Московский государственный технический университет
имени Н. Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети

«УТВЕРЖДАЮ»
Заведующий кафедрой ИУ-6
_____ Сюзев В.В.

Г. С. Иванова, Т.Н. Ничушкина

Разработка алгоритмов простейших программ

Методические указания по выполнению домашних заданий и лабораторных работ
по дисциплине Основы программирования

Москва 2013

Аннотация

Настоящее учебное пособие предназначено для студентов 1 курса кафедр ИУ6 и АК5, обучающихся по программе бакалавра техники и технологии направления «Информатика и вычислительная техника». В пособии рассмотрены основные приемы составления алгоритмов программ разветвленной и циклической структуры. Кратко пояснены математические методы решения некоторых задач вычислительной математики, а также пояснены приемы обработки массивов и матриц.

Оглавление

Введение

Умение хорошо писать программы для компьютера предполагает не только хорошее владение средствами разработки программ, но и хорошо развитое алгоритмическое мышление. Вся практика программирования говорит о том, что именно отсутствие алгоритмического мышления – причина неудач студентов при изучении программирования.

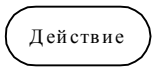
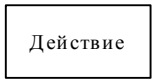
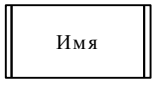
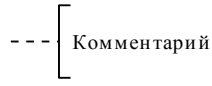
Однако алгоритмическое мышление, как умение построить последовательность действий, приводящую к решению задачи, можно и нужно развивать. Для этого необходимо проработать алгоритмы многих небольших программ, накапливая приемы, используемые при их составлении. Таких приемов сравнительно немного, и их совокупность образует ту базу, которая позволит студентам научиться писать программы.

В настоящем пособии представлены базовые алгоритмы, без изучения которых знания по данному предмету будут не полными.

Совершенно сознательно авторы не приводят текстов программ, ограничившись схематическими представлениями алгоритмов. Тексты программ, изобилующие деталями средств программирования, отвлекали бы внимание от самих алгоритмов, что было бы крайне нежелательно.

Для представления алгоритмов в пособии использованы графические обозначения основных алгоритмических блоков по ГОСТ 19.701–90 (см. таблицу 1).

Таблица 1

Название блока	Обозначение	Назначение блока
1. Терминатор		Начало, завершение программы или подпрограммы
2. Процесс		Обработка данных (вычисления, пересылки и т.п.)
3. Данные		Операции ввода-вывода
4. Решение		Ветвления, выбор, итерационные и поисковые циклы
5. Подготовка		Счетные циклы
6. Граница цикла		Любые циклы
7. Предопределенный процесс		Вызов процедур
8. Соединитель		Маркировка разрывов линий
9. Комментарий		Пояснения к операциям

Применение схем для изображения алгоритмов позволяет выполнить их достаточно формальное представление, к тому же задействующее более наглядное их зрительное представление.

Использование псевдокодов при этом не даст требуемого результата – образования у обучаемого некоторой базы, позволяющей самостоятельно разрабатывать алгоритмы.

1 Разветвляющиеся процессы

В процессе решения многих задач возникает ситуация, когда дальнейшие вычисления зависят от выполнения некоторого условия. Если условие будет выполнено, то вычисления будут производиться по одному определенному правилу, если условие не выполняется – по другому. Такие вычислительные процессы называют *разветвляющимися* (ветвящимися). Каждое отдельное направление вычислений называется *ветвью*.

В качестве примера разветвляющегося вычислительного процесса рассмотрим алгоритм вычисления корней квадратного уравнения

Пример 1.1. Определение действительных корней квадратного уравнения:

$$ax^2+bx+c=0.$$

В зависимости от значения дискриминанта $D = b^2 - 4ac$, уравнение имеет либо два действительных корня, либо один, либо вообще не имеет действительных корней. Поэтому, для того, чтобы найти корни квадратного уравнения, необходимо предварительно вычислить дискриминант D и проверить выполнение условий $D<0$ и $D=0$. Схема алгоритма вычисления корней квадратного уравнения приведена на рисунке 1.1.

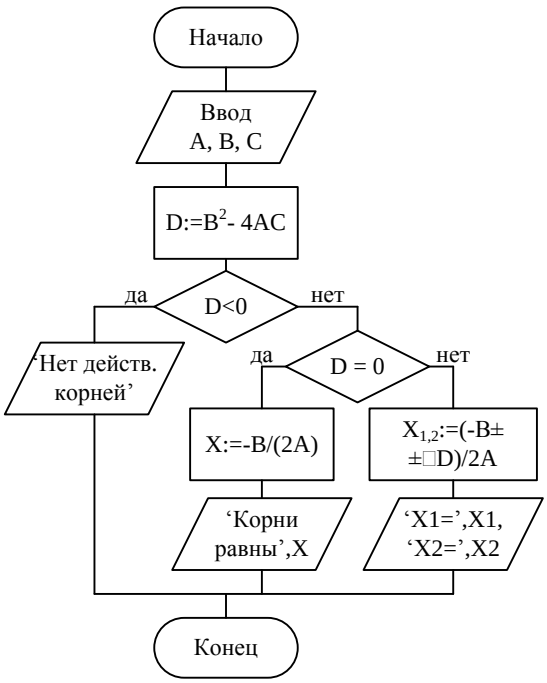


Рисунок 1.1 – Схема алгоритма вычисления действительных корней квадратного уравнения

Если условий много, то процесс составления алгоритма, содержащего минимальное количество проверок, может вызвать некоторые трудности. В этом случае бывает удобным использовать, так называемые, *таблицы решений*.

Таблицы решений. Таблица решений составляют следующим образом. По вертикали выписывают все условия, от которых зависят дальнейшие вычисления, а по горизонтали – все варианты вычислений. На пересечении каждого столбца и строки указывают:

- букву Y, если для данного варианта условие должно выполняться,
- букву N, если условие обязательно должно не выполняться,
- прочерк, если исход сравнения не важен.

Например, для алгоритма вычисления корней квадратного уравнения можно составить следующую таблицу:

	Нет корней	$x = -b/2a$	$x = \pm(-b\sqrt{D}/2a$
D < 0	Y	N	N
D = 0	N	Y	N

Схему алгоритма строят по таблице. Сначала проверяют выполнение первого условия. Из таблицы следует, что первое условие должно выполняться только для первого варианта решения, поэтому после его проверки ветвь «да» соответствует случаю «нет корней». В ветви «нет» необходимо проверить условие D=0 и, в зависимости от выполнения этого условия, указать оставшиеся два случая.

Иногда, составленная таблица решений имеет сложный вид. Рассмотрим, например, таблицу:

	P1	P2	P3	P4
Условие 1	Y	-	N	Y
Условие 2	N	Y	N	N
Условие 3	Y	-	-	N

где P1, P2, P3, P4 – варианты решений.

Если сразу приступить к построению алгоритма, то будет получена схема алгоритма, приведенная на рисунке 1.2.

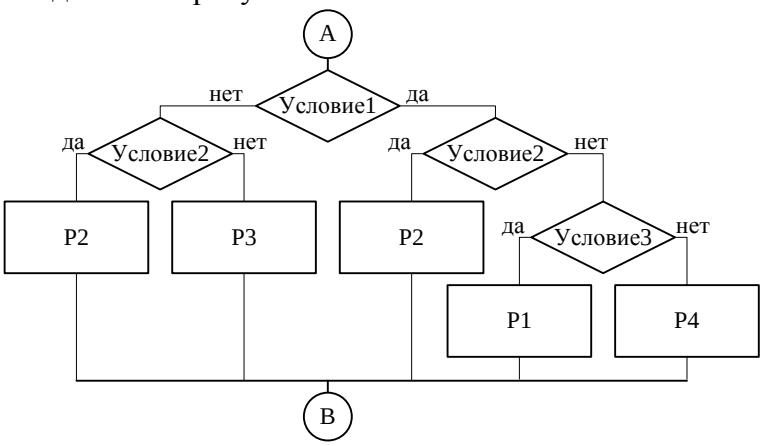


Рисунок 1.2 – Неоптимальная схема фрагмента алгоритма

Этот алгоритм не кажется простым, и его построение вызывает определенные трудности, но его можно существенно упростить, если в таблице поменять местами проверяемые условия. Кроме того, для удобства построения алгоритма, целесообразно поменять местами столбцы таблицы:

	P1	P4	P3	P2
Условие 2	N	N	N	Y
Условие 1	Y	Y	N	-
Условие 3	Y	N	-	-

Алгоритм, построенный по такой таблице, окажется значительно проще, к тому же его построение требует меньших усилий (см. рисунок 1.3).

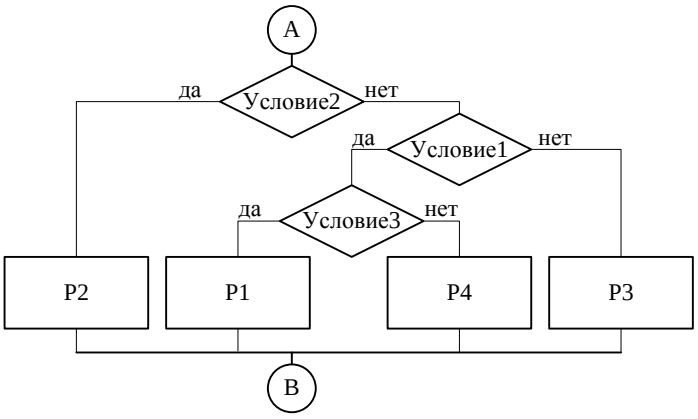


Рисунок 1.3 – Оптимизированный фрагмент схемы алгоритма

Рассмотрим еще один пример использования таблиц решений.

Пример 1.2. Решить систему уравнений:

$$\begin{cases} ax = b; \\ x + cy = 1. \end{cases}$$

При составлении алгоритма решения задачи необходимо рассмотреть 5 случаев:

1) ***a***=0, ***b***=0, тогда ***x***=1-***cy***, ***y*** - любое число;

2) ***a***=0, ***b***≠0, тогда решений нет;

3) ***a***≠0, ***c***=0, ***a***=***b***, тогда x=1, y любое число;

4) ***a***≠0, ***c***=0, ***a***≠***b***, тогда решений нет:

5) ***a***≠0, ***c***≠0, тогда ***x***=***b***/***a***, ***y***=(***a***-***b***)/(***ac***).

Составим таблицу решений, указывая в первом столбце условие, которое необходимо проверить:

	1	2	3	4	5
$a = b$	–	–	*Y*	*N*	–
$a = 0$	*Y*	*Y*	*N*	*N*	*N*
$b = 0$	*Y*	*N*	–	–	–
$c = 0$	–	–	*Y*	*Y*	*N*

Теперь преобразуем таблицу для более удобной реализации. На первое место целесообразно поставить условие ***a***=0, т.к. соответствующая строка не содержит прочерков, и, следовательно, действия однозначно разделятся на две ветви. Вторым лучше проверять условие ***b***=0, т.к. на оставшиеся два условия оно не оказывает никакого влияния. Третьим условием удобно взять ***c***=0. В результате получили таблицу в следующем виде:

	1	2	3	4	5
$a = 0$	*Y*	*Y*	*N*	*N*	*N*
$b = 0$	*Y*	*N*	–	–	–
$c = 0$	–	–	*Y*	*Y*	*N*
$a = b$	–	–	*Y*	*N*	–

Используя эту таблицу, мы можем построить алгоритм, в котором достаточно сделать всего четыре проверки условий (см. рисунок 1.4).

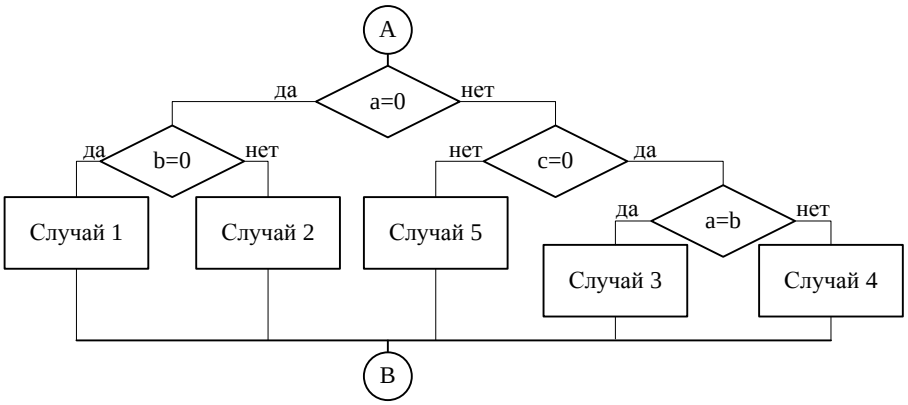


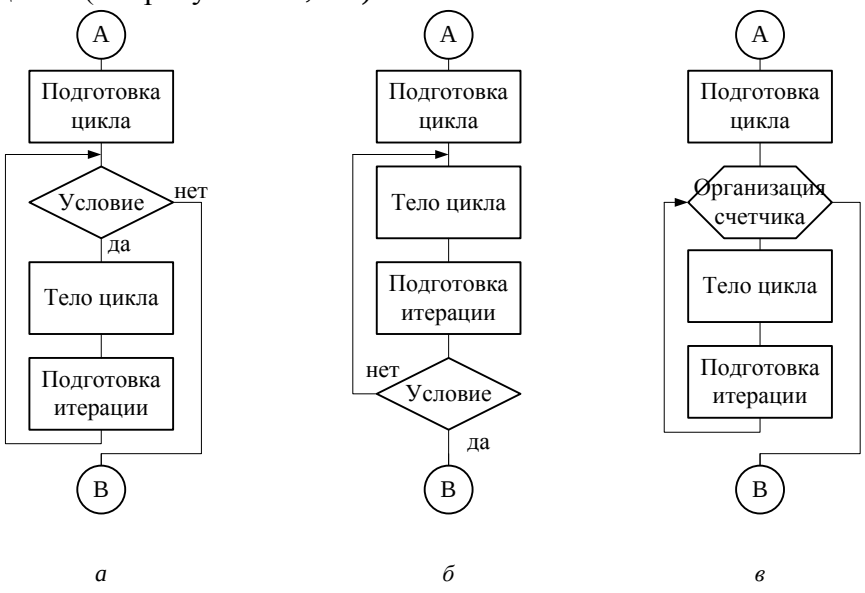
Рисунок 1.4 – Фрагмент схемы алгоритма решения системы уравнений

2 Циклические процессы. Алгоритмы решения задач вычислительной математики

2.1 Типы циклических процессов

При решении многих задач возникает необходимость многократного повторения одних и тех же действий, но над различными значениями переменных. Такие вычислительные процессы называются *циклическими*, а многократно повторяющиеся участки – *циклами*.

В любом процедурном языке программирования существуют средства реализации трех типов алгоритмов циклической структуры: цикла-пока, цикла-до и счетного цикла (см. рисунки 2.1, *а-в*).



а *б* *в*

Рисунок 2.1 – Три структурные конструкции циклов:
а – цикл-пока; б – цикл-до; в – счетный цикл

Каждый из трех алгоритмов циклических процессов состоит из нескольких основных этапов:

- *подготовка цикла* – задание начальных значений переменным, изменяющимся в цикле;
- *тело цикла* – действия, выполняемые непосредственно в цикле;
- *подготовка итерации* – изменение значений переменных для нового выполнения тела цикла;
- *условие* – проверка условия продолжения или окончания цикла;
- *организация счетчика* – задание начального и конечного значения, а также шага переменной цикла.

Тело цикла может иметь сложную структуру, в том числе может включать ветвления и другие циклы (*вложенные циклы*). При программировании алгоритмов со структурой вложенных циклов необходимо выполнять следующее правило: внутренний оператор цикла и принадлежащая ему область действий должны полностью содержаться в области внешнего оператора цикла. Иными словами внешний цикл всегда начинается раньше, а заканчивается позже, чем внутренний цикл.

Различают циклы с заданным числом повторений и циклы с заранее не известным числом повторений (*итерационные циклы*). Итерационные циклы харак-

теризуются последовательным приближением к исходному значению с заданной точностью.

В конкретных типах циклических процессов указанные этапы цикла могут присутствовать в различных сочетаниях, однако порядок их проектирования не меняется:

1) определяются действия, составляющие тело цикла;

2) заполняется блок подготовки итерации;

3) определяется условие выхода из цикла (счетчик или логическое выражение);

4) заполняется блок подготовки цикла.

Пример 2.1. Задано натуральное число *a*. Получить все члены последовательности *a, a*^{2}*, a*^{3}, …, не превышающие числа *b*.

Схема алгоритма решения этой задачи представлена на рисунке 2.2.

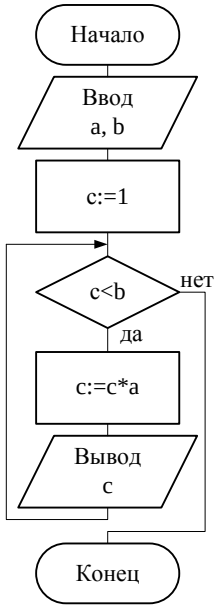


Рисунок 2.2 – Схема алгоритма получения последовательности

При подготовке цикла присваивается начальное значение вспомогательной переменной *c*. Тело цикла включает последовательное вычисление переменной *c*, которая равна *a, a*^{2}*, a*^{3}, … Изменение значения *c* происходит до тех пор, пока оно не станет большим или равным значению *b*. Если $a \geq b$, то не будет выведено не одного элемента последовательности.

2.2 Табулирование функции

Типичным примером циклического процесса с заданным числом повторений является задача *табулирования функции* (получение таблицы значений функции).

Задача сводится к вычислению значений функции $y = f(x)$ при изменении x от начального значения a до конечного значения b с постоянным шагом h. Такая программа реализуется посредством цикла с заданным числом повторений, причем число повторений определяется по формуле:

$$n = \left[\frac{b-a}{h}\right] + 1.$$

Решение задачи можно осуществить с помощью алгоритма, схема которого приведена на рисунке 2.3.

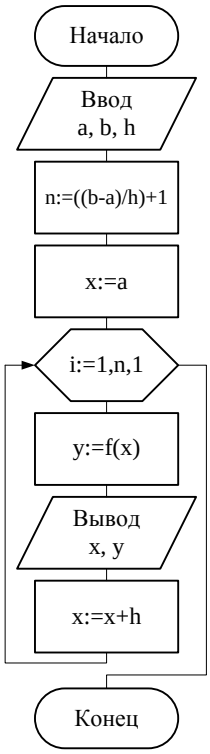


Рисунок 2.3 – Схема алгоритма вычисления значений функции

Здесь при подготовке цикла вычисляется количество его повторений и присваивается начальное значение переменной *x*. В теле цикла рассчитывается значение функции в точке *x* и, затем, выводится значение *x* и *y*. При подготовке итерации изменяется значение *x*.

2.3 Нахождение суммы ряда

Многие задачи вычислительной математики сводятся к вычислению суммы рядов вида $S = \sum_{i=1}^{\infty} r_i$. Определение суммы предполагает последовательное вычисление члена ряда r_i и прибавление его к частичной сумме *S* до тех пор, пока не будет выполнено условие выхода из цикла. Общий вид схемы алгоритма решения такой задачи приведен на рисунке 2.4.

Условие окончания цикла зависит от поставленной задачи, как правило, оно связано с достижением требуемой точности.

Примеры постановок задач.

1. Найти сумму первых *n* членов ряда $S = \sum_{i=1}^{\infty} r_i$. В этом случае условие выхода из цикла: $i > n$.

2. Вычислять сумму ряда $S = \sum_{i=1}^{\infty} r_i$ до тех пор, пока очередной член ряда r_i не будет меньше заданного ε. Здесь условием выхода из цикла будет: $| r_i | < \varepsilon$.

3. Вычислить сумму ряда $S = \sum_{i=1}^{\infty} r_i$ с точностью ε, если известно точное значение суммы, равное *A*. Здесь условие выхода из цикла: $| S - A | < \varepsilon$.

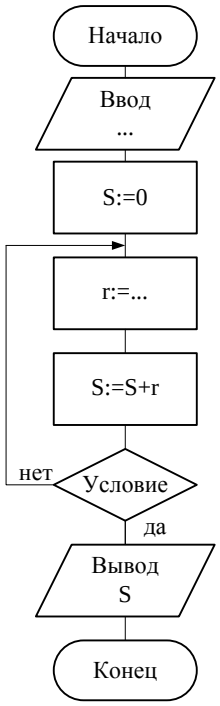


Рисунок 2.4 – Обобщенная схема алгоритма вычисление суммы ряда

Очередной член ряда вычисляют одним из следующих способов:

- в простейших случаях – по формуле общего члена ряда, например:

1) $S=\sum_{n=1}^{\infty}\frac{1}{n^2+1}$, $r_n=\frac{1}{n^2+1}$;

2) $S=\sum_{n=1}^{\infty}\frac{\cos nx}{n}$, $r_n=\frac{\cos nx}{n}$;

- если в формулу общего члена входят целые степени и факториалы, то – по рекуррентным формулам через уже вычисленное значение предыдущего члена, исключая повторные вычисления.

В этом случае каждый последующий член отличается от предыдущего на один и тот же сомножитель, поэтому вычисления с использованием предыдущего значения будут наиболее эффективны, например:

1) $S=\sum_{n=1}^{\infty}\frac{x^n}{n!}$, $r_n=r_{n-1}\frac{x}{n}$;

2) $S=\sum_{n=1}^{\infty}\frac{(-1)^n x^{2n}}{n!}$, $r_n=r_{n-1}\frac{(-1)\,x^2}{n}$;

- если член ряда удобно представить в виде произведения, один из сомножителей вычисляется по рекуррентному соотношению (т.е. по предыдущему члену), второй – непосредственно, например:

$$S=\sum_{k=1}^{\infty}\frac{(-1)^k x^k}{k},\quad r_n=\underbrace{(-1)^n x^n}_{p_n}\frac{1}{n}=p_{n-1}\frac{(-1)x}{n}=p_{n-1}\cdot(-1)x\cdot\frac{1}{n}\quad .$$

2.4 Приближенное вычисление определенных интегралов

Пусть требуется вычислить определенный интеграл $\int_a^b f(x)dx$, где *f(x)* – некоторая *непрерывная* функция, заданная на промежутке [*a, b*]. Как известно, значение определенного интеграла равно площади криволинейной трапеции, ограниченной подынтегральной функцией. Простейшими методами приближенного вычисления определенных интегралов являются метод прямоугольников и метод трапеций.

2.4.1 Метод прямоугольников

Разобьем отрезок [*a,b*] точками $a=x_0 < x_1 < x_2 <... < x_{n-1} <x_n=b$, причем $|x_i, x_{i+1}|=(b-a)/n$. Заменим площадь каждой криволинейной трапеции с основанием $[x_i, x_{i+1}]$ площадью прямоугольника со сторонами $(b-a)/n$ и $f(x_i)$ (см. рисунок 2.5).

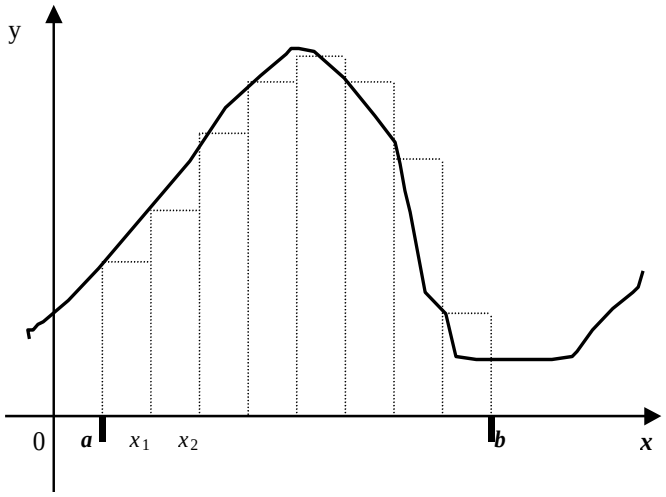


Рисунок 2.5 – Метод прямоугольников

Тогда площадь каждого такого прямоугольника вычисляется по формуле: $S_i=f(x_i)(b - a)/n$. Сумма всех площадей полученных прямоугольников приближенно равна значению определенного интеграла:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} \frac{(b-a)f(x_i)}{n} = \frac{(b-a)}{n}\sum_{i=0}^{n-1} f(x_i) .$$

Очевидно, что чем больше количество отрезков разбиения, тем выше точность вычислений.

Пример схемы алгоритма вычисления определенного интеграла методом прямоугольников приведен на рисунке 2.6.

В этом алгоритме через *N* обозначено количество точек разбиения, *d* – длина отрезков разбиения. Тело цикла алгоритма содержит внутренний цикл с заданным числом повторений. Число повторений внутреннего цикла представляет собой количество прямоугольников, на которые мы разбили криволинейную трапецию. После отработки внутреннего цикла проверяется условие достижения требуемой точности вычисления. Если точность недостаточна, то количество отрезков разбиения увеличивается (например, удваивается) и вычисления площади повторяются вновь.

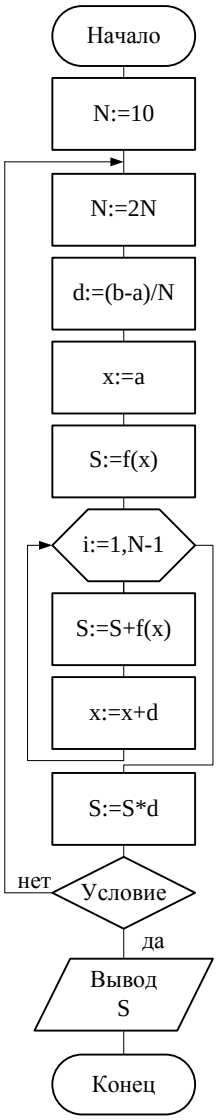


Рисунок 2.6 – Схема алгоритма вычисления интеграла методом прямоугольников

Условие точности зависит от поставленной задачи. Будем пользоваться двумя способами оценки погрешности:

1. Если задано точное значение интеграла, равное I, то условием выхода из цикла будет условие $| S - I | < \varepsilon$.

2. Если не задано точное значение интеграла, то будем сравнивать сумму, полученную при последнем разбиении ***n***, с суммой, полученной при предыдущем значении параметра *n*.

2.4.2 Метод трапеций

Метод трапеций аналогичен методу прямоугольников, только здесь криволинейная трапеция заменяется не прямоугольниками, а трапециями (см. рисунок 2.7).

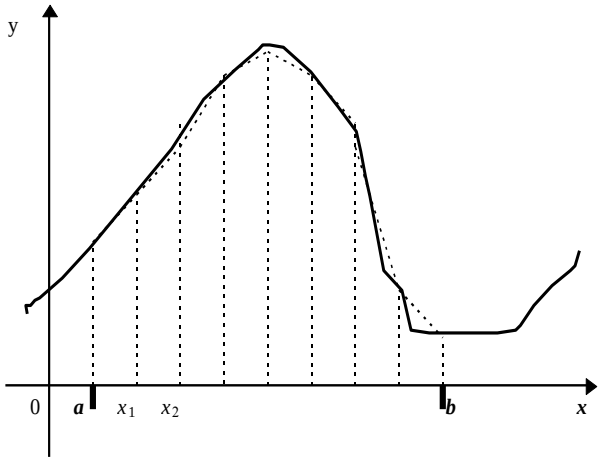


Рисунок 2.7 – Метод трапеций

Формула приближенного вычисления определенного интеграла методом трапеций примет вид:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} \frac{(b-a)(f(x_i)+f(x_{i+1}))}{2n} = \frac{(b-a)}{n}\sum_{i=0}^{n-1} \frac{f(x_i)+f(x_{i+1})}{2}$$

или

$$\int_a^b f(x)dx \approx \left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right)\frac{(b-a)}{n}$$

2.5 Определение корней уравнения

Рассмотрим уравнение *f(x)*=0, о котором известно, что оно имеет корень на отрезке [*a, b*], причем функция *f(x)* на этом интервале непрерывна и на концах отрезка [*a, b*] принимает различные знаки (т.е. *f(a)f(b)*<0). Излагаемые здесь методы состоят в приемах, при помощи которых находится новый интервал [a_1, b_1], такой что

$$a \le a_1 < x_0 < b_1 \le b,$$

где x_0 корень уравнения.

2.5.1 Метод половинного деления

Этот метод является одним из самых простых. Приближение к корню уравнения здесь осуществляется с помощью деления отрезка [*a, b*] пополам. В результате мы получаем новую точку *c* (см. рисунок 2.8).

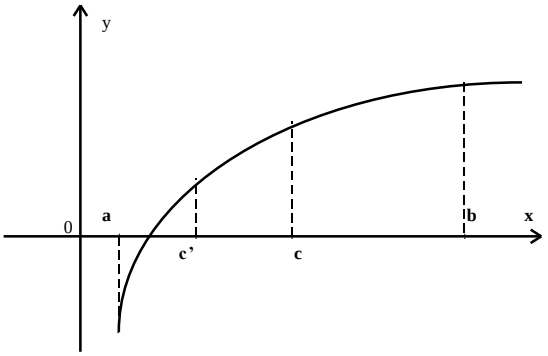


Рисунок 2.8 – Метод половинного деления

Далее проверяется, является ли точка ***c*** корнем уравнения *f(x)*=0. В случае положительного ответа задача решена, в случае отрицательного мы сравниваем

знак функции в точке *c* со знаками функций в концах отрезка и, в дальнейшем, будем рассматривать тот отрезок ([*a, c*] или [*c, b*]), где функция принимает разные знаки. Новый отрезок снова делится пополам, и все действия повторяются вновь. Таким образом, длина рабочего отрезка будет постоянно уменьшаться, а искомый корень уравнения будет оставаться внутри этого отрезка. Вычисления прекращаются тогда, когда будет достигнута заданная точность вычислений ε (т.е. $|f(c)|< \varepsilon$).

2.5.2 Метод хорд

Метод хорд аналогичен методу половинного деления, но дает более быстрое приближение к корню уравнения. Здесь новая точка выбирается по несколько другому закону. Точки с координатами (*a, f*(*a*)) и (*b, f*(*b*)) соединяются отрезком (хордой) и точка ***c*** выбирается как точка пересечения полученной хорды с осью OX(см. рисунок 2.9.)

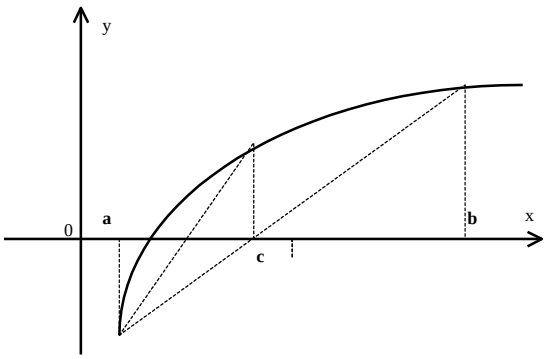


Рисунок 2.9 – Метод хорд

Координату точки ***c*** находят по формуле:

$$c=a-f(a)\frac{b-a}{f(b)-f(a)} \text{ или } c=b-f(b)\frac{b-a}{f(b)-f(a)}$$

2.6 Нахождение длины кривой

Пусть на отрезке [***a***,***b***] уравнением ***y=f(x)*** задана непрерывная кривая. Нужно найти длину, заданной кривой. Задачу можно решить, если разбить отрезок [*a,b*] на несколько частей, см. рисунок 2.10.

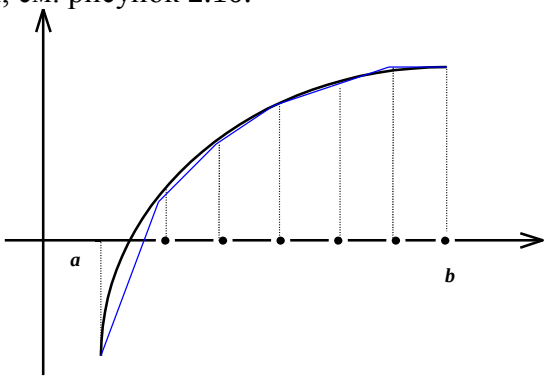


Рисунок 2.10 – Метод хорд нахождения длины кривой

Если теперь провести прямые, параллельные оси OY, то кривая будет разбита на такое же количество более мелких кривых. Соединив точки разбиения кривой прямым линиями, получим ломаную, длину которой определить не представляет большого труда, поскольку каждая вершина имеет координаты (x_i, *f*(*x*)). Очевидно, что чем больше звеньев у ломанной, тем ближе ее длина к длине кривой.

Поэтому алгоритм вычисления можно построить следующим образом: нужно увеличивать количество точек разбиения и вычислять длину ломанной до тех пор, пока не будет достигнута требуемая точность.

3 Массивы

Массив – это упорядоченная совокупность однотипных данных, с каждым из которых связан упорядоченный набор целых чисел, называемых *индексами*. Индексы определяют положение элемента в массиве.

Работа с массивом сводится к действиям над его элементами. Для обращения к конкретному элементу массива необходимо указать имя массива и значение индекса (индексов) элемента. Все задачи по работе с массивами можно разбить на следующие классы:

1. Последовательная обработка элементов массива.
2. Выборочная обработка элементов массива.
3. Изменение порядка следования элементов без изменения размеров исходного массива.
4. Переформирование массива с изменением его размеров.
5. Одновременная обработка нескольких массивов или подмассивов.
6. Поиск в массиве единственного элемента, отвечающего некоторому условию.

Выделение шести перечисленных классов обусловлено тем, что каждому из них соответствуют приемы программирования. В реальной жизни задачи в чистом виде встречаются довольно редко. Однако любая сложная задача может быть разбита методом пошаговой детализации на более простые задачи, относящиеся к указанным выше классам. Следует также учитывать, что существует специфика реализации решений всех классов задач, зависящая от количества индексов массивов.

3.1 Приемы обработки одномерных массивов

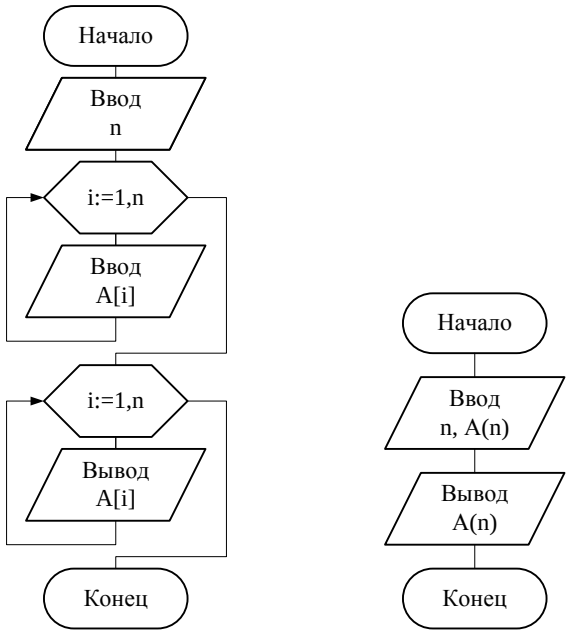
Одномерными называют массивы, для доступа к элементам которых необходимо задавать один индекс. Рассмотрим наиболее распространенные приемы программирования обработки одномерных массивов.

3.1.1 Последовательная обработка элементов массива

Примерами подобного класса задач служат: нахождение суммы элементов, произведения элементов, среднего арифметического, среднего геометрического, подсчет количества элементов, отвечающих определенному условию или обладающих некоторыми признаками, а также их суммы, произведения и т.д. Кроме того, к этой группе могут быть отнесены задачи ввода и вывода массивов. Особенностью задач класса является то, что количество обрабатываемых элементов массива известно. Это позволяет использовать счетные циклы, параметр которых обеспечивает доступ к элементам. Однако, возможно применение и других менее эффективных в рассматриваемом случае типов циклов.

Рассмотрим некоторые типовые алгоритмы задач первого типа.

Пример 3.1. Пусть необходимо ввести или вывести массив. Для этого нужно последовательно перебрать все элементы массива. Если количество вводимых элементов известно заранее, то для этой операции удобно использовать счетные циклы. При неизвестном количестве элементов лучше использовать циклы с пред- или пост условиями. При выводе количество элементов может быть известно заранее или задаваться пользователем. Схема алгоритма, при помощи которого осуществляется ввод количества элементов n и ввод-вывод одномерного массива A(n) представлен на рисунке 3.1, *а*.



а *б*

Рисунок 3.1 – Ввод-вывод одномерного массива

Поскольку ввод-вывод массивов всегда осуществляется одинаково, каждый раз расписывать эти операции нет необходимости. На схеме алгоритма операции ввода-вывода массива обычно не раскрывают (см. рисунок 3.1, *б*).

Пример 3.2. Сравнительно часто встречается задача нахождения наибольшего или наименьшего элемента массива (например, при построении графика функции).

Вспомогательной переменной *Amax* присваивается значение первого элемента массива, затем последовательно все элементы массива сравниваются с переменной *Amax*, если значение *Amax* оказывается больше, чем очередной элемент, то *Amax* присваивается новое значение (см. рисунок 3.2).

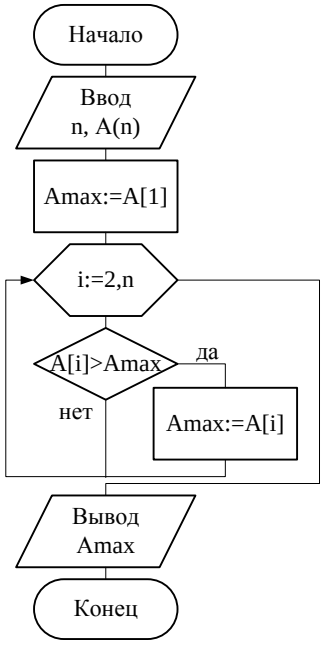


Рисунок 3.2 – Схема алгоритма нахождения максимального элемента массива

Пример 3.3. Пусть необходимо найти среднее арифметическое значение положительных элементов целочисленного массива, кратных трем. Для решения этой задачи сначала необходимо двум вспомогательным переменным *Sum* и *Kol*, которые будут использоваться для накопления суммы требуемых элементов и их количества, присвоить нулевые значения (см. рисунок 3.3).

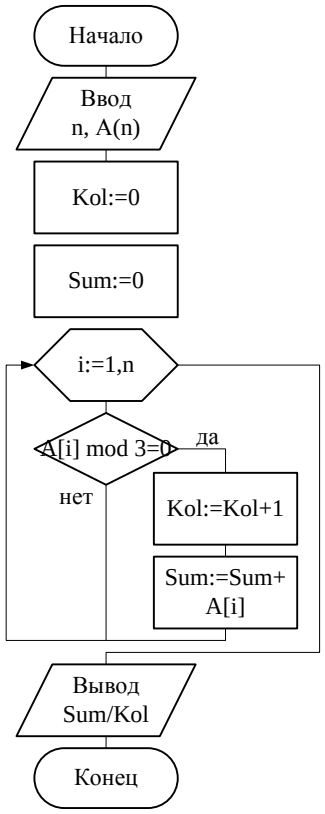


Рисунок 3.3 – Схема алгоритма нахождения среднего арифметического элементов, кратных 3

После этого осуществляется перебор всех элементов и, если очередной удовлетворяет условию, его значение добавляется к содержимому переменной *Sum*, а значение переменной *Kol* увеличивается на единицу. После просмотра массива среднее арифметическое определяется делением накопленной суммы на количество найденных элементов.

Пример 3.4. Пусть необходимо заменить все отрицательные элементы массива на нулевые.

Несложно увидеть, что схема очень похожа на предыдущую, поскольку, как в предыдущем алгоритме, осуществляется последовательная проверка всех элементов массива.

Схема алгоритма решения задачи показана на рисунке 3.4.

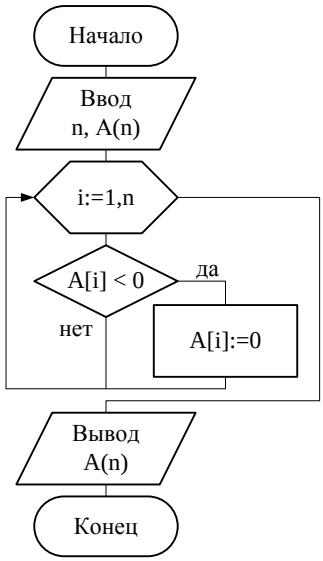


Рисунок 3.4 – Схема алгоритма замены отрицательных элементов массива нулями

3.1.2 Выборочная обработка элементов массива

К задачам данного типа относятся задачи по формулировке сходные с задачами первого класса, но обработка ведется не надо всеми элементами массива, а только теми, которые имеют вполне определенное значение индексов. Поэтому, особенностью этого класса задач является наличие определенного закона изменения индексов рассматриваемых элементов. С целью уменьшения времени работы программы и увеличения ее эффективности, программисту следует найти этот закон и обеспечить его исполнение. В зависимости от закона изменения индексов могут использоваться все виды циклов, а также их сочетание.

Пример 3.5. Определить максимальный элемент, среди элементов целочисленного массива, стоящих на четных местах.

При решении этой задачи начинать надо со второго элемента и увеличивать номер на два после каждой проверки. Количество анализируемых элементов при этом уменьшается вдвое. Схема алгоритма решения задачи представлена на рисунке 3.5.

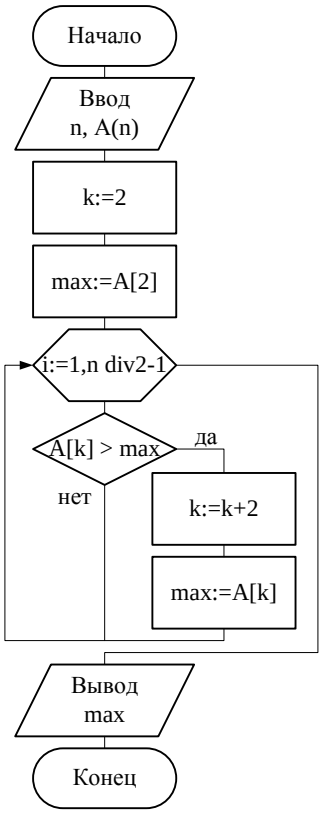


Рисунок 3.5 – Схема алгоритма поиска максимального элемента, записанного на четном месте

Пример 3.6. Заменить каждый третий элемент массива значением 0 – если он четен и остатком от деления элемента на пять, если он нечетен.

В этом случае нас интересует каждый третий элемент, поэтому количество просматриваемых элементов уменьшается втрое, а значение *k* в цикле увеличивается на три (см. рисунок 3.6).

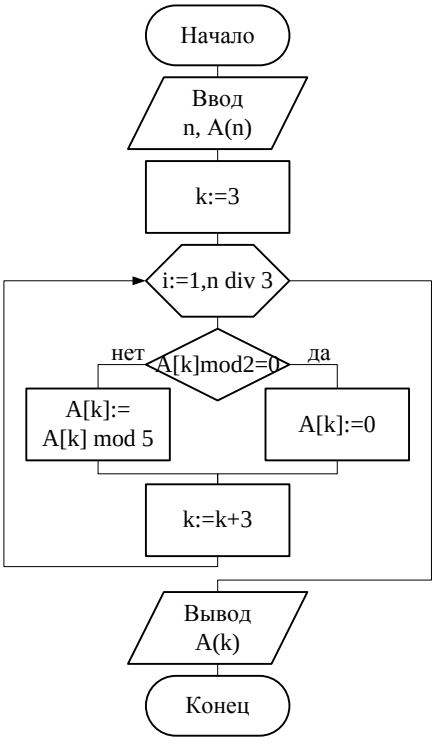


Рисунок 3.6 – Схема алгоритма замены каждого третьего элемента массива

3.1.3 Изменение порядка следования элементов без изменения размеров исходного массива. Сортировка массива

Примерами подобной обработки служат: замена значений некоторых элементов на другие в соответствии с определенными правилами, сортировка массивов по убыванию, возрастанию, и по любому другому признаку, перестановки и сдвиги элементов различного характера и др. Особенностью этого класса задач является то, что, несмотря на постоянство количества элементов массива, для перестановки или замены элемента сначала его необходимо найти. Таким образом, чтобы преобразовать массив, требуется несколько раз его просмотреть. Поэтому даже для одномерного массива требуется применение нескольких вложенных циклов. Рассмотрим несколько типовых приемов.

Пример 3.7. Переформировать массив таким образом, чтобы сначала шли все положительные элементы, затем отрицательные и в конце – нулевые.

Прежде всего, задачу сложно решить, не используя дополнительный массив, поскольку запись элементов происходит в заранее неизвестном порядке.

Также необходимо учитывать, что номера элементов в исходном и новом массиве не совпадают. В качестве индекса исходного массива мы, как и ранее, будем использовать переменную цикла. А для указания номера элемента в новом массиве нам понадобится специальный индекс(ы), который придется изменять отдельным оператором.

И последнее, осуществить требуемое переформирование массива за один проход невозможно, поскольку количество элементов каждого типа неизвестно. Задачу можно решить, используя три цикла, в которых в новый массив последовательно переписываются сначала положительные элементы, затем отрицательные и наконец – нулевые. Однако возможно и использование двух циклов: в первом перепишем в начало нового массива положительные элементы, а в конец – нулевые, начиная с последнего, а во втором просто допишем в середину отрицательные элементы.

Окончательный вариант алгоритма приведен на рисунке 3.7.

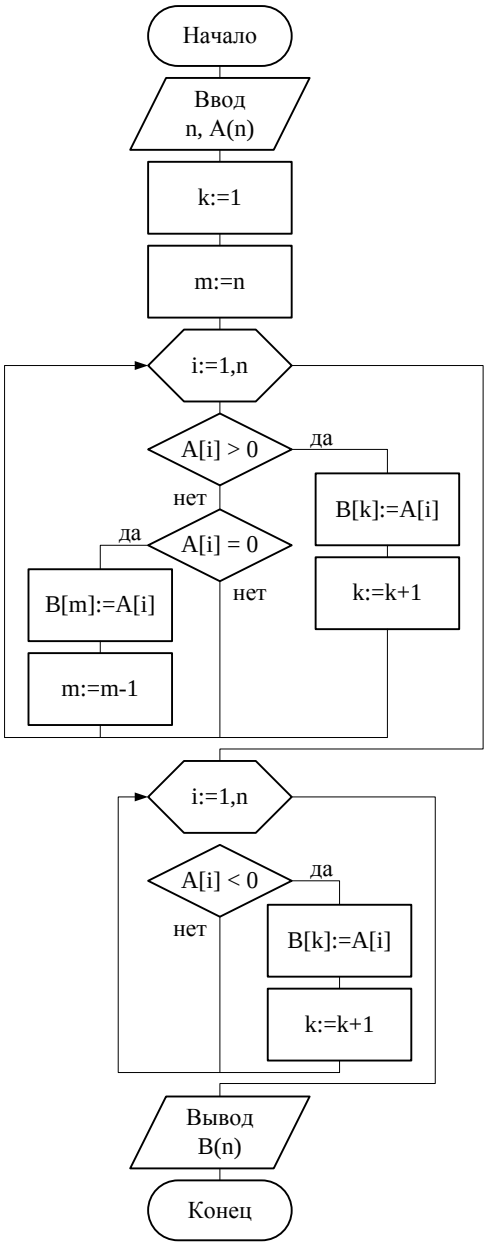


Рисунок 3.7 – Схема алгоритма перестановки элементов массива

Однако наиболее распространенными задачами второго класса являются сортировки массивов.

Сортировка – это изменение порядка элементов в массиве таким образом, что значения элементов становятся упорядоченными по некоторому закону. На практике сортируемые массивы обычно имеют большую размерность, поэтому для любого алгоритма сортировки оценивают *вычислительную сложность*, которая определяется зависимостью времени сортировки от количества сортируемых элементов. Чем меньше это время, тем более эффективен алгоритм. По той же причине интерес представляют такие методы сортировки, которые позволяют экономно использовать оперативную память, т.е. не требуют дополнительных массивов.

Сортировка с помощью выбора. Сортировка с помощью выбора представляет собой один из самых простых методов сортировки. Он предполагает следующую последовательность действий:

- выбирается наименьший элемент из всего массива;
- найденный наименьший элемент меняется местами с первым элементом;
- затем этот процесс повторяется с оставшимися *n*-1 элементами, *n*-2 элементами и т.д. до тех пор, пока не останется один, самый большой элемент массива.

Пример алгоритма сортировки с помощью выбора приведен на рисунке 3.8.

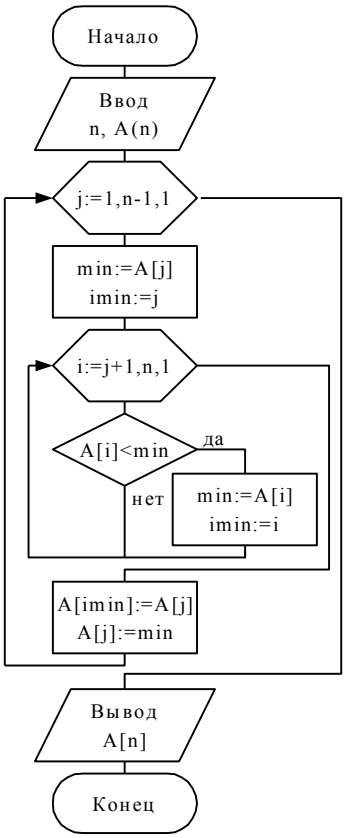


Рисунок 3.8 – Схема алгоритма сортировки выбором

Сортировка с помощью обмена. Алгоритм прямого обмена основывается на сравнении и смене мест для пары соседних элементов. Сравнения продолжают до тех пор, пока не будут упорядочены все элементы. Такой метод широко известен под именем «пузырьковая сортировка», т. к. элементы массива можно рассматривать как пузырьки, которые поднимаются к началу массива.

Пусть, например, массив состоит из элементов (2, 5, 12, 1, 8, 4). Если сортировать массив по возрастанию, то после первого «прохода» по массиву мы получим следующий массив (2, 5, 1, 8, 4, 12). После второго «прохода» - (2, 1, 5, 4, 8, 12), после третьего – (1, 2, 4, 5, 8,12).

Анализ показывает, что сортировку можно прекратить, когда при очередном проходе не понадобилось выполнить ни одной перестановки. Кроме того, следует учесть, что после каждого прохода по меньшей мере один элемент становится на место, поэтому можно каждый раз сокращать количество просматриваемых элементов. Схема алгоритма сортировки с помощью обмена представлена на рисунке 3.9.

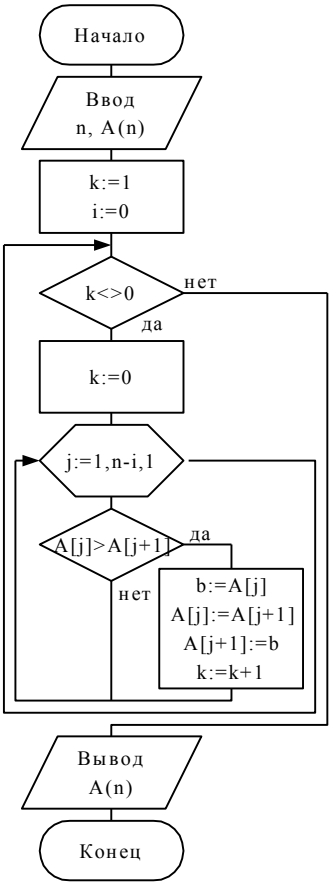


Рисунок 3.9 – Схема алгоритма обменной сортировки

Сортировка методом вставки. Алгоритм сортировки с помощью вставки можно описать следующим образом. При каждом шаге, начиная с i=2, из исходной последовательности извлекается i-й элемент и перекладывается в готовую последовательность на нужное место. В реальном процессе поиска подходящего места удобно сначала сравнивать извлеченный элемент с очередным элементом a_j, а затем либо a_j сдвигается вправо, либо a_i вставляется на освободившееся место. Процесс поиска может закончиться при выполнении одного из двух следующих условий: либо найден элемент a_j меньший, чем a_i, либо достигнут левый конец массива.

Такой типичный случай повторяющегося процесса с двумя условиями окончания позволяет нам воспользоваться хорошо известным приемом «барьера». Для этого необходимо расширить диапазон индекса в описании массива от 0 до n. Алгоритм сортировки приводится на рисунке 3.10.

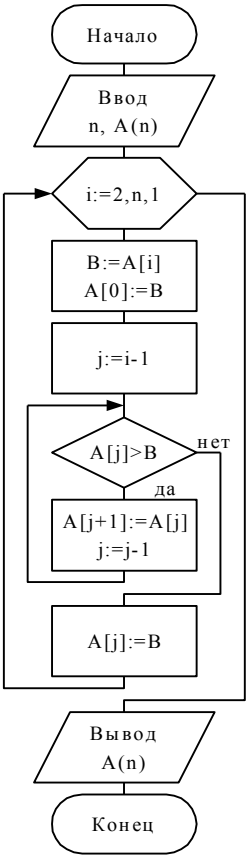


Рисунок 3.10 – Схема алгоритма сортировки вставками

Метод Шелла. Метод состоит в том, что упорядочиваемый массив делится на группы таким образом, что в каждой группе находятся элементы, отстоящие друг от друга на расстоянии d. Каждая группа сортируется методом вставки, затем массив делится на новые группы с шагом d-1 и т.д. пока количество групп не станет равным единице. Обычно d выбирается таким образом, чтобы в каждой группе находилось два элемента, т.е. $d=[n/2]$.

3.1.4 Переформирование массива с изменением его размеров

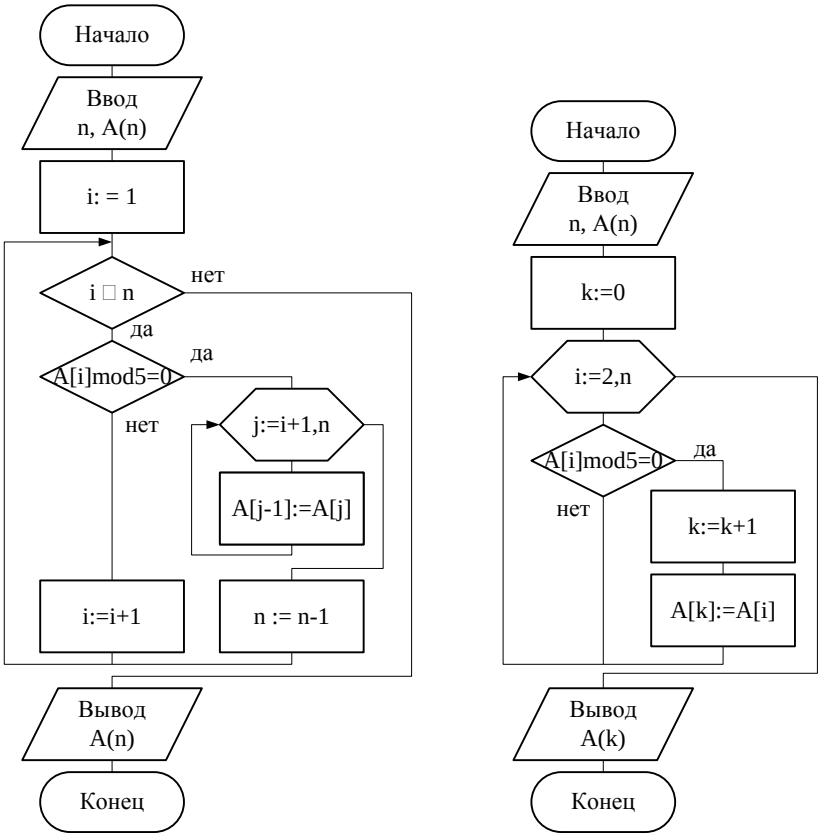
Примерами этого класса задач могут служить: вычеркивание и вставка элементов, отвечающих определенным условиям или обладающих заданными признаками и т. д. При этом, для удаления или добавления элемента может производиться как поэлементная, так и выборочная обработка элементов массива. Особенностью этого класса задач является изменение размеров исходного массива. Кроме того, вычеркивание или вставка элементов требуют сдвига всех элементов той части массива, которая расположена после удаляемого или вставляемого элемента. Реализация алгоритмов решения задач этого типа, как правило, требует вложенных циклов. При этом, внутренний цикл для сдвига можно выполнить, используя счетный цикл. Внешний цикл обхода содержит переменную верхнюю границу, поэтому лучше использовать циклы с пред или пост условием. Если на соответствие условию проверяются все элементы, то параметр внешнего цикла меняется на 1. Однако после сдвига на месте старого анализируемого элемента может оказаться элемент, отвечающий требуемому условию. Поэтому для определения количества повторений цикла сдвигов необходим цикл с пост условием, а не оператор ветвления (используется три вложенных друг в друга цикла). При выборочном анализе элементов массива параметр внешнего цикла меняется на раз-

ную величину, в зависимости от наличия или отсутствия сдвига и для определения выполнять сдвиг или нет, можно использовать оператор условной передачи управления. При большом количестве удаляемых или вставляемых элементов вычислительная сложность таких алгоритмов достаточно велика.

Пример 3.7. Пусть необходимо вычеркнуть из массива целых чисел все числа, кратные 5.

Для решения задачи можно предложить два алгоритма:

- просматривая массив, будем находить очередной удаляемый элемент, а затем путем сдвига переносить все элементы, расположенные после него, затирая удаляемый элемент. Полученный массив окажется на 1 элемент меньше. Изменив размер массива, следует продолжить просмотр элементов (см. рисунок 3.11, *а*);
- просматривая массив, будем переписывать все остающиеся элементы на очередное место. В этом случае потребуется две переменные индекса: в качестве первой будем использовать переменную цикла, а вторую будем менять независимо после записи очередного остающегося элемента (см. рисунок 3.11, *б*).



а *б*

Рисунок 3.11 – Схемы алгоритма решения задачи:
а – с двумя циклами; *б* – с одним циклом

В первом алгоритме внешний цикл не может быть счетным, поскольку в счетном цикле переменная *i* меняется при каждой итерации, а нам не надо ее менять, если *i*-й элемент удаляется, поскольку на его место в результате сдвига переносится следующий, еще не проверенный элемент. Замена цикла на цикл-пока позволяет менять *i* только, когда это необходимо.

Несложно видеть, что вычислительная сложность второго алгоритма меньше, чем первого. Но первый алгоритм легче воспринимается. Практика показыва-

ет, что второй способ практически всегда применим, если элементы из массива удаляются. При добавлении элементов второй способ не применим, поскольку при отсутствии дополнительного массива переписываемые элементы рано или поздно начнут затирать еще не проверенные элементы массива.

Пример 3.8. Предположим, нам необходимо переформировать массив вещественных чисел, добавив в него нулевые элементы после положительных элементов, стоящих на четных местах.

В данной задаче без второго цикла не обойтись. Причем необходимо учесть, что при сдвиге элементов ни один элемент не должен быть затерт, поэтому двигать элементы придется с последнего.

Кроме того, необходимо обеспечить обход элементов, расположенных на четных местах. Для этого параметр цикла обхода необходимо менять на 2 единицы. После обнаружения требуемого элемента следует обеспечить сдвиг всех оставшихся элементов, при этом все четные элементы окажутся на нечетных местах и для правильного добавления необходимо перейти через три элемента, а не через два. Необходимо также посчитать количество добавленных элементов, чтобы в конце распечатать весь массив, и конечно следует предусмотреть свободное место в массиве, куда можно добавлять элементы. Окончательный вариант алгоритма представлен на рисунке 3.12.

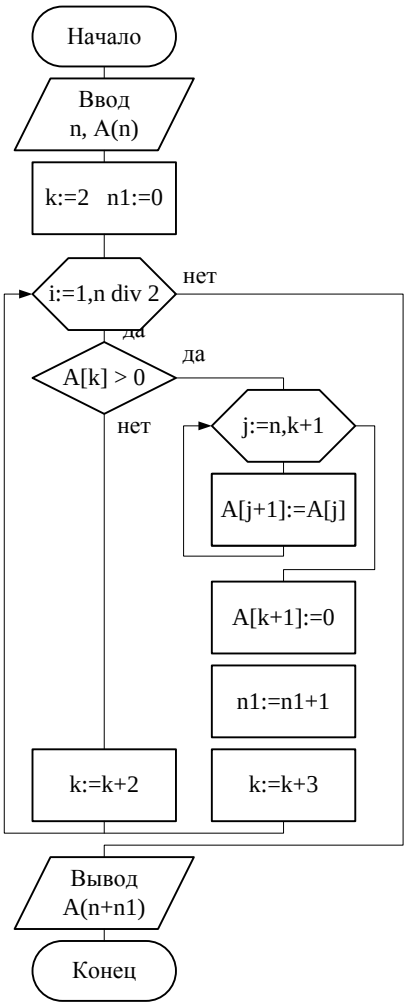


Рисунок 3.12 – Схема алгоритма добавления элементов в массив

3.1.5 Одновременная обработка нескольких массивов или подмассивов

К этому типу относятся задачи по слиянию массивов, переписи элементов одного массива, отвечающих определенному условию или имеющих некоторый признак, в другой, формирование нового массива из элементов исходного, преобразованных по некоторой формуле или подчиняющиеся определенному закону. Особенностью задач этого класса является то, что у каждого массива свой индекс, свой закон и диапазон его изменения. При программировании задач этого класса можно использовать как счетные, так и циклы с пред или пост условием, причем выбор зависит от того каков закон изменения индексов и как легче организовать их изменение.

Пример 3.9. Даны два массива, отсортированные по возрастанию. Объединить их в третий массив, сохраняя сортировку.

В алгоритме (см. рисунок 3.13) для каждого массива используется свой индекс и своя граница индекса.

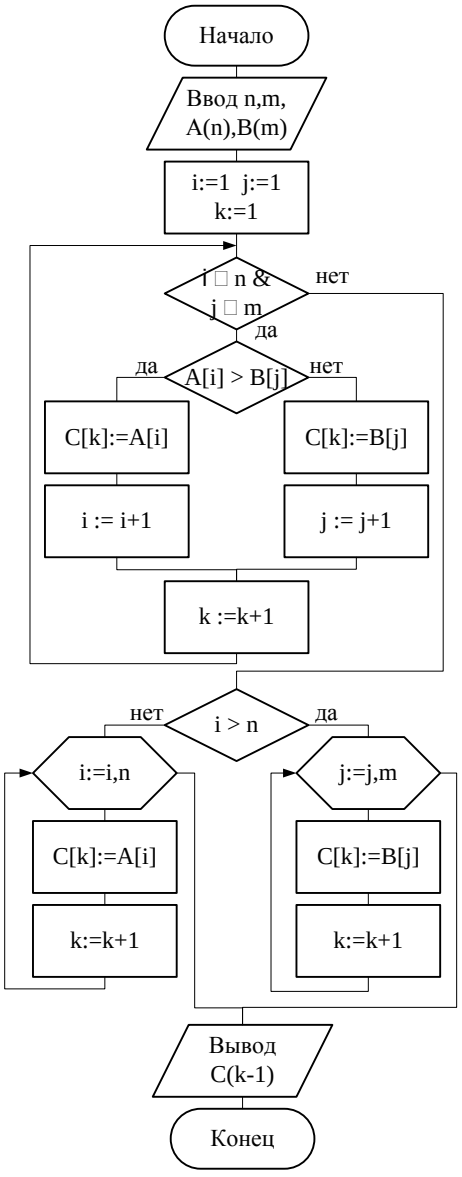


Рисунок 3.13 – Схема алгоритма слияния сортированных массивов

Сначала мы сравниваем элементы массивов и пишем в результирующий массив минимальный элемент из исходных массивов, а затем, когда один из массивов заканчивается – переписываем в массив С оставшиеся элементы второго массива.

3.1.6 Поиск в массиве единственного элемента, отвечающего некоторому условию

Примерами подобного рода задач могут служить поиск первого отрицательного, первого положительного и любого первого элемента, отвечающего некоторому условию, поиск первого или единственного элемента, равного некоторому конкретному значению. Особенность задач этого класса в том, что нет необходимости просматривать весь массив. Просмотр нужно закончить сразу, как только требуемый элемент будет найден. При этом может производиться как поэлементный просмотр, так и выборочная обработка массива. Однако, в худшем случае, для поиска элемента требуется просмотреть весь массив. Такой тип поиска называется линейным. Если массив не очень большой, затраты времени линейного поиска не столь заметны. Но при солидных объемах информации время поиска становится серьезным показателем. Поэтому существуют методы, позволяющие уменьшить время поиска: поиск с барьером и двоичный поиск. Чаще всего при программировании поисковых задач используются циклы с пред или пост условием, в которых условие выхода формируется из двух условий. Одно условие - пока элемент не найден, а второе - пока есть элементы массива. После выхода из цикла осуществляется проверка, по какому из условий произошел выход.

Рассмотрим следующие примеры.

Пример 3.10. Задан целочисленный массив из n элементов. Присвоить переменной *k* значение *true*, если в массиве существуют два одинаковых элемента, стоящих рядом, и *false* в противном случае.

Эту задачу можно решить, например, если последовательно сравнить каждый элемент массива с последующим. Так как рассматриваются все элементы, то одно условие – это достижение конца массива, а другое – обнаружение двух одинаковых соседних элементов. Схему алгоритма для решения этой задачи см. на рисунке 3.14.

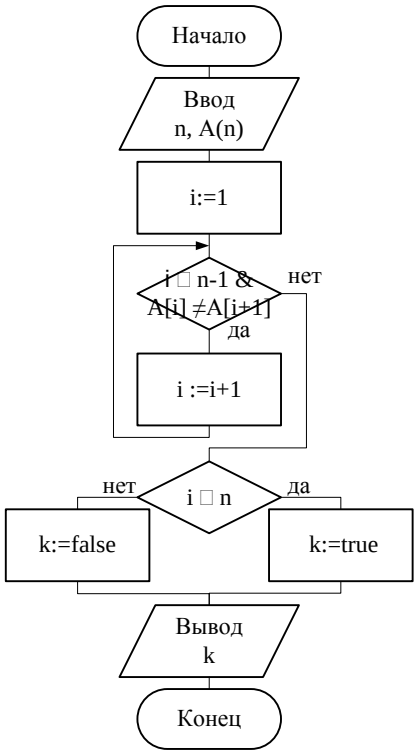


Рисунок 3.14 – Схема алгоритма поиска пары одинаковых элементов в массиве

Пример 3.11. Необходимо определить среди элементов, стоящих на четных местах первый отрицательный элемент.

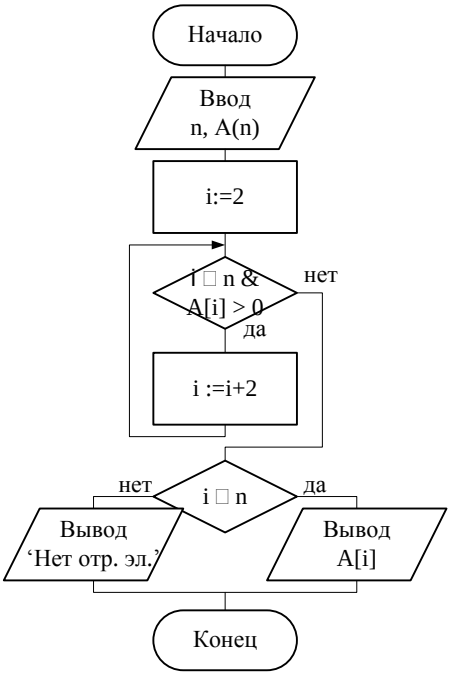


Рисунок 3.15 – Схема алгоритма поиска в массиве первого отрицательного элемента

3.2 Приемы обработки матриц

Двумерными называются массивы, имеющие два типа индексов. По аналогии с математикой, иногда такие массивы называют *матрицами*. Для простоты изложения в дальнейшем будем придерживаться именно этой терминологии.

Рассмотрим наиболее распространенные приемы программирования обработки матриц. Следует отметить, что программирование всех классов задач для матриц имеет свою специфику, основанную на том, что матрица, фактически, является массивом одномерных массивов. Это значит, что в каждом классе имеется гораздо больше различных вариантов решений, да и самих задач тоже. Особенно много сочетаний различных классов задач в одной конкретно поставленной задаче обработки некоторой матрицы.

3.2.1 Последовательная обработка элементов матрицы

Для матриц к данному типу могут рассматриваться не только относительно всего массива в целом, но также только для строк или только для столбцов. В этом случае при реализации программы появляются некоторые особенности. Кроме того, к этой группе могут быть отнесены задачи ввода и вывода матриц. Особенности же программирования первого класса задач для матриц основном те же, что и в разделе 4.1.1. Однако, возможно появление и некоторых новых приемов, связанных со спецификой матриц.

Рассмотрим некоторые типовые алгоритмы задач первого класса.

Пример 3.12. Задана квадратная целочисленная матрица размера $n \times n$. Найти наибольший элемент матрицы и количество таких элементов.

Для решения этой задачи нужно организовать цикл для перебора всех элементов матрицы. Так как матрица является двумерным массивом, то потребуется два цикла (один для перебора строк, второй – столбцов). В остальном, алгоритм решения этой задачи (см. рисунок 3.16) схож с алгоритмом нахождения максимального элемента одномерного массива.

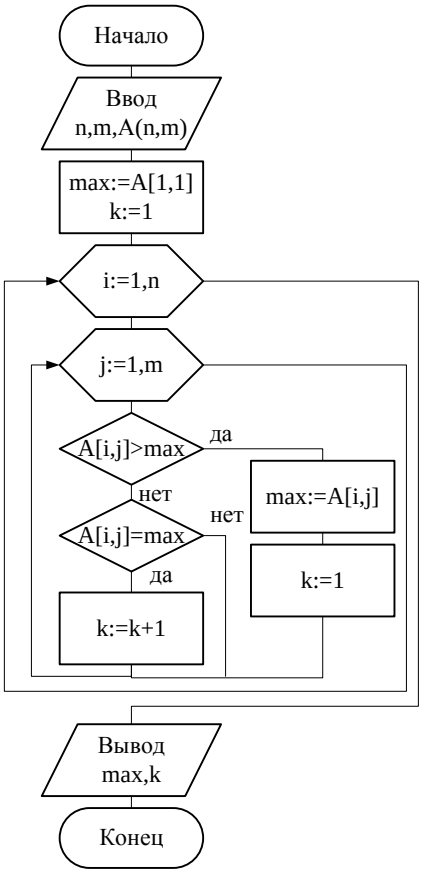


Рисунок 3.16 – Схема алгоритма поиска максимальных элементов матрицы

Пример 3.13. Задана целочисленная матрица $A(m,n)$. Вычислить сумму каждой стоки и результат записать в массив $B(n)$.

Для вычисления суммы элементов некоторой строки с номером i необходимо организовать цикл для перебора всех элементов данной строки. Поэтому параметром этого цикла следует выбрать номер столбца j. Перед циклом с параметром j необходимо задать начальное значение суммы S=0. После окончания цикла результат присваивается соответствующему элементу массива b. Для обеспечения обхода всех строк матрицы параметром внешнего цикла следует выбрать номер строки i (см. рисунок 3.17).

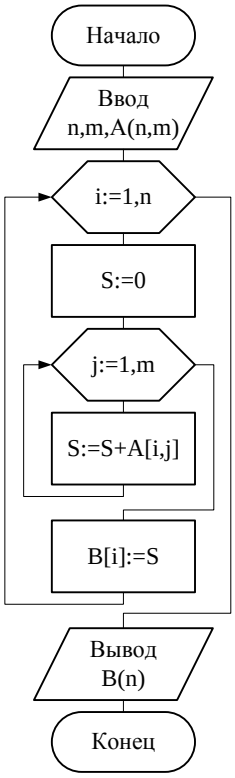


Рисунок 3.17 – Схема алгоритма поиска максимальных элементов матрицы

3.2.2 Изменение порядка следования элементов без изменения размеров исходной матрицы.

В качестве примеров задач данного типа для матриц рассмотрим задачу перемещения элементов.

Пример 3.14. Дана действительная квадратная матрица порядка $2n$. Получить новую матрицу, переставляя ее блоки размера $n \times n$ в соответствии с рисунком.

Для решения этой задачи нужно определить правило, по которому будут преобразовываться индексы матрицы. Будем считать, что после преобразования получается новая квадратная матрица размерности $2n$. Тогда внимательно изучив соответствие между элементами матрицы A и матрицы B, получаем следующий закон преобразования:

$$B(i,j)=\begin{cases} A(i+n,j+n), & \text{если} \quad i\le n, j\le n;\\ A(i+n,j-n), & \text{если} \quad i\le n, j\ge n;\\ A(i-n,j+n), & \text{если} \quad i\ge n, j\le n;\\ A(i-n,j-n), & \text{если} \quad i\ge n, j\ge n. \end{cases}$$

Схема алгоритма решения представлена на рисунке 3.18.

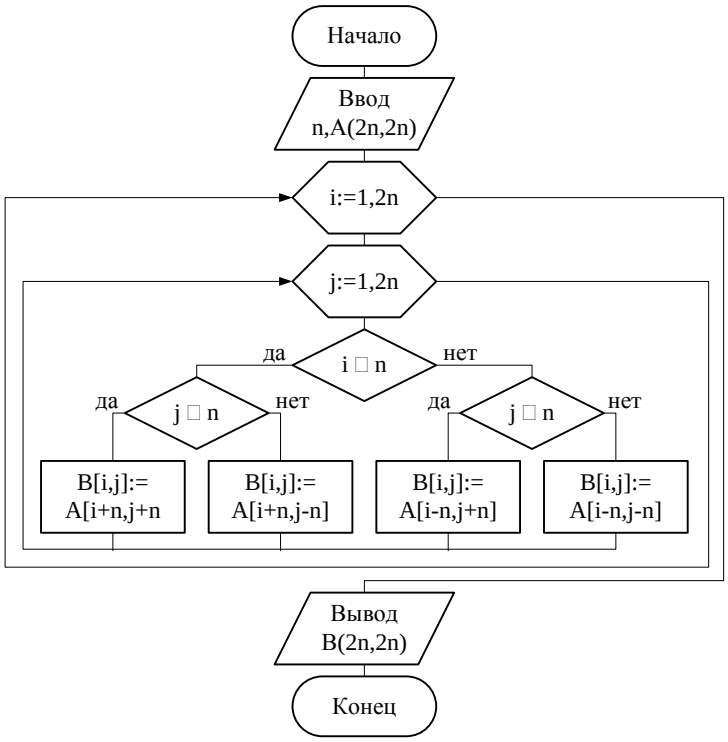


Рисунок 3.18 – Схема алгоритма перестановки элементов матрицы

При решении задач с матрицами прочих типов также используются приемы, применяемые при обработке одномерных массивов, поэтому отдельно они в настоящем пособии рассматриваться не будут.

Литература

1. Г.С. Иванова. Программирование. Учебник для ВУЗов. – М.: Кнорус, 2013.