выдает сообщение о том, что корни уравнения комплексные, и выводит на печать действительную *Re* и мнимую *Im* части этих корней.

2.2.3 Алгоритм циклической структуры

Алгоритм циклической структуры — это алгоритм, в котором предусмотрено неоднократное выполнение одной и той же последовательности действий.

Эту последовательность действий называют телом цикла, а весь алгоритм циклом.

Различают два вида циклических алгоритмов:

1. Алгоритмы, в которых заранее известно число последовательных повторений тела цикла.

2. Алгоритмы, в которых число повторений заранее неизвестно, а должно быть определено в процессе вычисления.

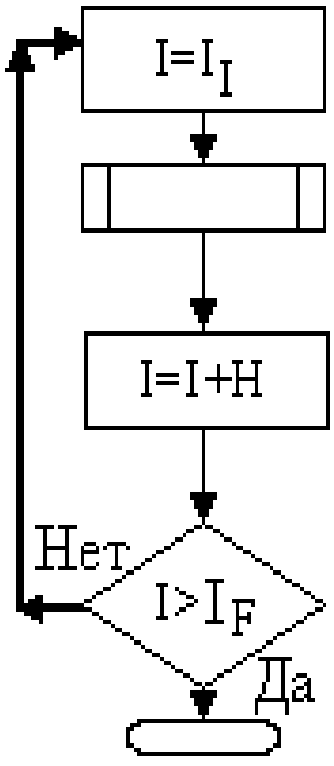
Для того, чтобы реализовать алгоритм первого типа, очевидно, необходимо подсчитать сколько раз повторятся вычисления. Для такого подсчета вводят в цикл специальный *параметр цикла* — переменную, которая изменяется при

повторных вычислениях. Значение этой переменной сравнивается с заданным числом повторений, на основании сравнения, делается заключение о необходимости продолжать или закончить циклический процесс.

Таким образом, для организации цикла с фиксированным числом повторений необходимо предусмотреть следующее:

1. Задать начальное значение параметру цикла.
2. Перед каждым повторением цикла изменять значение параметра цикла.
3. Проверять условие выполнения окончания повторений, сравнивая значение параметра цикла с заданным числом повторений.
4. Осуществлять переход к началу цикла, если условие повторения выполняется, и выход из цикла, если условие повторения не выполняется.

Эти требования на языке схем алгоритмов могут быть записаны следующем образом (рис. 2.10).



Присваивание параметру I начального значения I_I

Тело цикла (последовательность действий, для выполнения которой организуется цикл)

Изменение значения параметра цикла на некоторую величину H

Проверка условия окончания цикла (цикл заканчивается если $I > I_F$)

Выход из цикла

Рисунок 2.10

Циклические алгоритмы применяются для организации огромного числа вычислительных процессов. Поэтому для описания действий, которые надо

произвести при циклических вычислениях, в схемах алгоритмов используется специальный значок —  .

Внутри этого значка записывают параметр цикла *I*, а после знака равенства: начальное значение параметра цикла I_I, конечное значение параметра цикла I_F, шаг изменения параметра цикла *H* при следующем его прохождении (рисунок 2.11).

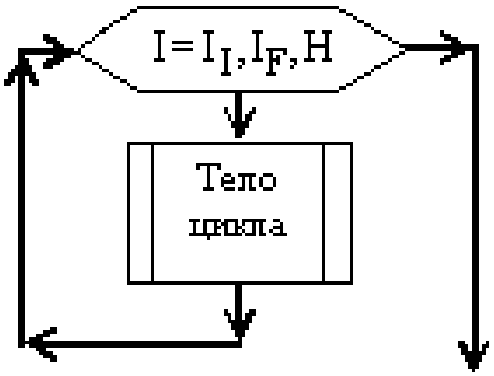


Рисунок 2.11

Пример 3. Найти сумму первых *N* чисел натурального ряда *S*=1+2+3+4+…+*N*.

Для решения задачи «в лоб» необходимо:

1. Положить S = 0, I = 1(параметр цикла).
2. Вычислить первое число (I = 1).
3. S=S+I.
4. Проверить, не пора ли кончить вычисления I > N.
5. Если вычисления надо продолжать, то прибавить второе число I=I+1=2.
6. S=S+I.
7. Проверить, не пора ли кончить вычисление I > N .
8. и т.д. (рисунок 2.12)

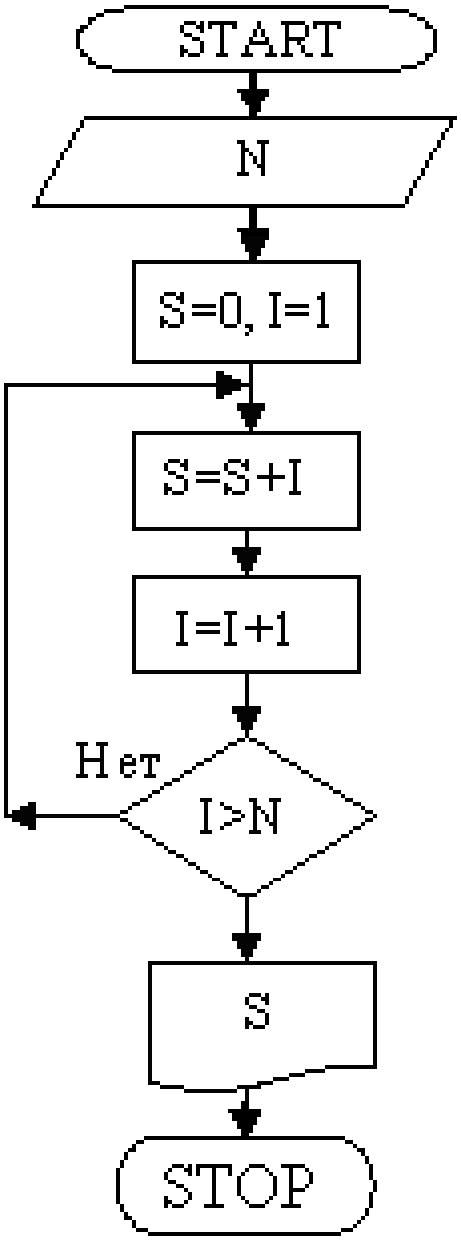


Рисунок 2.12

Тот же алгоритм с использованием стандартного значка цикла с заданным числом повторений можно записать более компактно. При этом, **если шаг *H* приращения параметра цикла равен 1**, то его можно **не указывать внутри значка цикла**, как это показано на схеме алгоритма (рис. 2.13).

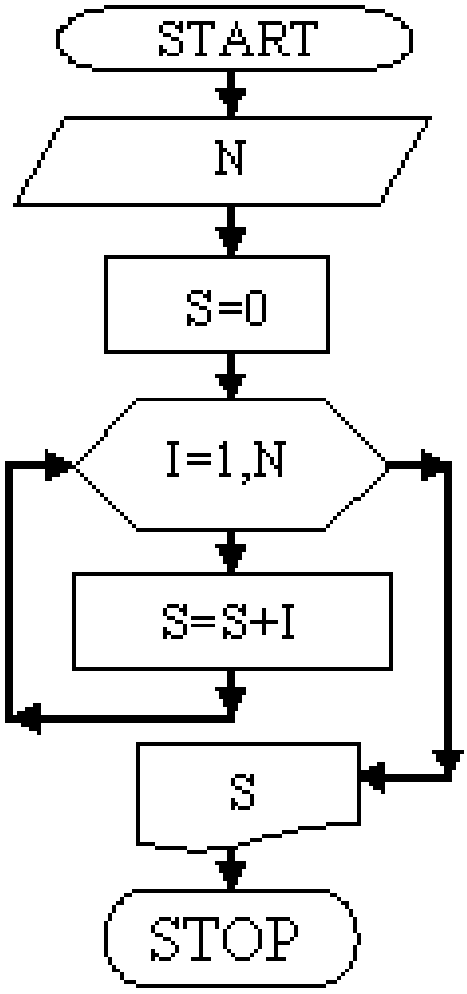


Рисунок 2.13

Заметим, что данный пример приведен только с целью демонстрации работы циклических алгоритмов. На самом деле вычислять сумму натуральных чисел, пользуясь им, крайне нерационально при большом N. Следует запомнить, что **эффективность алгоритмов проявляется, именно, при больших размерах входных данных у решаемых задач**. Поэтому гораздо проще подсчитать сумму первых N натуральных чисел как сумму членов арифметической прогрессии: $S = N\cdot(N+1)/2$. В этом случае вообще не надо организовывать никаких циклов!

Однако если усложнить поставленную задачу, то без циклов уже не обойтись. Найти сумму квадратов первых N чисел натурального ряда $S=1^2+2^2+3^2+4^2+\ldots+N^2$. В этом случае решение будет аналогичным уже рассмотренному выше, и структурная схема алгоритма практически не изменится

(рис. 2.13). Отличие будет только в теле цикла, где вместо оператора $S = S + I$ будет стоять оператор $S = S + I \times I$.

Возникает вопрос, можно ли сделать и этот алгоритм более быстрым с вычислительной точки зрения? Безусловно, если воспользоваться известным из математики соотношением:

$$S = \sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6},$$

которое легко доказывается по индукции. Это самый короткий и правильный способ. Но предположим, что надо вычислить следующую сумму $S_1 = \sum_{i=1}^{N} a_i \cdot i^2$, где коэффициенты a_i — заданные константы, для которой нет известных формул. Поэтому придется организовывать циклический алгоритм. (Попробуйте сделать это самостоятельно в качестве упражнения).

Для расчета искомой суммы квадратов S необходимо сделать N умножений. Умножение является *медленной* операцией, поэтому ее желательно заменить более быстрыми операциями. Для относительно небольших чисел N (меньших 2^{30}) можно предложить следующий алгоритм.

Допустим, нам известно число i^2 при некотором i-м прохождении цикла. Тогда при переходе к следующему значению $(i+1)^2 = i^2 + 2 \times i + 1$ нет необходимости вычислять квадрат i заново, а можно воспользоваться уже имеющимся в памяти значением.

Здесь начинающий программист резонно возразит — при таком способе вычисления $(i+1)^2$ никакого упрощения не будет, потому что всё равно используется одно умножение $2 \times i$, более того, добавляются еще два сложения!

Однако вся соль данной операции заключается в том, что умножение произвольных целых чисел $i \times i$ заменяется умножением i на 2. Поскольку ПК работает в двоичной системе, то в ней умножение (и деление) **целого** числа на числа, кратные степени двойки, можно реализовать просто сдвигом данного целого числа на соответствующее число разрядов влево (при делении – вправо).

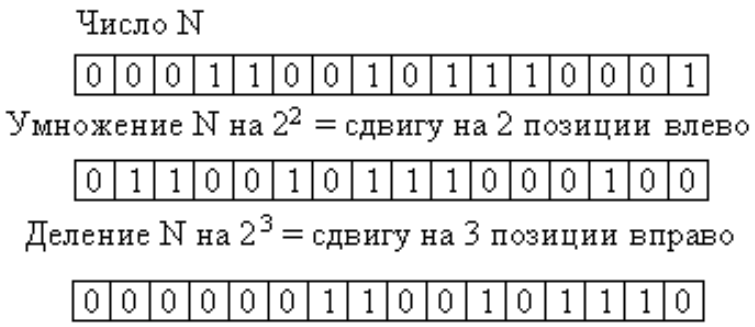


Рисунок 2.14

Такие операции сдвига выполняются в ПК практически мгновенно (гораздо быстрее, чем самые быстрые операции сложения или вычитания). Поясним это на примере. Допустим, имеется некоторое целое число N, которое в двоичной системе записывается в виде: $N_{(2)} = 1100101110001$. Тогда его умножение на число $k = 2^p$ и деление на число $m = 2^q$ будут реализованы с помощью сдвигов, как показано на рис. 2.14, где $p = 2$, а $q = 3$. При сдвиге на соответствующее число позиций вправо или влево образующиеся пустые ячейки заполняются нулями. В результате получаются числа:

$$N \times 2^2 = 110010111000100 \text{ и } N/2^3 = 1100101110.$$

Заметим, что при делении в ПК целых нечетных чисел на 2^q результат будет целым, а не дробным числом! В нашем примере исходное число N было нечетным, т.к. в последнем младшем разряде у него стоит 1. Но при делении N на $2^3 = 8$ результат деления стал целым и четным (!).

Операция сдвига в языке С обозначается N << k (сдвиг числа N на k позиций влево, эквивалентно его умножению на 2^k) или N >> p (сдвиг числа N на p позиций вправо, эквивалентно его делению на 2^p).

Таким образом, операция умножения двух целых чисел заменяется при каждом прохождении цикла на две операции сложения плюс одну операцию сдвига, что позволяет экономить вычислительное время!

На рис. 2.15 показана структурная схема этого алгоритма подсчета суммы квадратов первых N натуральных чисел. В алгоритме используется дополни-

тельная ячейка *q*, в которой хранится квадрат числа i^2 с предыдущего прохождения цикла. Но при реализации алгоритма на языке С необходимо описать переменные *S* и *q* типом *long int*!

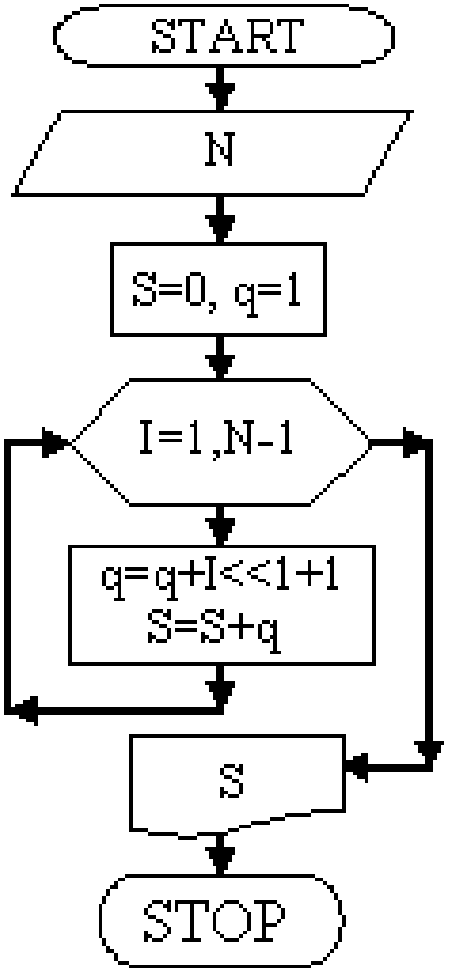


Рисунок 2.15

Теперь рассмотрим циклические алгоритмы, число повторений которых заранее неизвестно и должно быть определено в процессе выполнения данного цикла. Отличительной особенностью подобных алгоритмов является то, что внутри тела цикла должно быть предусмотрено условие его окончания и выхода из цикла, иначе повторения цикла будут происходить сколь угодно долго. Как говорят программисты — произойдет *«зацикливание»* алгоритма.

В языке С подобные алгоритмы реализуются с помощью специальных конструкций *while* и *do … while*.

Рассмотрим конкретные практические примеры использования циклического алгоритма с заранее неизвестным числом его повторений.

Пример 4. Заданы натуральные числа m и n. Надо найти их наибольший общий делитель (НОД) и наименьшее общее кратное (НОК).

Для нахождения НОД известен алгоритм, носящий имя древнегреческого математика и философа Евклида (III в до н.э.). Идея алгоритма заключается в следующем. Введем обозначения: $\mathrm{GCD}(m,n)$ — наибольший общий делитель чисел m и n; $\min(m,n)$ — минимальное из этих чисел. Тогда при его поиске требуется следовать следующему правилу (алгоритм Евклида):

$$\mathrm{GCD}(m,n) = \mathrm{GCD}(\min(m,n), |m - n|).$$

В результате неоднократного применения этого правила к паре чисел большее из них уменьшается. Этот итерационный процесс завершается, когда числа оказываются равными друг другу. Здесь количество повторений заранее неизвестно, поэтому в алгоритме надо использовать цикл с предусловием.

m = 168 n = 105 x = u =168 y = v = 105	нет	нет	нет	нет	да
	нет	да	нет	да	
	x = 63 u =273	y = 42 v =378	x = 21 u =651	y = 21 v =1029	
					u =840 x = 21; u =840

Э.В. Дейкстра предложил расширенный алгоритм, основанный на алгоритме Евклида и одновременно с НОД определяющий НОК чисел m и n. Не будем вдаваться в математические тонкости этого алгоритма, потому что они требуют знания теории чисел. Блок-схема алгоритма показана на рис. 2.16. После его работы переменной x присваивается НОД чисел $x = \mathrm{GCD}(m,n)$, а переменной u — НОК.

Справа от самого алгоритма показан пример его тестирования, который позволяет продемонстрировать технологию тестирования алгоритмов.

Легко убедиться разложением чисел $m = 168$ и $n = 105$ на простые сомножители, что их НОД равен 21, а НОК — 840. То есть для данного примера алгоритм работает правильно. Как видно из приведенных выше примеров тестирования алгоритма «вручную», это относительно несложно сделать только для простых задач, где число задействованных переменных не очень велико. Для более сложных задач тестировать зачастую проще уже написанные программы, а не алгоритмы, которые они реализуют.

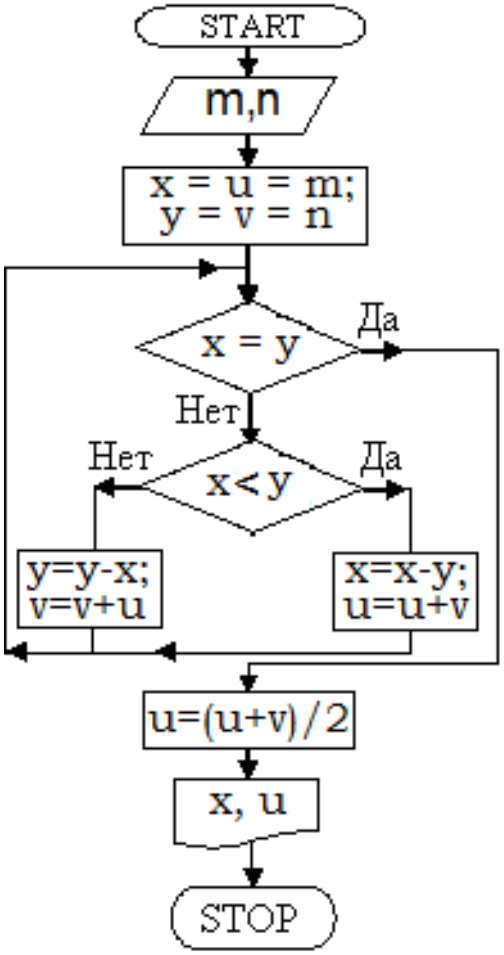


Рисунок 2.16

Для тестирования вручную следует оставить достаточно свободного места справа от блок-схемы. Контрольный пример не должен быть слишком сложным — это затрудняет тестирование, но и не быть тривиальным — это может привести к случайному совпадению с правильным решением. В первом столбце таблицы справа от блок-схемы, записываются переменные или условия, значения которых могут изменяться. Начиная со второго столбца, «сверху–вниз» записываются результаты выполнения алгоритма. Начало нового цикла соответствует добавлению нового столбца таблицы. Разработчик алгоритма должен выполнять команды формально, строго придерживаясь предписаний в блок-схеме.

Далее приведен изящный код на C, реализующий этот алгоритм, который приписывают самому Э.В. Дейкстре.

Код программы для алгоритма с рис. 2.16.

```
unsigned m,n,x,y,u,v;
// Здесь должна быть функция ввода m и n
```

```
x=u=m;
y=v=n;
while (x!=y)
 if (x<y) {y-=x; v+=u;}
  else {x-=y; u+=v;}
u = (u + v)>>1;
// x – НОД чисел m и n, u – НОК m и n
```

Рассмотрим другой интересный *пример 5*. Пусть требуется решить уравнение вида $f(x) = 0$, где функция f известна при этом она непрерывная и монотонная, но данное уравнение не имеет аналитического решения (например, $f(x) = e^x+\mathrm{tg}x$). Поэтому подобные уравнения решают численными методами. Для этого сначала «локализуют» искомый корень уравнения. Это самостоятельная задача, которая может быть гораздо более сложной, чем последующий поиск локализованного корня, но она решается для каждой функции $f(x)$ своими методами, поэтому здесь не изучается. Под локализацией корня понимают операцию нахождения такого отрезка $[a,b]$ на оси x, что функция на концах этого отрезка имеет разные знаки (например, $f(a) < 0$, а $f(b) > 0$). Тогда интуитивно понятно, что непрерывная монотонная функция обязательно пересечет ось абсцисс где-то внутри отрезка $[a,b]$ (это можно и строго доказать!) Описанная ситуация иллюстрируется нижеприведенным рис. 2.17. Можно рассмотреть и обратную ситуацию, когда функция монотонно убывает, т.е. $f(a) > 0$, а $f(b) < 0$, но предлагаем сделать это читателю самостоятельно. Для решения поставленной задачи будем использовать **метод деления отрезка** $[a,b]$ **пополам**.

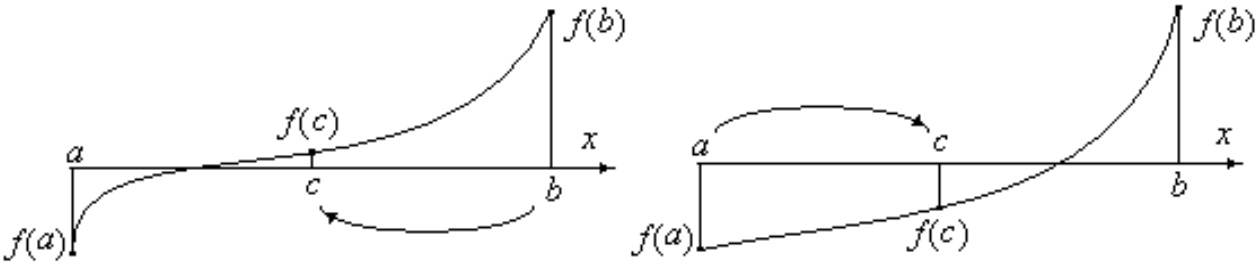


Рисунок 2.17

Идея метода иллюстрируется тем же рисунком.

1. Находится середина отрезка $[a,b]$, точка $c = (a + b)/2$.
2. Вычисляется значение функции в этой точке $f(c)$.
3. Если $f(c) > 0$ (это означает, что монотонная функция f пересекает ось абсцисс слева от точки c, как показано на левом рис. 2.17), то $b=c$, т.е. можно в два раза сократить исходный отрезок, перенеся точку b в точку c.
4. В противном случае $f(c) < 0$ (это означает, что монотонная функция f пересекает ось абсцисс справа от точки c, как показано на правом рис. 2.17), тогда $a=c$, и отрезок $[a,b]$ также сокращается вдвое.
5. Вновь выполняются пункты 1÷4, пока $f(c)$ не станет равным нулю.

Здесь необходимо оговорить один важный момент. Возникает естественный вопрос — когда $f(c)$ достигнет нуля? Дело в том, ПК считает $f(c)$ равной нулю, только когда $|f(c)| < 10^{-78}$. Но это очень малое число, и работа алгоритма при реализации его на компьютере может затянуться на очень долгое время. Поэтому при численном решении уравнений на ПК задаются необходимой точностью вычисления корня, которая характеризуется числом ε, выбираемым достаточно малым, исходя из требуемой точности решения задачи. Обычно $\varepsilon = 10^{-15} \div 10^{-7}$. Тогда выполнение неравенства $|f(c)| < \varepsilon$ означает, что корень уравнения найден с *заданной точностью*.

Эти рассуждения позволяют предложить блок-схему алгоритма решения задачи (рис. 2.18), но еще раз оговоримся: данный алгоритм справедлив только для монотонных **неубывающих** функций! В этом алгоритме проверка условия $|f(c)| < \varepsilon$, где в качестве ε используется параметр *eps*, является условием прекращения выполнения цикла.

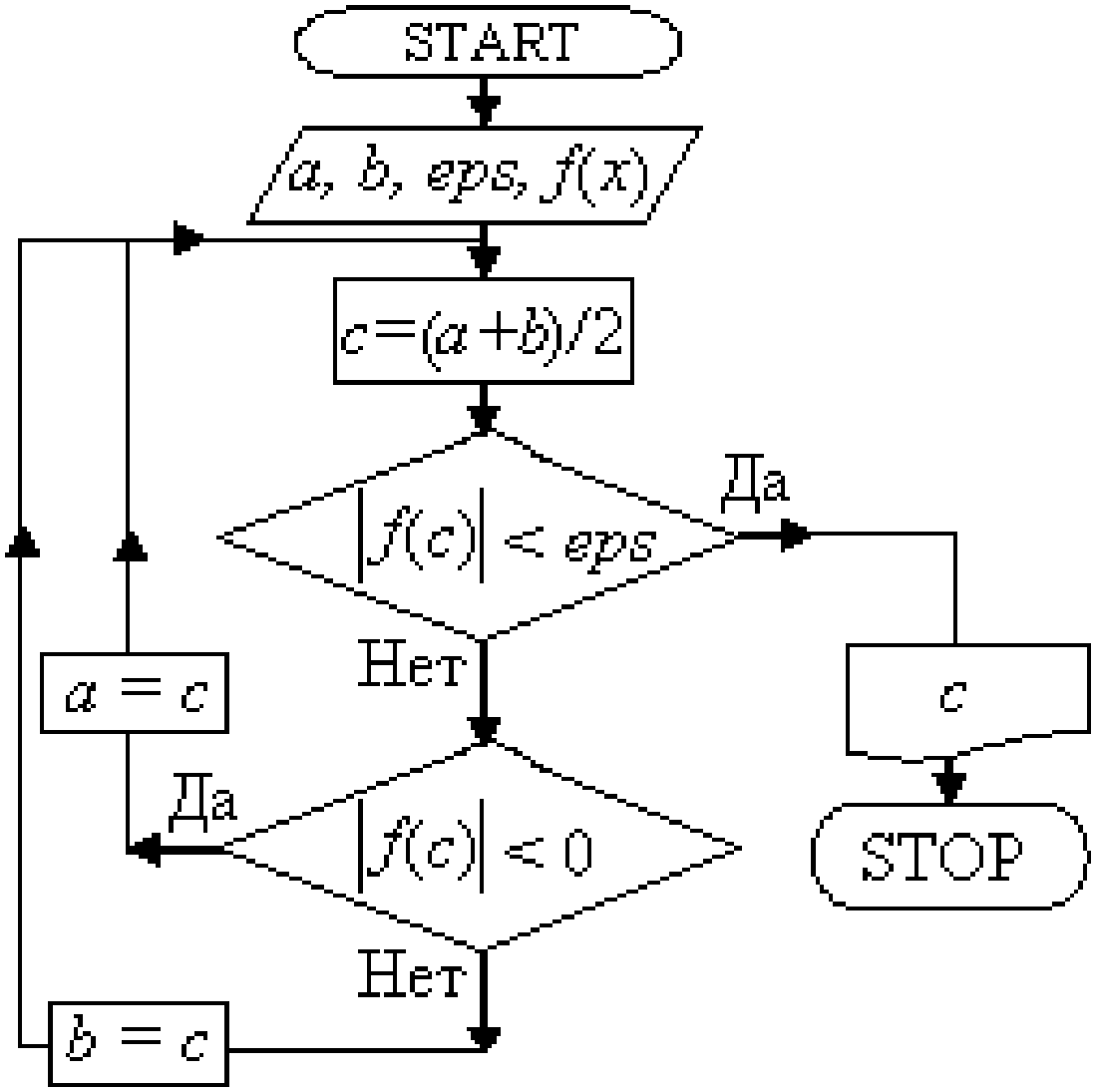


Рисунок 2.18

2.2.4 Вложенные циклы

Внутри одной циклической структуры можно использовать сколь угодно много других циклических алгоритмов. Такие алгоритмы называются у программистов «*вложенными циклами*». При этом в языке С внутри одного цикла любого типа разрешено использовать любые другие типы циклов. Для примера рассмотрим алгоритм подсчета числа счастливых билетов в мотке билетов, используемом в общественном транспорте.

Пример 6. Как известно, все билеты имеют шестизначные номера от 000000 до 999999. «Счастливыми» будем считать билеты, сумма первых трех цифр которых равна сумме последних трех цифр (есть и другие определения;

например, билет, у которого сумма цифр, стоящих на нечетных позициях, равна сумме цифр, стоящих на четных позициях, называется «счастливым по-московски»). Требуется подсчитать число счастливых билетов в полном мотке, содержащем миллион билетов.

Самый простой алгоритм решения этой задачи «в лоб» (хороший для иллюстрации вложенных циклов, но совершенно нерациональный!) заключается в следующем (Рис. 2.19). Введем целую переменную *counter*, в которой будем хранить число найденных счастливых билетов. Создадим шесть вложенных циклов, в которых параметры циклов будут изменяться от 0 до 9. На каждом шаге будем сравнивать сумму значений параметров внешних трех циклов с суммой параметров внутренних трех циклов. Если эти суммы окажутся равными (счастливый билет), то переменная *counter* увеличивается на единицу. После завершения работы вложенных циклов переменная *counter* будет содержать общее число счастливых билетов. На рис. 2.16 показана блок-схема данного алгоритма, где обозначены переменные $I1,\ldots, I6$ — параметры соответствующих вложенных циклов.

Вложенные циклы работают следующим образом. Сначала все параметры циклов $I1=I2=\ldots=I6 = 0$. Выполняется тело самого внутреннего цикла. Поскольку в этом случае $I1+I2+I3 = I4+I5+I6 = 0$, то значение счетчика увеличивается на 1. После этого параметр самого внутреннего цикла $I6$ увеличивается на единицу $I6 = 1$. Тело цикла выполняется снова. Далее $I6 = 2$, и т.д. При этом все остальные параметры циклов $I1,\ldots, I5$ сохраняют свои нулевые значения. Так продолжается пока параметр цикла $I6$ не станет равным 10. На этом вложенный внутренний цикл заканчивает свою работу, и управление передается циклу с параметром $I5$. Этот параметр увеличивается на единицу, и весь вложенный цикл с параметром $I6$ повторяется полностью для значения $I5 = 1$. Когда он отработает ($I6$ пробежит все значения от 0 до 9), то $I5$ увеличится на единицу. Процесс будет продолжаться, пока параметр цикла $I5$ не станет равным 10. Тогда управление передается циклу с параметром $I4$. Параметр $I4$ увеличивается на единицу, и

вложенные в него циклы полностью отрабатываются (параметры *I*5 и *I*6 пробегают все значения от 0 до 9). И так далее.

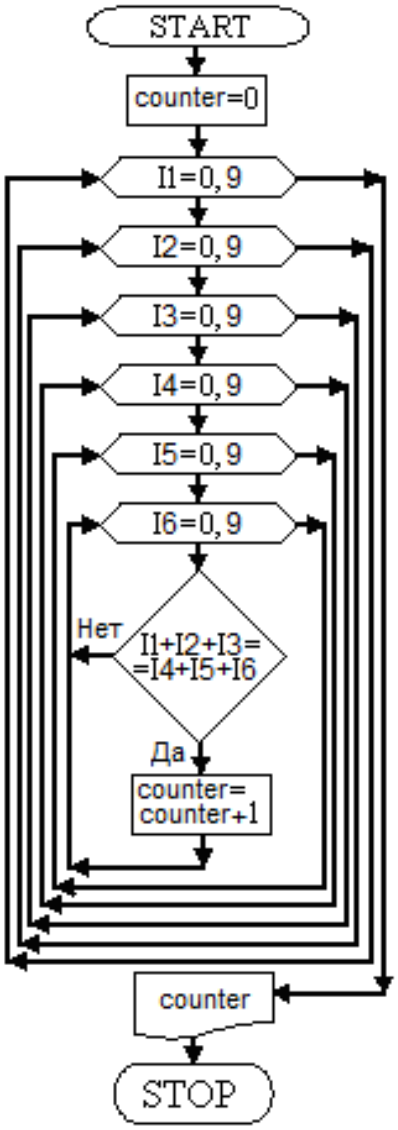


Рисунок 2.19

Описываемый процесс повторяется до тех пор, пока все шесть параметров циклов не пробегут все значения от 0 до 9. Несложно подсчитать, что в результате будет выполнено 10^6 операций сравнения и $4 \cdot 10^6$ операций сложения целых чисел (как правило, время, затрачиваемое на операцию сравнения, существенно превосходит время, затрачиваемое на быструю операцию сложения, особенно, когда сравниваются нецелые числа). Еще раз заметим, что предлагаемый алгоритм подсчета числа счастливых билетов очень нерациональный, хотя интуитивно понятный и простой. Здесь он приводится только с целью демонстрации

работы вложенных циклов. Ниже дан фрагмент кода программы на C, реализующий этот алгоритм.

```
long counter = 0;
for (int i1=0; i1<10; i1++)
  for (int i2=0; i2<10; i2++)
    for (int i3=0; i3<10; i3++)
      for (int i4=0; i4<10; i4++)
        for (int i5=0; i5<10; i5++)
          for (int i6=0; i6<10; i6++)
            if (i1+i2+i3 == i4+i5+i6) counter++;
```

После отработки этого фрагмента кода переменная *counter* будет содержать число 55252. Убедитесь в этом самостоятельно.

2.2.5 Два полезных алгоритма

Теперь попытаемся ответить на вопрос, как улучшить алгоритм расчета числа счастливых билетов? Для этого необходимо изучить начальные основы высшей математики, в частности, комбинаторики и математического анализа.

Рассматриваемая задача является занимательной, поэтому она была давно решена [1] в общем виде для произвольного четного числа знаков в номере билета и произвольной системы счисления, цифры которой используются для записи номеров. Были получены следующие формулы для расчета числа шестизначных счастливых билетов и десятичной системы счисления:

– наиболее удобная для аналитического подсчета:

$$N(6) = \sum_{k=0}^{2} (-1)^k C_6^k \cdot C_{32-10k}^5 = C_{32}^5 - C_6^1 \cdot C_{22}^5 + C_6^2 \cdot C_{12}^5; \qquad (2.1)$$

– удобная для написания эффективного алгоритма на ПК:

$$N(6) = 2\left[\frac{1}{4}\sum_{k=0}^{9}(2+3k+k^2)^2 + \sum_{k=10}^{13}(-107+27k-k^2)^2\right]. \qquad (2.2)$$

В выражении (2.1) введено стандартное обозначение C_n^k — число сочетаний из *n* по *k*. Оно обозначает, сколькими способами можно выбрать *k* элементов из имеющихся *n* элементов (например, сколькими способами можно выбрать трех кандидатов из десяти имеющихся — C_{10}^3). Из определения числа сочетаний понятно, что числа *k* и *n* могут быть только целыми и неотрицательными, более того, всегда $n \geq k$. Числа C_n^k еще называют биномиальными коэффициентами, поскольку они описывают разложение бинома Ньютона:

$$(a+b)^n = \sum_{k=0}^{n} C_n^k a^k b^{n-k},$$

из которого тривиально следует, что сумма биномиальных коэффициентов равна $\sum_{k=0}^{n} C_n^k = 2^n$. Для подсчета числа сочетаний в математике известна формула:

$$C_n^k = \frac{n!}{k!(n-k)!}, \qquad (2.3)$$

где *n*! (читается *эн-факториал*) равен произведению всех натуральных чисел от 1 до *n* (по определению 0! = 1). Из (2.3) тривиально следует свойство числа сочетаний $C_n^k = C_n^{n-k}$.

Видно, что выражение (2.1) достаточно сложно рассчитать вручную, даже с использованием калькулятора. Гораздо проще подсчитать его с помощью программы на ПК. Но для этого необходимо составить алгоритм вычисления числа сочетаний из *n* по *k*.

Начинающему программисту это может показаться очень легкой задачей, поскольку несложно воспользоваться формулой (2.3) «в лоб». Действительно, очень просто составить алгоритм и написать код программы вычисления факториала любого натурального числа. Фрагмент кода функции, рассчитывающей значение факториала заданного натурального числа, дан ниже.

```
long factorial(unsigned n)
{ if (n==0) return 1;
  else
```

```
  { long f=1;
    for (unsigned i=1; i<=n; f*=i++)}
  return f;
} // End of factorial
```

(Поскольку вход функции не может быть отрицательным, перед началом расчета факториала производится проверка равенства *n* нулю. В этом случае в соответствии с определением факториала функция выдает значение 1).

Теперь число сочетаний из *n* по *k* может быть подсчитано с помощью следующей функции.

```
long C(unsigned n, unsigned k)
{ if (n<k) {cout <<"Неверное использование функции n<k";
exit(1);}
  return factorial(n)/factorial(k)/factorial(n-k);
} // end C
```

Тогда провести вычисление числа счастливых билетов по формуле (2.1) можно с помощью следующего кода:

```
long N = C(32,5) – C(6,1)*C(22,5) + C(6,2)*C(12,5);.
```

Казалось бы, все очень просто, причем код получился не на много длиннее, чем его аналог с шестью вложенными циклами. Однако неприятность тут таится в следующем. Функция факториал является очень быстрорастущей. Например, число 32!, которое надо будет рассчитать при вычислении C_{32}^5, компьютер не сможет вычислить точно, поскольку оно превышает максимальное целое число типа *unsigned long*. Действительно, $32! \approx 2{,}6313083693369353 \times 10^{35}$, что намного больше $2^{32}–1$ = 4294967295 (максимальное число типа *unsigned long* в ПК), хотя число C_{32}^5 = 201376 относительно небольшое. Поэтому имеет

смысл создать эффективный алгоритм расчета числа сочетаний C_n^k, поскольку сочетания встречаются во многих других приложениях.

Алгоритм подсчета числа сочетаний из *n* по *k*. Проанализируем формулу (2.3). Из определения числа сочетаний следует, что *n* не может быть меньше любого из чисел *k* и *n*–*k*. Поэтому дробь (2.3) можно сократить на большее из чисел *k*! и (*n*–*k*)!

Если *k* > *n*–*k*, то после сокращения на *k*! в числителе дроби останется только произведение чисел *n*, *n*–1, *n*–2, …, *k*+1 (число сомножителей в числителе будет *n*–*k*), а в знаменателе остается число (*n*–*k*)! (тоже *n*–*k* сомножителей). В противном случае *k* < *n*–*k*, тогда дробь сокращается на (*n*–*k*)!, и в числителе остаются сомножители *n*, *n*–1, *n*–2, …, *n*–*k*+1 (ровно *k* сомножителей), а в знаменателе остается число *k*! (тоже *k* сомножителей).

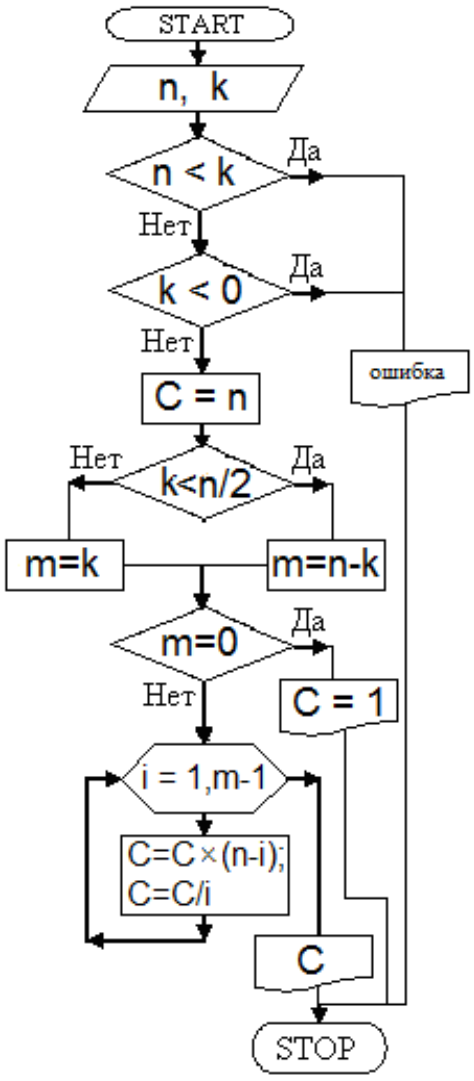


Рисунок 2.20

Эти соображения помогают предложить эффективный алгоритм расчета числа сочетаний C_n^k. Его блок-схема показана на рис. 2.20. Вначале производится проверка корректности входных данных. Если $n < k$ или $k < 0$, то выдается сообщение об ошибке. Далее алгоритм проверяет, какое из чисел k или $n–k$ больше и организует цикл, где параметр цикла пробегает все значения от 1 до наименьшего из них. Чтобы выходная переменная C не росла очень быстро, в теле цикла умножения на оставшиеся в числителе дроби (2.3) сомножители чередуются с делениями на сомножители из знаменателя той же дроби.

Возможный код, реализующий этот алгоритм, приведен ниже. Здесь входными параметрами ее являются переменные типа *int*, поэтому сначала проводится проверка условия $0 \le k \le n$.

```
#define Err1 {cout <<"Incorrect use of function. n < k"; exit(1);}
#define Err2 {cout <<"Incorrect use of function. k < 0"; exit(2);}
```

Это сообщения программисту о неправильном использовании функции.

```
unsigned long Comb (int n, int k)
// Функция подсчета числа сочетаний из n по k, которые
// должны удовлетворять условию 0 ≤ k ≤ n

{ unsigned m,i; //m=min(k,n-k)–вспомогательная пер-ная
  long double comb=n;
 // Проверка корректности использования функции
  if (n < k) Err1
  if (k < 0) Err2
// Определение минимального из чисел k и n-k
  if (k > (n>>1)) m = n - k;
   else m = k;
 // Подсчет числа сочетаний
  if (m == 0) return 1;
```

```
  else
  { for (i=1; i<m;)
    { comb* = n - i++;
      comb/ = i; }
  return comb; }
} //End of Comb
```

Приведенный код позволяет решить поставленную задачу о подсчете числа счастливых билетов, используя формулу (2.1), поскольку максимальное число, встречающееся в ней равно $C_{32}^{5} = 201376 < 4294967295$. Но, если нужно подсчитать, например, $C_{40}^{20} = 137846528820$, то потребуется использовать функцию Comb типа *double* или *long double*. Тогда в приведенном выше коде надо будет внести некоторые исправления (пожалуйста, сделайте их самостоятельно). Число счастливых билетов можно подсчитать с помощью кода, аналогичного уже описанному выше:

```
long N = Comb(32,5) – Comb(6,1)*Comb(22,5) + Comb(6,2)*Comb(12,5);
```

После его реализации на персональном компьютере получается то же число счастливых билетов 55252 среди миллиона билетов в мотке.

Теперь попробуем воспользоваться формулой (2.2). Непосредственные расчеты при ее использовании не представляют никаких трудностей. Зададимся вопросом: можно ли упростить вычисления по этой формуле? Из анализа (2.2) видно, что потребуется несколько раз подсчитать значения многочлена второй степени для различных k и найти квадраты полученных значений. Предлагаемая ниже методика расчета значения квадратного многочлена не даст большого выигрыша по скорости счета. Но существует большое число практических задач, где необходимо вычислять значение многочлена произвольной степени r в

конкретной точке x, когда известны все его коэффициенты a_0, a_1, …, a_r. Подобные многочлены в математике называются *полиномами* степени r:

$$P_r(x) = \sum_{i=0}^{r} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_r x^r. \qquad (2.4)$$

Поэтому попробуем решить следующую задачу. Составить алгоритм расчета значения полинома (2.4) по заданным степени полинома r, его коэффициентам a_0, a_1, …, a_r и величине x.

Сначала воспользуемся прямым алгоритмом, не учитывающим особенности машинной арифметики. Алгоритм решения этой задачи «в лоб» показан на рисунке 2.21.

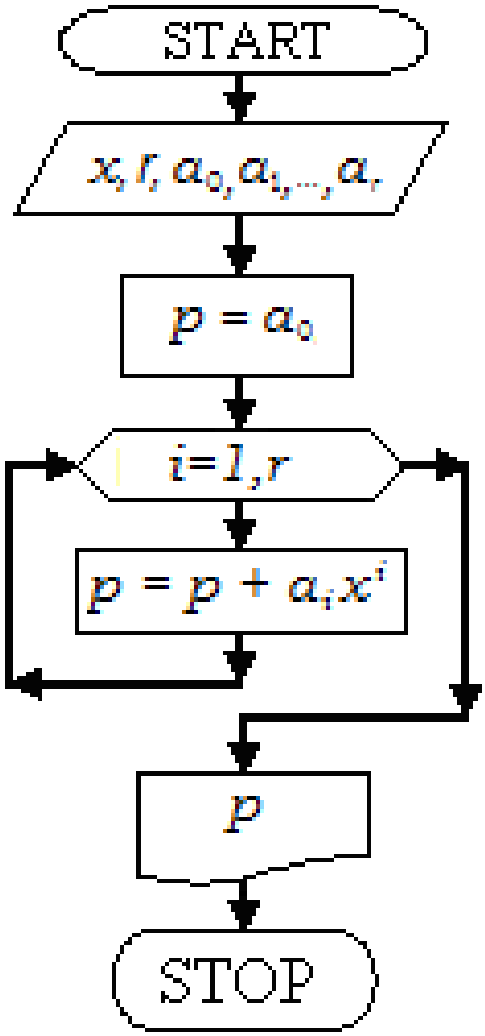


Рисунок 2.21

Он не представляет никаких сложностей: создается цикл, в теле которого к результату прибавляется очередное слагаемое. В языке С нет операции возведения в степень, но в библиотеках всех компиляторов есть функция

```
double pow(double x, double y),
```

которая выдает результат возведения *x* в степень *y*: $z = x^y$.

Ниже приведен код программы, реализующей алгоритм с рисунка 2.21. Входные параметры – коэффициенты полинома передаются в функцию в виде массива Arr размера *r*+1. Каждый элемент массива содержит соответствующий коэффициент. Более подробному описанию массивов данных посвящен следующий раздел. Здесь нам достаточно знать, что с элементами массива можно работать так же, как и с обыкновенными переменными.

```
double Polynom(double Arr[], unsigned r, double x)
// Функция расчета значения полинома в точке x по заданным коэффициентам
// и степени полинома; pow(a,b) – библиотечная функция, возводящая a в степень b
 { double p=Arr[0];
   unsigned i;
// Arr – массив коэффициентов полинома a0, a1,…, ar; r – степень полинома;
// x – значение переменной, для которого рассчитывается Pr(x);
// p – вспомогательная переменная
  for (i=1; i<=r; i++) p+=Arr[i]*pow(x,i);
  return p;
 }//End Polynom
```

Проведем анализ предложенного алгоритма. Он потребовал *r* сложений, *r* умножений и *r* операций возведения в степень. Читателю следует знать, что операция возведения в степень — одна из самых медленных в ЭВМ, поэтому

приведенный алгоритм можно признать неудачным с точки зрения временной сложности. Попробуем составить более быстрый алгоритм.

Алгоритм расчета значения полинома по схеме Горнера. Здесь на помощь приходит следующая схема. Представим формулу (2.4) в виде:

$$P_{\text{n}}(x) = \sum_{i=0}^{r} a_i x^i = (\ldots(a_r \cdot x + a_{r-1}) \cdot x + \ldots + a_1) \cdot x + a_0 \qquad (2.5)$$

Выражение (2.5) не очень удобно для восприятия с точки зрения математика, но для программиста оно более подходящее, потому что из него следует, что вычисление значения полинома не требует операций возведения в степень. Составим алгоритм вычисления полинома, который получил название **«схема Горнера»** (*Horner's rule*) по имени своего создателя, английского математика У.Д. Горнера (1786–1837). Его блок-схема показана на рисунке 2.22. Сравнивая первый и второй алгоритмы, можно заметить, что второй требует тоже r сложений и r умножений, но операции возведения в степень в нем отсутствуют. Поэтому по временной сложности алгоритм Горнера предпочтительнее предыдущего алгоритма.

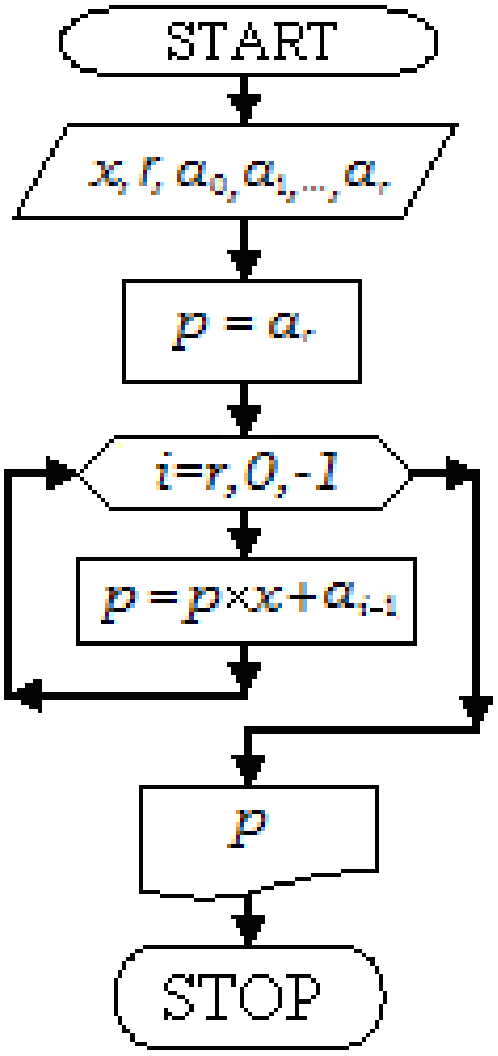


Рисунок 2.22. Алгоритм Горнера

Пользуясь этим алгоритмом, несложно написать код реализующей его программы. В ней все обозначения остаются прежними, что и у функции Polynom, приведенной выше.

```
double Horner(double Arr[], unsigned r, double x)
// Функция расчета значения полинома методом Горнера в точке x
// по заданным коэффициентам и степени полинома
 { double p=Arr[r];
   unsigned i;
   for (i=r; i;) p=p*x+Arr[--i];
   return p;
 }// End Horner
```

Теперь можно вернуться к вычислению количества счастливых билетов по формуле (2.2). Обозначим в этом выражении $a_0 = 2$, $a_1 = 3$, $b_0 = 107$, $b_1 = -27$. Кроме того, потребуется ввести еще вспомогательные переменные S_0 и S_1, в которых будут накапливаться значения сумм, и переменную p. Блок-схема этого алгоритма показана на рисунке 2.23. В нем значения полиномов второй степени из выражения (2.2) рассчитываются по схеме Горнера (при этом экономится по одной операции умножения для каждого k).

Далее приводится код программы расчета числа счастливых билетов по формуле (2.2). Описанный алгоритм считается наиболее эффективным. Легко убедиться непосредственным подсчетом, что он требует минимального числа умножений и вообще не требует операций сравнения чисел.

```
// Код расчета числа счастливых билетов по формуле (2.2)
long S1 = 0, S2 = 0, p;
long a0 = 2, a1 = 3, b0 = 107, b1 = -27;
for (int k=0; k<10; k++)
{ p = (k+a1)*k + a0;
  p* = p;
  S1+ = p; }
S1>>2;
for (int k=10; k<14; k++)
{ p = (k+b1)*k + b0;
  p* = p;
  S2+ = p; }
N = (S1 + S2)<<1;
```

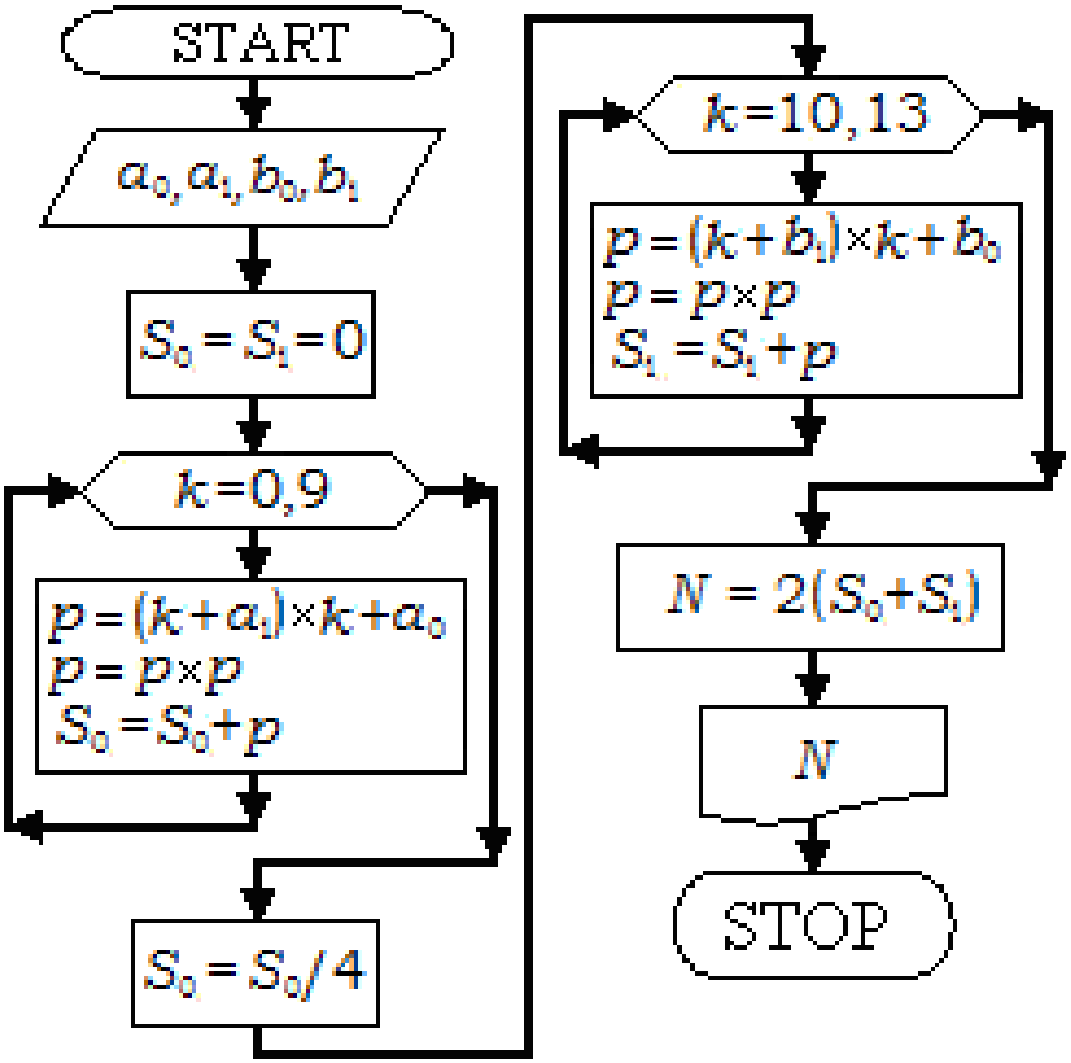


Рисунок 2.23. Эффективный алгоритм расчета числа счастливых билетов

Здесь стоит остановиться еще на одной тонкости программирования. В последнем алгоритме операция возведения числа в квадрат, присутствующая в формуле (2.2) под знаками сумм, заменена операцией умножения. Выше уже упоминалось, что возведение в степень z=pow(x,y) очень медленная операция, потому что компьютеру все равно, возводить число *x* в квадрат или, например, в степень 3,1415926535; эта операция производится следующим образом:

$$z = \exp(y \cdot \ln(x)).$$

При этом функции вычисления натурального логарифма и экспоненты от аргумента реализуются путем разложения их в ряд Тейлора с числом членов разложения, достаточным для достижения нужной точности (кстати, расчет значения

ряда разложения в конкретной точке производится по схеме Горнера!). Поэтому возведение в целую степень, когда показатель относительно мал $z_1=x^2$, $z_2=x^3$, $z_3=x^4$, проще вычислять с помощью соответственно кода z1=x*x; z2=x*x*x; b=x*x; z3=b*b; (возведение в четвертую степень производится за 2 умножения вместо трех!).

Подведем резюме. В подразделе описаны два эффективных алгоритма – расчета числа сочетаний из n по k и вычисления значения полинома в точке по схеме Горнера, часто встречающиеся во многих практических задачах.

Начинающему программисту может показаться, что описанные попытки упростить решение задачи, которая очень просто решается без каких-либо математических или алгоритмических «ухищрений», — совершенно нерациональная трата времени и сил. На самом деле, современный ПК моментально обработает код с шестью вложенными циклами, и решение задачи вычисления числа счастливых билетов таким способом будет «наипростейшим» с точки зрения затрат времени на разработку алгоритма решения и соответствующего программного кода (код с 6 вложенными циклами можно написать, вообще без предварительного составления блок-схемы алгоритма). Но предположим, что число цифр в номере билета будет 8 или 10. Тогда надо будет составить программы с восемью или десятью вложенными циклами соответственно. Компьютеру потребуется произвести 10^8 или 10^{10} сравнений чисел (сто миллионов и десять миллиардов соответственно!). Мы видим, что даже незначительное, на первый взгляд, увеличение размера входной информации приводит к значительному росту временной сложности программы. При этом расчет по формулам, аналогичным (2.1) или (2.2), практически не увеличит времени счета. Поэтому, если новичок хочет стать квалифицированным программистом, то ему придется решать сложные задачи большой размерности, где и проявляется выигрыш от использования эффективных алгоритмов. Начинать учиться разрабатывать такие алгоритмы надо даже при решении относительно простых задач.

3 ИНФОРМАЦИОННЫЕ МАССИВЫ ДАННЫХ

В предыдущем разделе при расчете значения полинома в точке уже рассматривался массив данных, в котором содержалась информация о коэффициентах полинома. Теперь рассмотрим этот вопрос подробнее.

3.1 Определение массива в языке С

Массив — упорядоченное множество однородных элементов, имеющих имя. Для того чтобы можно было пользоваться элементами массива в программе, необходимо выполнить следующие операции:

1) Описать массив, для чего надо указать в программе тип элементов массива (символьные, целые, с плавающей запятой и т.д.), задать имя массива (указатель на начало массива или точнее — адрес первого элемента массива) и, наконец, задать число элементов массива. В языке С массив должен быть описан до его использования в программе, а сама операция описания массива выглядит следующим образом:

```
int Array[NUM];
```

где служебное слово int задает тип массива и означает, что все элементы массива будут целыми числами (возможны и другие типы массивов, для которых используются свои служебные слова), и дает информацию для компилятора, сколько памяти выделить под каждый элемент массива; *Array* — имя массива, которое является указателем (адресом) на начало самого первого элемента массива в памяти компьютера; число в квадратных скобках после имени массива *NUM* должно быть обязательно **целой константой**, которая задает число элементов массива.

2) Присвоить начальные значения элементам объявленного массива.

Положение элементов в массиве в языке С упорядочивается по одному измерению. Позиция элементов в массиве задается посредством индекса, записанного в квадратных скобках после имени массива. В языке С все массивы начинаются с индекса с номером 0 (ноль), поэтому после объявления, приведенного выше, элементы массива будут иметь номера от 0 до *NUM*–1. Например, запись:

```
Array[5]=23; ,
```

сделанная после объявления массива *Array*, означает, что элементу массива с индексом 5 (расположенному шестым в массиве!) присваивается значение 23. Данная запись будет корректной, если целая константа *NUM*, используемая для задания числа элементов массива при его объявлении, была не меньше 6.

После объявления массива и присваивания начальных значений его элементам с ними можно работать так же, как с обычными переменными. Чтобы добраться до содержимого конкретного элемента массива, надо указать имя массива, а затем — номер его позиции в массиве, помня о том, что первый элемент массива имеет номер 0. В качестве индекса элемента массива может быть использовано любое выражение, имеющее целый тип (индекс должен быть всегда целым числом!), поэтому допустима, например, следующая запись:

```
int i=2, j=3, k;
k=Array[2*i+j];
```

Но надо обязательно следить за тем, что переменные, используемые для вычисления индекса, должны получить конкретное числовое значение до начала выполнения действий с элементами массива и, кроме этого, значение целого выражения, используемого в качестве индекса массива, должно быть обязательно **неотрицательным** и **не превосходить** значения *NUM*–1. Дело в том, что компилятор языка С не проверяет, вышло ли значение индекса за границы описанного массива. Положение элемента массива находится компилятором по следующей схеме:

1) Встретив в программе имя массива (в нашем примере *Array*) находится его адрес в памяти компьютера и определяется тип массива (задан при его объявлении). В примере, приведенном выше, массив *Array* имеет тип int, значит, каждый его элемент занимает в памяти ПК два байта.
2) Вычисляется значение индексного выражения. В примере: 2*i+j=7.
3) К адресу первого элемента массива *Array* добавляется значение полученного индекса, умноженное на размер одного элемента массива в байтах. Полученное целое число задает адрес искомого элемента. В описанном примере

размер одного элемента равен двум байтам, поэтому адрес элемента в массиве с индексом 7 (восьмого по очереди в массиве) будет Array+7*2.

4) Компилятор берет из памяти целое число, расположенное в следующих двух байтах после найденного адреса, и присваивает это значение переменной *k*. (Если массив имеет другой тип, то выбирается число байт в соответствии с размером одного элемента массива).

Теперь в случае, когда полученный индекс вышел за границы массива (в приведенном примере, например, число *NUM* = 6), всё равно компилятор отсчитает от адреса *Array* полученное число байт и возьмет следующие два байта по полученному адресу. Такая ситуация может привести к непредсказуемым последствиям в работе программы! Поэтому программист должен самостоятельно следить и предотвращать возможные выходы индексных выражений за границы массивов. Это иллюстрируется рисунком 3.1, где задан массив int Ar[8] целых чисел, индексы элементов которого имеют номера от 0 до 7.

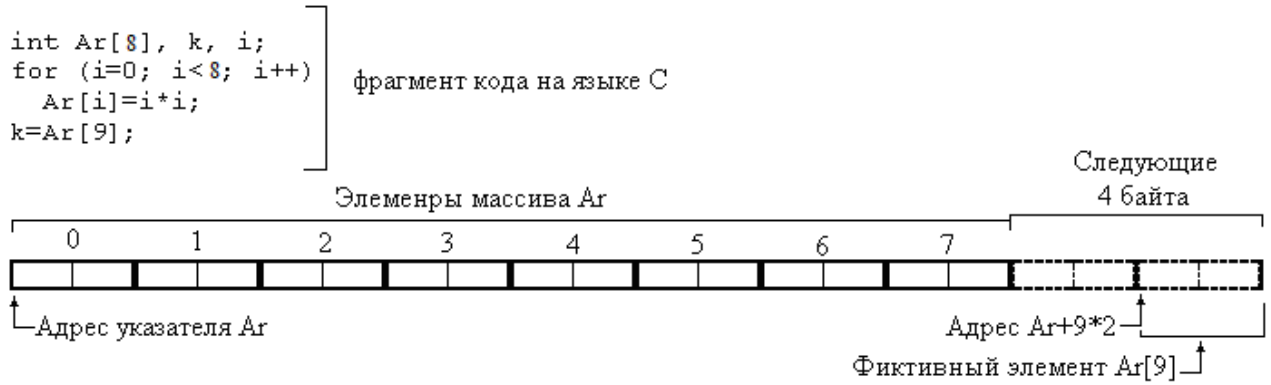


Рисунок 3.1

Данный массив инициализирован квадратами целых чисел от 0 до 7, т.е. элементам массива присвоены значения: 0, 1, 4, 9, 16, 25, 36, 49. После чего переменной *k* присваивается значение несуществующего элемента массива *Ar*[9]. Однако компилятор С вместо выдачи сообщения об ошибке (как это произошло бы при компиляции программы на языке PASCAL) находит элемент массива по адресу *Ar*+9×2 и формирует из содержимого следующих 2-х байтов некоторое целое число, которое и присваивается переменной *k*.

Необходимо помнить, что элементы информационного массива данных располагаются в памяти ПК по строкам. В языке С не определены многомерные массивы, однако элементами массива могут быть любые сколь угодно сложные объекты, кроме функций. В частности, элементом массива может быть другой массив, поэтому можно определить и многомерные массивы в виде массива, состоящего из других массивов размерности на единицу меньших.

3.2 Многомерные массивы в языке С

Многомерные массивы в языке С — это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например, при выполнении объявления двумерного массива int Array2[4][3] в памяти выделяется участок для хранения значения переменной *Array2*, которая является указателем на массив из четырех указателей Array2[0], Array2[1], Array2[2], Array2[3]. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов.

Такое выделение памяти схематически показано на рисунке 3.2. Таким образом, объявление Array2[4][3] порождает в программе три разных объекта: указатель с идентификатором Array2, безымянный массив из четырех указателей Array2[0], Array2[1], Array2[2], Array2[3]и безымянный массив из двенадцати чисел типа int.

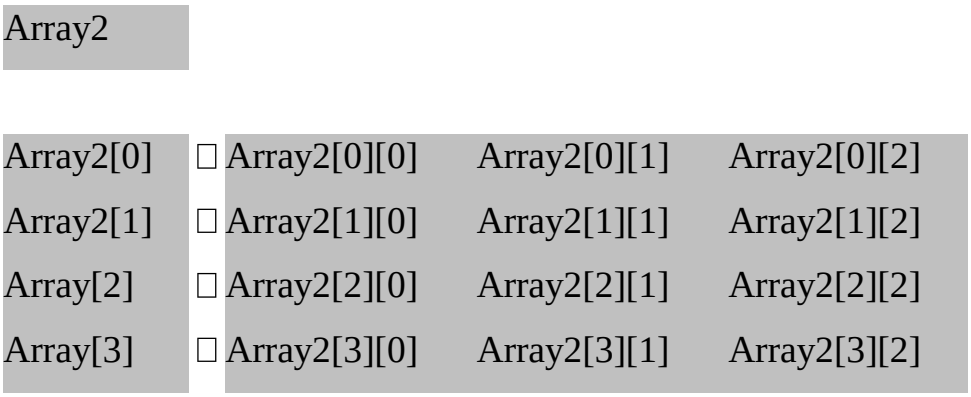


Рисунок 3.2. Распределение памяти для двумерного массива

Для доступа к безымянным массивам используются адресные выражения с указателем Array2. Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме Array2[2] или *(Array2+2). Для доступа к элементам двумерного массива чисел типа int должны быть использованы два индексных выражения в форме Array2[1][2] или эквивалентных ей *(*(Array2+1)+2) и (*(Array2+1))[2]. Следует учитывать, что с точки зрения синтаксиса языка C указатель Array2 и указатели Array2[0], Array2[1], Array2[2], Array2[3] являются константами, и их значения нельзя изменять во время выполнения программы.

Размещение трехмерного массива происходит аналогично, и объявление double Array3[3][4][5] порождает в программе, кроме самого трехмерного массива из шестидесяти чисел типа *double*, массив из четырех указателей на тип *double*, массив из трех указателей на массив указателей на *double*, и указатель на массив массивов указателей на *double*.

Как уже отмечалось, при размещении элементов многомерных массивов они располагаются в памяти подряд по строкам, т.е. быстрее всего изменяется **последний индекс**, а медленнее – **первый**. Такой порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение.

Рассмотрим примеры работы с многомерными массивами.

Пример 7. Составить алгоритм и программу нахождения минимального элемента в трехмерном массиве *Arr* размера $M \times N \times K$ и адреса его местонахождения (предполагается, что *M, N, K* — известные целые положительные константы, а элементы массива — заданные действительные числа).

Идея алгоритма поиска очень простая. Сначала будем считать минимальным самый первый элемент в массиве, индексы которого [0][0][0]. Далее надо просмотреть последовательно все элементы массива, сравнивая их с минимальным. Если какой-то текущий рассматриваемый элемент оказывается меньше минимального, то он становится новым минимальным, и его координаты запоминаются. После просмотра всех элементов массива будет найден самый минимальный. Алгоритм поиска показан на рисунке 3.3. Выходные параметры алгоритма: значение минимального элемента в массиве (*min*) и его координаты в массиве (*im, in, ik*).

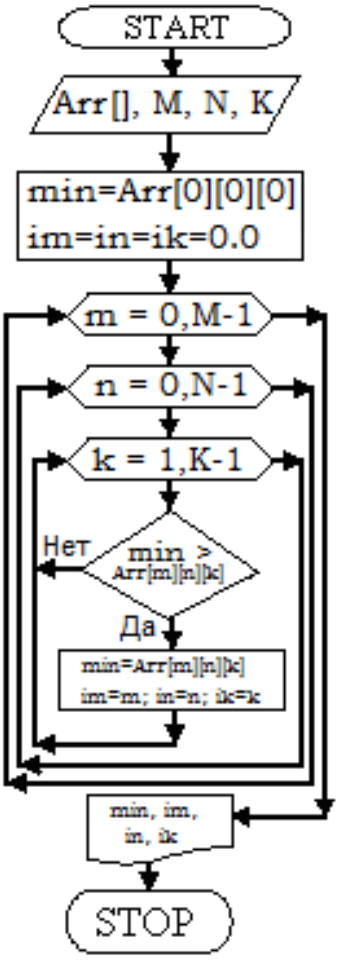


Рисунок 3.3

На основе этого алгоритма несложно составить код программы, реализующий его:

```
double Arr[M][N][K], min;
unsigned m, n, k, im, in, ik;
// Здесь должна быть функция инициализации массива Arr
min = Arr[0][0][0];
for (m=0; m<M; m++)
 for (n=0; n<N; n++)
  for (k=1; k<K; k++)
   if(min > Arr[m][n][k]))
   {
    min=Arr[m][n][k];
    im=m; in=n; ik=k;
   }
// Здесь должно быть функция вывода min, im, in, ik
```

Остановимся на одном неочевидном моменте выполнения этого кода. Приходится вычислять адрес элемента в массиве $M \times N \times K$ раз. Вычисление адреса элемента с номером $[i][j][r]$ в массиве *Arr* производится в соответствии с выражением: $Arr + (i \times N \times K + j \times K + r) \times 8$ (здесь число 8 определяет количество байт, отводимое в персональном компьютере переменной типа *double*). Поэтому поиск нужного элемента в массиве занимает определенное время, хотя современные компиляторы стараются максимально минимизировать его.

Попробуем написать код, который избавит компьютер от необходимости расчета адреса элемента массива на каждом шаге и сделает сам код несколько более изящным. Для этого в языке С существуют указатели на переменные. Если определить в программе указатель типа *double* на первый элемент массива *Arr*, например, следующим образом:

```
double Arr[], *ptr;
ptr = Arr;
```

то указатель *ptr* будет тоже указывать на первый элемент массива *Arr*. Тогда операция *ptr++ приведет к тому, что указатель будет указывать («перескочит») на следующий элемент массива *Arr*[0][0][1], а после выполнения операции *(ptr+5) он будет указывать на элемент, расположенный на 40 байт правее элемента *Arr*[0][0][1], и т.д. Поэтому можно предложить следующий альтернативный код, решающий ту же задачу.

```
double Arr[M][N][K], min, *ptr, *ptrmin;
unsigned m, n, k, im, in, ik, temp;
// Здесь должна быть функция инициализации массива Arr
ptr = Arr;
min = *ptr++;
temp = M*N*K;
for (m=1; m<temp; m++) // Цикл поиска минимального элемента
{ if (min > *ptr)
  { min = *ptr; ptrmin = ptr;}
  *ptr++;
}
// Определение координат минимального элемента в массиве
temp = (ptrmin – Arr)/sizeof(double);
im = temp/N/K;
in = temp/K – im*N;
ik = temp – im*N*K – in*K;
// Здесь должна быть функция вывода min, im, in, ik
```

Используя указатели, можно обойтись всего одним циклом и сократить время счета, поскольку отсутствует необходимость вычисления адреса текущего рассматриваемого элемента массива. В последних четырех строках программы вычисляются координаты найденного минимального элемента в массиве по адресу указателя на него (**ptrmin*). Переменной *temp* присваивается значение разности между адресами найденного минимального элемента и начала массива, которое будучи разделенным на число байт, занимаемое каждым элементом массива, показывает порядковый номер минимального элемента в массиве. Этот порядковый номер равен ($im \times N \times K + jm \times K + rm$). Откуда несложно вычислить параметры *im*, *jm* и *km* — координаты искомого элемента.

Пример 8. Рассмотрим пример посложнее. Требуется составить алгоритм перемножения двух прямоугольных матриц *MatM* (размера $M \times N$) и *MatN* (размера $N \times K$), а также написать код функции, реализующей этот алгоритм.

В теории алгоритмов известен алгоритм Штрассена перемножения матриц, который эффективнее предлагаемого ниже, но только для матриц большого размера $M, N, K > 60 \div 70$. Этот алгоритм рекурсивный, поэтому вернемся к нему позднее, когда будем рассматривать рекурсивные алгоритмы.

Как известно, можно перемножать только матрицы, у которых число столбцов первого сомножителя равно числу строк второго. В результате перемножения получается матрица *MatR* (размера $M \times K$).

Предлагаемый алгоритм следующий. Организуются два вложенных цикла, в которых параметры цикла изменяются от 0 до числа строк первого сомножителя и числа столбцов второго сомножителя соответственно. В теле цикла организуется еще один вложенный цикл, с помощью которого находится скалярное произведение выбранных строки первой матрицы и столбца второй. Результат вычисления скалярного произведения присваивается соответствующему элементу результирующей матрицы. Блок-схема алгоритма показана на рис. 3.4.

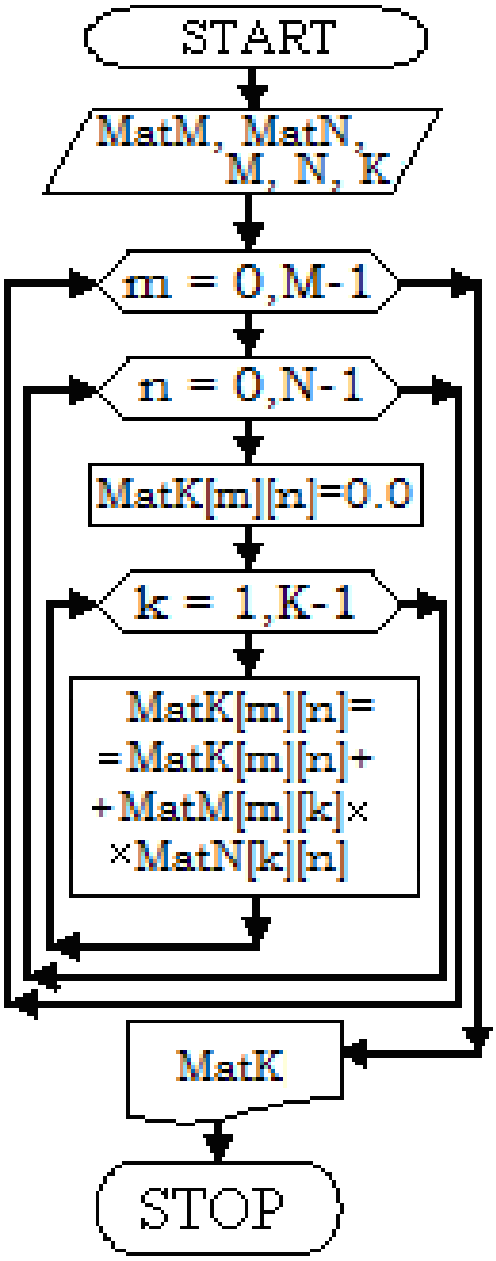


Рисунок 3.4

Код функции, реализующий этот алгоритм достаточно простой.

```
#define M // целая константа
#define N // целая константа
#define K // целая константа

double MatM[M][N], MatN[N][K], MatR[M][K];
// Здесь должна быть инициализация массивов MatM и MatN
void Matr_Mult (double MatM[], double MatN[], double MatR[])
```

```
{ for (int m=0; m<M; m++)
    for (int k=0; k<K; k++)
     { MatR[m][k] = 0.0;
       for (int n=0; n<N; n++)
       // Скалярное произведение строки и столбца
         MatR[m][k]+ = MatM[m][n]*MatN[n][k];
     }
}
```

Можно попробовать написать код перемножения матриц с использованием указателей на двумерные массивы, задающие их. Это избавит от необходимости нахождения адресов элементов, участвующих в расчете скалярного произведения. Хотя сам код получается длиннее и интуитивно не таким понятным, как приведенный выше, но он будет работать быстрее.

```
#define M //целое число
#define N //целое число
#define K //целое число

double MatM[M][N], MatN[N][K], MatR[M][K];
// Здесь должна быть функция инициализации массивов MatM и MatN
void Matr_Multipl (double MatM[], double MatN[], double MatR[])
// Здесь должна быть функция инициализации массивов MatM и MatN
void Matr_Multiplication (double MatM[], double MatN[], double MatR[])
{ double *ptrM, *ptrN, *ptrR; // указатели на элементы матриц
  ptrN = MatN; ptrR = MatR;
  for (int m=0; m<M; m++)
    for (int k=0; k<K; k++)
    { ptrM = MatM[m];
```

```
      *ptrR = 0.0;
      for (int n=0; n<N; n++)
      { *ptrR+ = *ptrM++**ptrN;
        *(ptrR + R);
      }
      *ptrR++;
      ptrN = &MatN[0][k+1];
    }
}
```

Приведенный алгоритм перемножения матриц требует порядка N^3 операций умножения для квадратных матриц размера $N \times N$.

Теперь рассмотрим основные принципы разработки эффективных алгоритмов и проиллюстрируем их на примере достаточно сложной для начинающего программиста задачи: обхода фигурой «конь» шахматной доски размера 8×8. Это так называемый *алгоритм поиска с возвратом*.

4 МЕТОДИКА РАЗРАБОТКИ АЛГОРИТМОВ

При разработке алгоритма используют следующие основные принципы.

1) **Поэтапной детализации алгоритма** (другое название – «проектирование сверху – вниз»). Этот принцип предполагает первоначальную разработку алгоритма в виде укрупненных блоков (разбиение задачи на подзадачи) и их постепенную детализацию.
2) **«От главного к второстепенному»**, предполагающий составление алгоритма, начиная с главной конструкции. При этом, часто, приходится «достраивать» алгоритм, двигаясь «в обратную сторону», например, от середины к началу.
3) **Структурирования**, т.е. использования только типовых алгоритмических структур при построении алгоритма. Нетиповой структурой считается, например, циклическая конструкция, содержащая в теле цикла дополнительные выходы из цикла. В программировании нетиповые структуры появляются в результате злоупотребления командой безусловного перехода (goto), использование которой считается «дурным тоном» у программистов на С, потому что в результате применения этого оператора программа хуже читается и существенно труднее отлаживается.

Попробуем продемонстрировать, как эти три принципа работают при решении следующей задачи.

Пример 9.

Постановка задачи. Существует способ обойти шахматным конем доску, побывав на каждом поле ровно по одному разу. Требуется построить алгоритм обхода доски.

Перед началом решения задачи полезно разузнать, что о ней уже известно. Оказывается, это — классическая задача, которая решается в различных постановках уже несколько столетий [2]. В частности, ей посвящены исследования известнейших ученых: швейцарского, русского и немецкого математика и механика Леонарда Эйлера (1707–1783); французского музыканта и математика А.Т. Вандермонда (1735–1796); немецкого и русского математика Ф.Г. Мин-

динга (1806–1885) и многих других. Известно много методов поиска маршрутов коня, но они, как правило, требуют знания многих разделов высшей математики, поэтому сложны для восприятия. По-видимому, самое простой способ построения маршрута принадлежит Варнсдорфу (H.C. von Warnsdorff), который в 1823 году предложил следующее правило обхода доски размером 8□8.

Идея решения задачи. На каждом ходу ставь коня на такое поле, из которого можно совершить наименьшее число ходов на еще не пройденные поля. Если таких полей несколько, разрешается выбирать **любое** из них.

Долгое время считалось, что правило Варнсдорфа работает безупречно. Но после появления достаточно скоростных ЭВМ было установлено, что его вторая часть не точная. Оказывается, что при наличии у коня нескольких «равноценных» возможностей, удовлетворяющих первой части правила, на самом деле они не все являются действительно равноценными. С помощью машинного эксперимента было показано, что произвольное применение правила может завести коня в тупик. Тем не менее, на практике правило Варнсдорфа весьма эффективно даже при вольном использовании его второй части [2].

Формализация задачи. Назовем термином «потенциал поля» количество допустимых ходов коня с этого поля. Введем следующие обозначения.

2¤	3¤	4¤	4¤	4¤	4¤	3¤	2¤
3¤	4¤	6¤	6¤	6¤	6¤	4¤	3¤
4¤	6¤	8¤	8¤	8¤	8¤	6¤	4¤
4¤	6¤	8¤	8¤	8¤	8¤	6¤	4¤
4¤	6¤	8¤	8¤	8¤	8¤	6¤	4¤
4¤	6¤	8¤	8¤	8¤	8¤	6¤	4¤
3¤	4¤	6¤	6¤	6¤	6¤	4¤	3¤
2¤	3¤	4¤	4¤	4¤	4¤	3¤	2¤

Рисунок 4.1. Матрица P

На рисунке 4.1 P — матрица 8×8, содержащая потенциалы полей; R — выходная матрица 8×8, содержащая решение задачи в виде номеров ходов коня; x, y — текущие координаты коня; S_x, S_y — векторы длины 8, содержащие возможные смещения коня относительно текущих его координат:

$$S_x = (\ 1, 2, 2, 1, –1, –2, –2, –1);$$

$$S_y = (–2, –1, 1, 2, 2, 1, –1, –2).$$

Формирование этих векторов иллюстрируется рисунком 4.2, где для произвольных координат коня x,y на бесконечной доске показаны координаты возможных его ходов при обходе их против часовой стрелки. Тогда векторы S_x и S_y будут содержать соответствующие разности между координатами поля x,y и координатами возможных с него ходов коня.

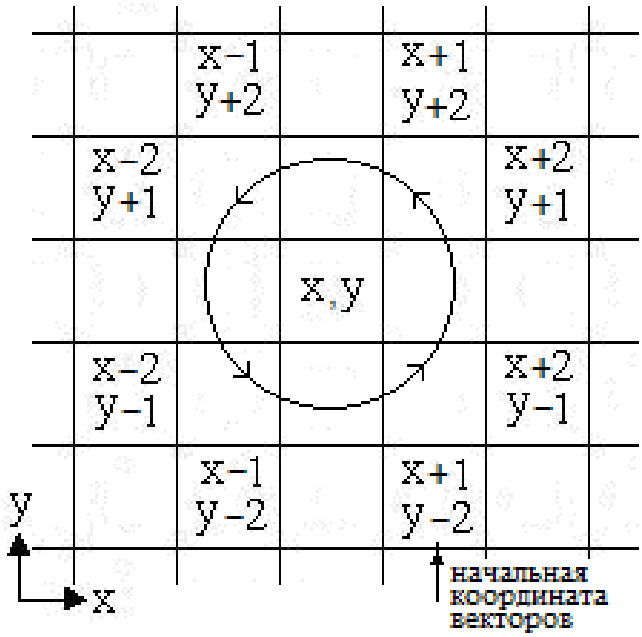


Рисунок 4.2. Окружность показывает направление обхода поля x,y

x_{min}, y_{min} — координаты поля с минимальным потенциалом для текущих (x,y).

p_{min} — значение минимального потенциала допустимого поля.

Сначала все элементы выходной матрицы R обнуляются. Если $R[x][y] = 0$, то конь не был еще на поле (x,y). По мере прохождения конем доски элементам матрицы R присваиваются натуральные числа, соответствующие номерам

ходов коня, при этом конь не может ходить на поля, которые он уже посещал ранее. После заполнения всей матрицы получается решение поставленной задачи.

Разработка алгоритма решения задачи. В соответствии со вторым принципом разработки алгоритма сначала сформируем его главную конструкцию, которая будет остовом будущего алгоритма.

После ввода исходных данных: матриц P, векторов S_x и S_y, координат исходного поля, с которого начнется обход доски x_0, y_0, — элемент матрицы приравнивается 1 ($R[x_0][y_0] = 1$). Далее выбираются последующие ходы, элементам матрицы R присваиваются значения следующих номеров, пока не будет заполнена вся матрица R. Поэтому основная конструкция алгоритма может быть представлена блок-схемой, показанной на рисунке 4.3.

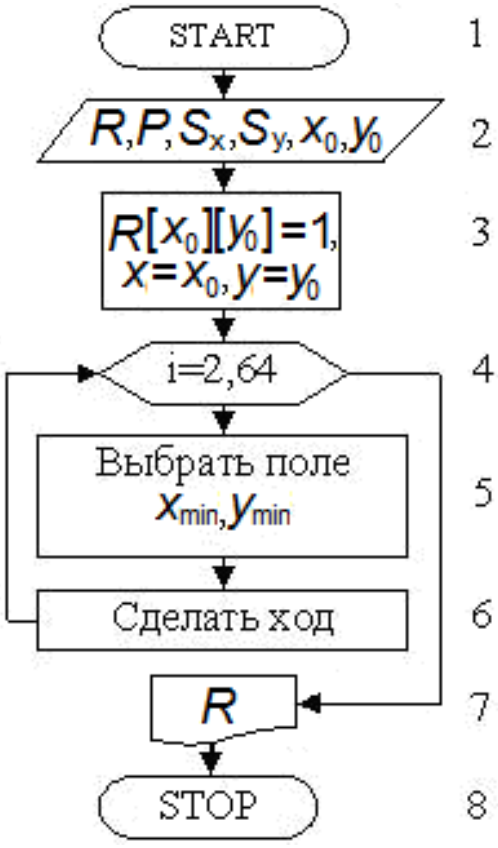


Рисунок 4.3 Основная конструкция алгоритма нахождения маршрута обхода конем шахматной доски

Справа от нее показаны номера блоков, некоторые из которых далее будут детализироваться.

Теперь можно приступить к детализации алгоритма, следуя принципам 1 и 2. Из рисунка 4.3 видно, что необходимо раскрыть содержание блоков 5 и 6; остальные блоки могут быть запрограммированы без дальнейших уточнений.

Начнем с блока 5. С текущего поля с координатами (*x*,*y*), на котором конь находится в настоящее время, нужно сделать ход на поле с наименьшим потенциалом. Поэтому в цикле поочередно перебираются все возможные ходы коня с поля (*x*,*y*) и запоминается поле с минимальным потенциалом. Перед началом перебора возможных ходов потенциалу присваивается фиктивное большое значение, например, 9. Далее, в процессе обхода возможных полей потенциал будет уменьшаться, пока не достигнет минимально возможного значения. Параллельно с поиском поля с минимальным потенциалом будут уменьшены на единицу потенциалы всех исследуемых полей, на которые возможен ход с поля (*x*,*y*), поскольку конь уже посетил это поле, и на него в дальнейшем ходить запрещено.

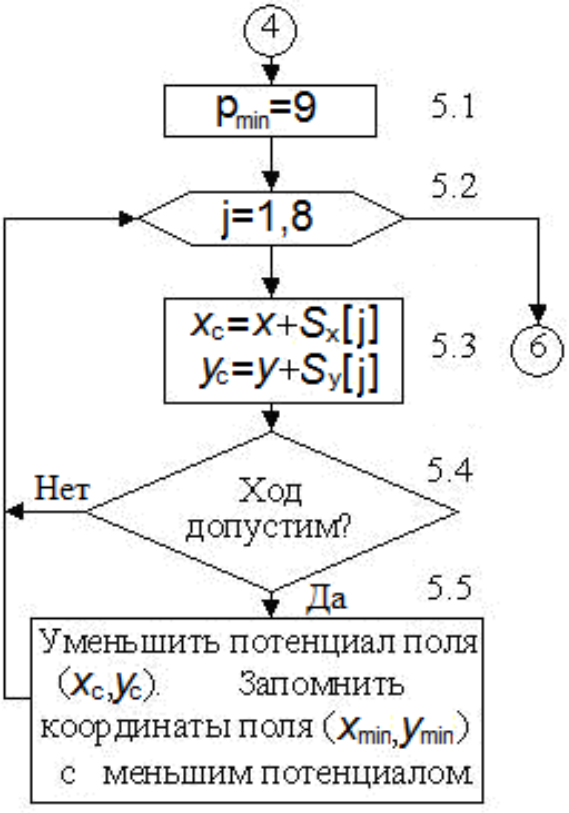


Рисунок 4.4 Детализация блока 5 алгоритма с рисунка 4.3

Схема алгоритма блока 5 показана на рисунке 4.4. Следует обратить внимание на нумерацию блоков в детализирующих блок-схемах. Число до первой точки является номером детализируемого блока в основной схеме (рисунок 4.3). Число после первой точки является номером блока в схеме детализации первого уровня, число после второй точки – второго уровня и т.д.

Входы в детализирующие блок-схемы и выходы из них показаны окружностями с номерами блоков — источников информации и получателей результатов. Например, для схемы с рисунка 4.4: данные поступают от блока 4 основной схемы, а по завершении работы этой части управление с блока 5.2 будет передано блоку 6 основной схемы.

Теперь в детализации нуждаются блоки 5.4 и 5.5. С поля (x,y) можно пойти на поле (x_c,y_c), если его координаты удовлетворяют неравенству $1 \le x_c,y_c \le 8$, кроме того, конь ранее не должен был посещать этого поля. Для проверки второго из сформулированных ограничений можно после посещения конем поля (x,y) его потенциал делать большим (например, 9), что будет свидетельствовать об невозможности повторного посещения. Возможный вариант реализации блока 5.4 показан на рисунке 4.5, где символом & (амперсанд) обозначена логическая операция «И».

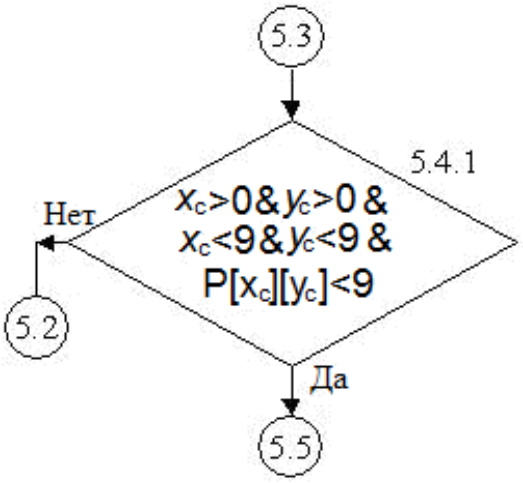


Рисунок 4.5. Детализация блока 5.4 алгоритма с рисунка 4.4

Схема, детализирующая блок 5.5, показана на рисунке 4.6. Идея поиска поля с минимальным потенциалом аналогична рассмотренному выше поиску минимального элемента в матрице.

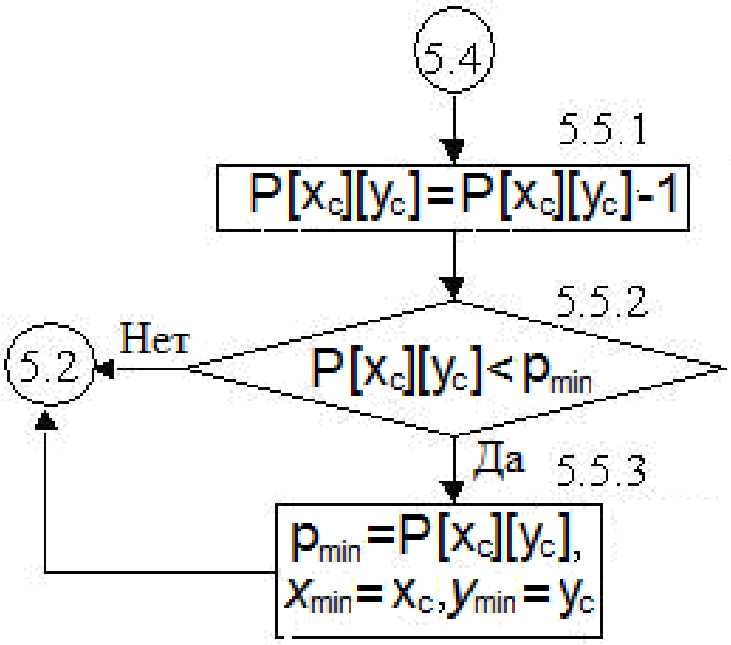


Рисунок 4.6. Детализация блока 5.5 алгоритма с рисунка 4.4

Детализация блока 5.6 основной схемы с рисунка 4.3 понятна без дополнительных пояснений, она показана на рисунке 4.7.

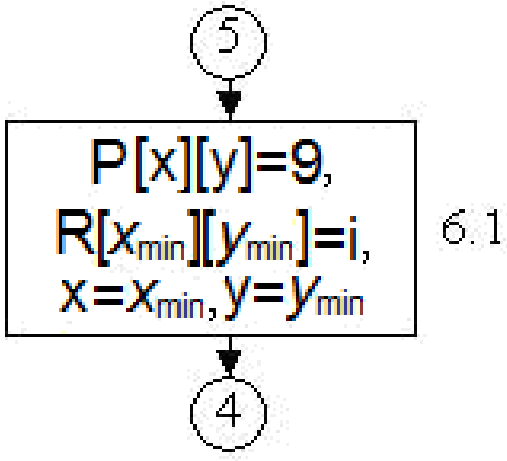


Рисунок 4.7. Детализация блока 6 алгоритма с рисунка 4.3

Фрагмент кода на C, реализующего алгоритм обхода шахматным конем доски размером 8×8, приведен ниже.

```
#include <stdio.h>
void Min_P(int x, int y, int P[], int Sx[], int Sy[],int xmin, int ymin)
// Функция находит поле с минимальным потенциалом для поля (x,y)
{ // Входные параметры x,y – координаты поля, где находится конь; P – матрица
  // потенциалов; Sx,Sy – вспомогательные векторы.
  // Выходные парамеры: xmin, ymin – координаты поля с минимальным потенциалом.
 int xc, yc, j, pmin = 9; // pmin – минимальный потенциал
 for (j=0; j<8; j++)
 {
  xc = x + Sx[j];
  yc = y + Sy[j];
  if(xc>-1 && yc>-1 && xc<8 && yc<8 && P[xc][yc]<9)
  {
   P[xc][yc]--; // уменьшаем потенциал поля, потому что с него нельзя уже пойти
                // на поле с координатами x,y
   if(P[xc][yc] < pmin)
   {
    pmin = P[xc][yc];
    xmin = xc;
    ymin = yc;
   }
  }
 }
} // End Min_P
```

```
int main()
{
 int P[8][8], Sx[8], Sy[8], R[8][8];
// P – матрица потенциалов размера 8x8, показанная при описании алгоритма;
// Sx,Sy – вспомогательные векторы размера 8, показанные при описании алгоритма;
int pmin, xmin, ymin, x, y, x0, y0, move; // move – счетчик ходов коня
 // Здесь должны стоять функции ввода матрицы P, векторов Sx и Sy,
 // ввода координат x0, y0 поля, с которого начнется обход доски.
 x = --x0; // Координаты исходного поля уменьшаются на 1, потому что массивы
 y = --y0; // определены от 0 до 7
 move = 1;
 R[x][y] = move;
 for (move=2; move<65; move++)
 {
  // Ищем поле с минимальным потенциалом, на которое может пойти конь с поля x,y
  Min_P(x, y, P[], Sx[], Sy[], xmin, ymin);
  P[x][y] = 9;
  R[xmin][ymin] = move;
  x = xmin;
  y = ymin;
 }
 // Здесь должна стоять функция вывода матрицы R на экран или печать
} // End main
```

После выполнения этого кода выходная матрица *R* будет содержать номера ходов коня. Причем последовательность этих ходов будет соответствовать маршруту коня по шахматной доске с поля (x_0,y_0).

Но помните, что в определенных случаях конь может зайти в «тупик» — оказаться на поле, откуда уже не будет разрешенных ходов. Поэтому в коде функции *Min_P* следует предусмотреть возможность аварийного выхода, если на ее выходе значение минимального потенциала будет равно 9 (это свидетельствует, что разрешенный ход не найден).

И, наконец, в матрице *R* координаты полей по *x* и *y* изменяются от 0 до 7. Попытайтесь самостоятельно получить на выходе путь коня в принятой шахматной нотации. Например, маршрут начинается с поля с координатами (x_0 = 3 и y_0 = 3). Тогда в матрице *R* элемент (*x* = 2,*y* = 2) станет равным 1.

Допустим, с этого поля конь пошел на поле (*x* = 0,*y* = 1), т.е. R[0][1] = 2; после этого был сделан ход на поле (*x* = 2,*y* = 0), т.е. R[0][1] = 3 и т.д. Задание заключается в преобразовании полученного результата в последовательность полей на шахматной доске, которые посещал конь. Для приведенного примера: c3 – a2 – b3 –…

В этом разделе рассказано о методике проектирования алгоритмов и на примере задачи обхода шахматным конем доски показано, как реализуется эта методика на практике, когда сначала формируется основной скелет алгоритма, а на последующих этапах производится детализация его отдельных блоков.

Используемые источники

1. Ландо, С. К. Счастливые билеты // Математическое просвещение. – МЦНМО, 1998. – № 2. – С. 127–132.

2. Гик, Е.Я. Шахматы и математика. – М.: «Наука». Главная редакция физико-математической литературы. 1983. – 176 с.