Московский государственный технический
университет
имени Н.Э. Баумана

Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

Т.Н. Ничушкина, В.В. Гуренко

РАЗРАБОТКА АЛГОРИТМОВ ПРОСТЕЙШИХ ПРОГРАММ

Электронное учебное издание

Учебное пособие
по выполнению лабораторных работ и домашних заданий
по курсу «Основы программирования»

Москва
(С) 2014 МГТУ им. Н.Э. БАУМАНА

УДК 004.021

Рецензенты: проф., д.т.н. Ирина Васильевна Сергеева,
доц., к.т.н. Наталия Владимировна Новик

Ничушкина Т.Н., Гуренко В.В.
Разработка алгоритмов простейших программ. Электронное учебное издание. – М.: МГТУ имени Н.Э. Баумана, 2014. – 47 с.: ил.

В учебном издании представлены основные приемы разработки алгоритмов программ простой структуры. Рассмотрены особенности решения наиболее распространенных задач вычислительной математики и приведены соответствующие алгоритмы. Описаны приемы обработки массивов, проиллюстрированные наиболее интересными примерами алгоритмов.

Издание предназначено для студентов кафедры ИУ-6 «Компьютерные системы и сети» МГТУ им. Н.Э. Баумана и студентов, обучающихся по аналогичной программе на Аэрокосмическом факультете университета (АК-5). Может быть полезно студентам других профилей, изучающим программирование в соответствии с учебным планом направления подготовки 230100 «Информатика и вычислительная техника».

Рекомендовано Учебно-методической комиссией НУК «Информатика и системы управления» МГТУ им. Н.Э. Баумана

Ничушкина Татьяна Николаевна
Гуренко Владимир Викторович

РАЗРАБОТКА АЛГОРИТМОВ ПРОСТЕЙШИХ ПРОГРАММ

© МГТУ им. Н.Э. Баумана, 2014

Оглавление

ОГЛАВЛЕНИЕ.....1

ВВЕДЕНИЕ.....3

ГЛАВА 1. РАЗВЕТВЛЯЮЩИЕСЯ ПРОЦЕССЫ.....4

Контрольные вопросы.....5

ГЛАВА 2. ЦИКЛИЧЕСКИЕ ПРОЦЕССЫ. АЛГОРИТМЫ РЕШЕНИЯ ЗАДАЧ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ.....7

2.1. Типы циклических процессов.....7

2.2. Табулирование функции.....8

2.3. Нахождение суммы ряда.....8

2.4. Приближенное вычисление определенных интегралов.....9

2.4.1. Метод прямоугольников.....9

2.4.2. Метод трапеций.....9

2.5. Определение корней уравнения.....10

2.5.1. Метод половинного деления.....10

2.5.2. Метод хорд.....10

2.6. Нахождение длины кривой.....10

Контрольные вопросы.....11

ГЛАВА 3. МАССИВЫ.....12

3.1. Приемы обработки одномерных массивов.....12

3.1.1. Последовательная обработка всех элементов массива.....12

3.1.2. Выборочная обработка элементов массива.....13

3.1.3. Изменение порядка следования элементов массива. Сортировка.....13

3.1.4. Переформирование массива с изменением его размера.....14

3.1.5. Одновременная обработка нескольких массивов или подмассивов.....15

3.1.6. Поиск в массиве элемента, отвечающего заданному условию.....16

3.2. Приемы обработки матриц.....16

3.2.1. Последовательная обработка элементов матрицы.....16

3.2.2. Изменение порядка следования элементов матрицы.....17

Контрольные вопросы.....17

ЛИТЕРАТУРА.....19

Настоящее учебное пособие предназначено для студентов 1 курса, изучающих дисциплину «Основы программирования» на кафедре ИУ-6 МГТУ им. Н.Э. Баумана в соответствии с программой подготовки бакалавров техники и технологии направления «Информатика и вычислительная техника». Основная задача, которую авторы ставили перед собой, – помочь студентам в освоении базовых приемов алгоритмизации задач, которые непосредственно востребуются при выполнении лабораторных работ и домашних заданий по названной учебной дисциплине. Глава 1 посвящена разветвляющимся вычислительным процессам. В главе 2 рассмотрены циклические процессы на примере наиболее известных и применимых на практике задач вычислительной математики. В главе 3 показаны и пояснены основные приемы обработки одномерных массивов и матриц.

Введение

Изящное и технически грамотное написание компьютерных программ предполагает не только хорошее владение средствами их разработки, но и достаточно развитое алгоритмическое мышление. Сложившаяся на протяжении десятилетий практика обучения программированию говорит о том, что именно недостаток алгоритмического мышления – основная причина неудач студентов в процессе освоения программирования.

Алгоритмическое мышление как умение выстраивать логически безупречную последовательность действий на пути к решению задачи, можно и нужно развивать. Опыт показывает, что одним из наиболее действенных способов достижения этого является проработка алгоритмов ряда небольших, но полезных программ, приводящая к формированию базы приемов программирования. Таких приемов сравнительно немного, но их накопление, обобщение и умение осознанно применять на практике позволяет студентам научиться писать программы.

В настоящем учебном пособии представлены базовые алгоритмы, без изучения которых знания, умения и навыки в области программирования не будут отличаться полнотой. Авторы сознательно не приводят тексты программ и ограничиваются схематическими представлениями алгоритмов: тексты программ, изобилующие деталями конкретных средств программирования, неизбежно отвлекали бы внимание от самих алгоритмов, что было бы крайне нежелательно.

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

Таблица 1 – Обозначения алгоритмических блоков

Название блока	**Обозначение**	**Назначение блока**
1	2	3
1. Терминатор	Действие	Начало, завершение программы или подпрограммы
2. Процесс	Действие	Обработка данных (вычисления, пересылки и т.п.)
3. Данные	Данные	Операции ввода-вывода

Таблица 1 – продолжение

1	2	3
4. Решение	Условие	Ветвления, выбор, итерационные и поисковые циклы
5. Подготовка	Действия	Счетные циклы
6. Граница цикла	Начало Конец	Любые циклы
7. Предопределенный процесс	Имя	Вызов процедур
8. Соединитель	Имя	Маркировка разрывов линий
9. Комментарий	Комментарий	Пояснения к операциям

Применение схем для изображения алгоритмов, во-первых, позволяет достаточно формально их представлять и, во-вторых, дает более наглядное визуальное восприятие. Использование псевдокодов не приводит к требуемому результату – формированию у обучающихся необходимой базы, позволяющей самостоятельно разрабатывать алгоритмы.

ГЛАВА 1. РАЗВЕТВЛЯЮЩИЕСЯ ПРОЦЕССЫ

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

В качестве примера разветвляющегося вычислительного процесса рассмотрим алгоритм нахождения 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 вовсе не имеет действительных корней. Поэтому 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.

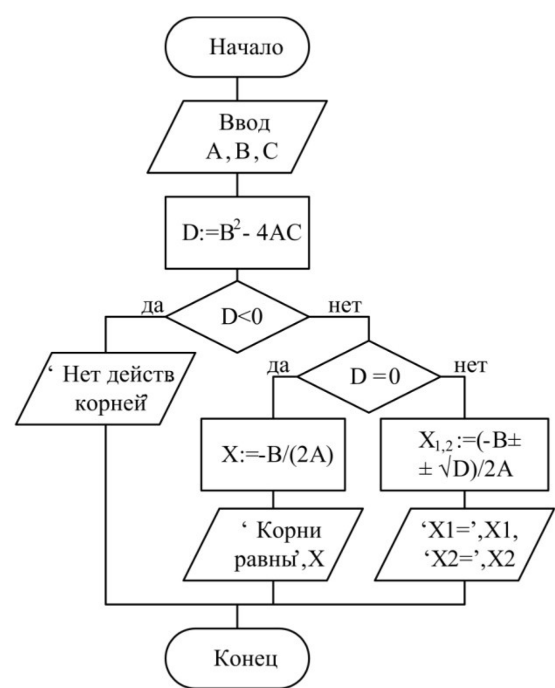


Рисунок 1.1 – Схема алгоритма нахождения действительных корней квадратного уравнения

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

Таблицы решений. Таблица решений составляется 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 $ax^2 + bx + c = 0$ можно составить следующую таблицу:

	Корней нет	$x = -b / 2a$	$x = (-b \pm \sqrt{D}) / 2a$
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 его выполнения указать оставшиеся два случая. В итоге получаем алгоритм, схема которого приведена выше.

Иногда составленная таблица решений приобретает 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 довольно громоздкого алгоритма, приведенная на рисунке 1.2.

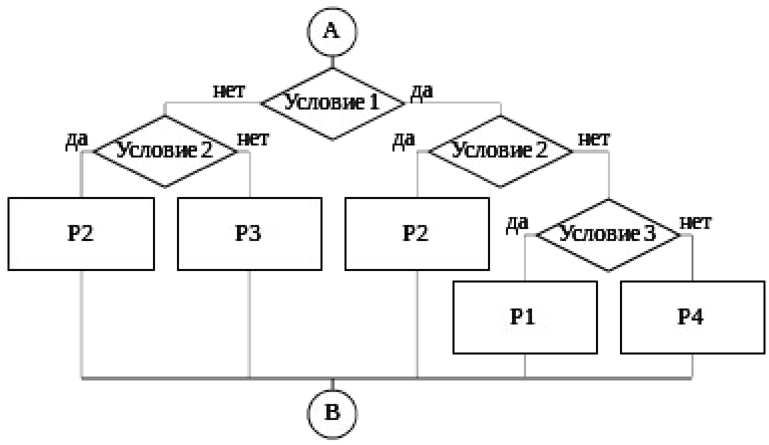


Рисунок 1.2 – Фрагмент схемы неоптимального алгоритма

Такой алгоритм нельзя считать приемлемым. Однако его можно существенно упростить, если в таблице поменять местами проверяемые условия. Кроме того, для удобства построения алгоритма целесообразно также 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 оптимизированной таблице, окажется значительно проще, и его построение потребует меньших усилий (см. рисунок 1.3).

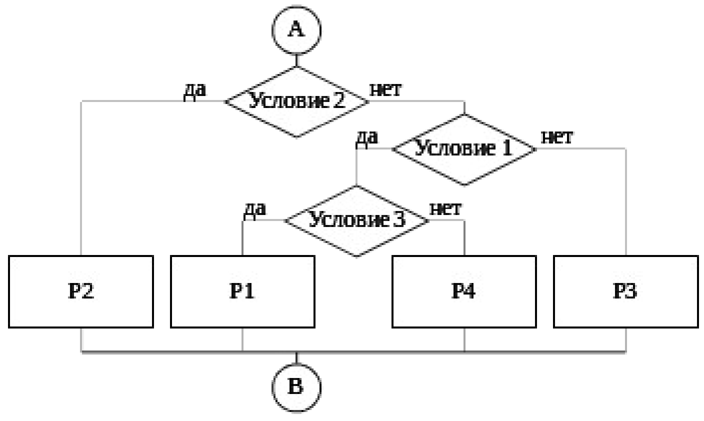


Рисунок 1.3 – Фрагмент схемы оптимизированного алгоритма

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 необходимо рассмотреть следующие 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 условия, которые следует 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, а значит, 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 получим следующую оптимизированную таблицу:

	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, можно построить алгоритм, в котором достаточно сделать всего четыре проверки условий (см. рисунок 1.4).

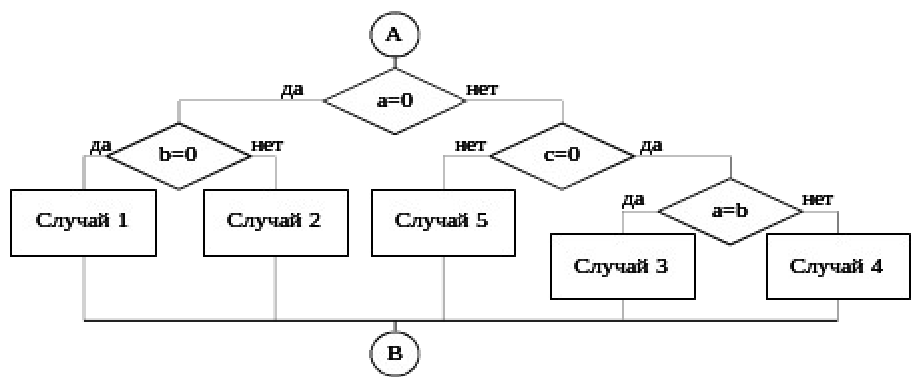


Рисунок 1.4 – Фрагмент схемы алгоритма решения системы уравнений

Контрольные вопросы

1. Какой вычислительный процесс называют разветвляющимся?
2. Что такое таблица решений и какова ее структура?
3. В каких случаях следует применять таблицы решений?
4. Каков принцип построения алгоритма по таблице решений?
5. Как и с какой целью можно оптимизировать таблицу решений?

ГЛАВА 2. ЦИКЛИЧЕСКИЕ ПРОЦЕССЫ. АЛГОРИТМЫ РЕШЕНИЯ ЗАДАЧ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ

2.1. Типы циклических процессов

При решении задач нередко требуется многократно повторить одну и ту же последовательность действий, но над различными значениями переменных. Такие вычислительные процессы называют *циклическими*, а многократно повторяющиеся участки алгоритмов – *циклами*.

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

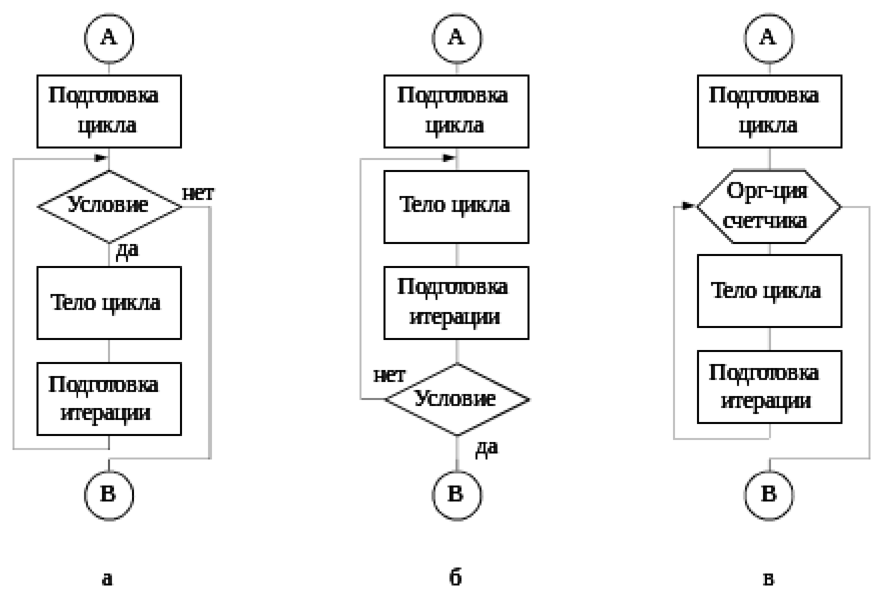


Рисунок 2.1 – Структурные конструкции циклов:

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 очередного 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 включающую ветвления и другие циклы, которые называют *вложенными циклами*. При алгоритмизации задач с использованием вложенных циклов необходимо соблюдать 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 неизвестным числом повторений – так называемые *итерационные циклы.* Последние характеризуются постепенным приближением к требуемому значению с заданной точностью или достижением некоторого другого условия.

В конкретных случаях циклических процессов указанные выше 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.

Рассмотрим пример.

Пример 2.1. Задано действительное число $a>1$. Получить все члены последовательности a, a^2, a^3, …, не превышающие действительного числа b.

Из условия задачи ясно, что исходными данными являются числа a и b, которые, например, вводятся с клавиатуры. Именно с их помощью будет формироваться нужное число степеней a. Степени можно получить последовательным умножением a на само себя. Результат умножения будем хранить во вспомогательной переменной c. Эти действия и являются телом цикла. Так как $a^0=1$, то в блоке подготовки цикла переменной c устанавливаем начальное значение, равное 1. Условие выхода из цикла – это превышение числа b числом c.

Схема алгоритма решения задачи представлена на рисунке 2.2.

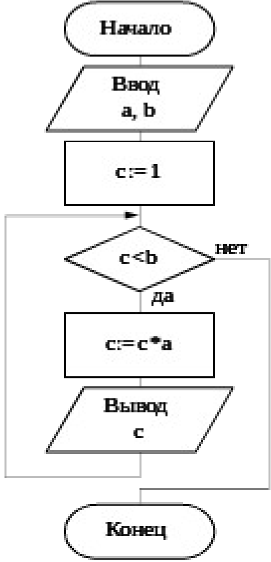


Рисунок 2.2 – Схема алгоритма получения последовательности чисел

В теле цикла последовательно вычисляется переменная *c*, которая принимает значения *a*, a^2, a^3, … Процесс вычисления продолжается до тех пор, пока *c* не станет большим или равным значению *b*. Если $a \geq b$, то будет выведено только само число *a*.

2.2. Табулирование функции

Типичным примером циклического процесса с заданным числом повторений является задача *табулирования функции*, то есть получения таблицы ее значений.

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

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

где скобки [ ] означают целую часть от деления. Для решения задачи можно предложить алгоритм, схема которого приведена на рисунке 2.3.

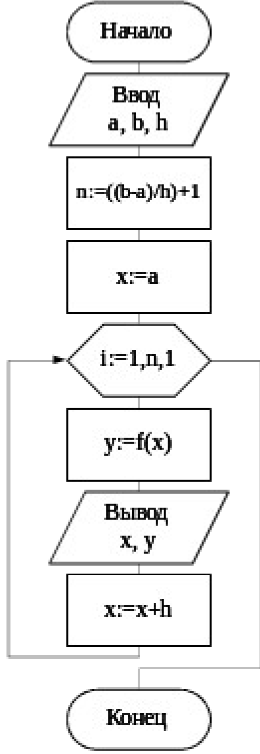


Рисунок 2.3 – Схема алгоритма табулирования функции

При подготовке цикла определяется число его повторений *n* и присваивается начальное значение переменной *x*. В теле цикла рассчитывается значение функции в точке *x* и выводятся значения *x* и *y*. При подготовке итерации изменяется значение *x*, получающее приращение на величину *h*.

2.3. Нахождение суммы ряда

Многие задачи вычислительной математики сводятся к вычислению суммы ряда вида

$$S = \sum_{i=1}^{\infty} r_i \,.$$

Определение суммы предполагает последовательное вычисление членов ряда r_i и каждый раз прибавление очередного вычисленного значения к частичной сумме *S*. Процесс продолжается, пока не будет выполнено условие выхода из цикла. Общий вид схемы алгоритма решения такой задачи приведен на рисунке 2.4. Условие завершения цикла зависит от поставленной задачи. Как правило, оно связано с достижением требуемой точности.

Примеры постановки задачи.

1. Найти сумму первых n членов ряда $S = \sum_{i=1}^{\infty} r_i$. В этом случае условием выхода из цикла будет выполнение неравенства $i > n$.
2. Вычислять сумму ряда $S = \sum_{i=1}^{\infty} r_i$ до тех пор, пока не будет достигнут очередной член ряда r_i, меньший заданного ε. Здесь условие выхода из цикла следующее: $|r_i| < \varepsilon$.
3. Вычислить сумму ряда $S = \sum_{i=1}^{\infty} r_i$ с точностью ε, если известно точное значение суммы, равное A. Условие выхода из цикла: $|S - A| < \varepsilon$.

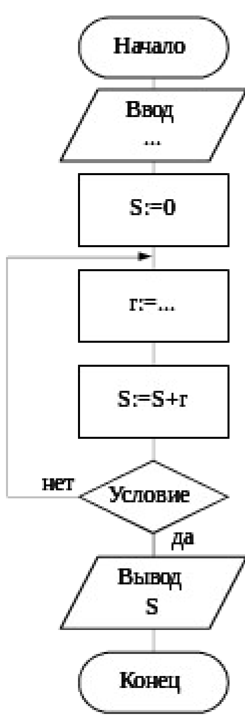


Рисунок 2.4 – Обобщенная схема алгоритма вычисление суммы ряда

Очередной член ряда вычисляют одним из следующих способов:

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

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

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

- по рекуррентным формулам через уже вычисленное значение предыдущего члена ряда, если в формулу общего члена входят, к примеру, целые степени и факториалы. При этом

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

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

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

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

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

2.4. Приближенное вычисление определенных интегралов

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

2.4.1. Метод прямоугольников

Разобьем отрезок [a,b] точками $a=x_0 < x_1 < x_2 <\ldots < x_{n-1} < x_n=b$, 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).

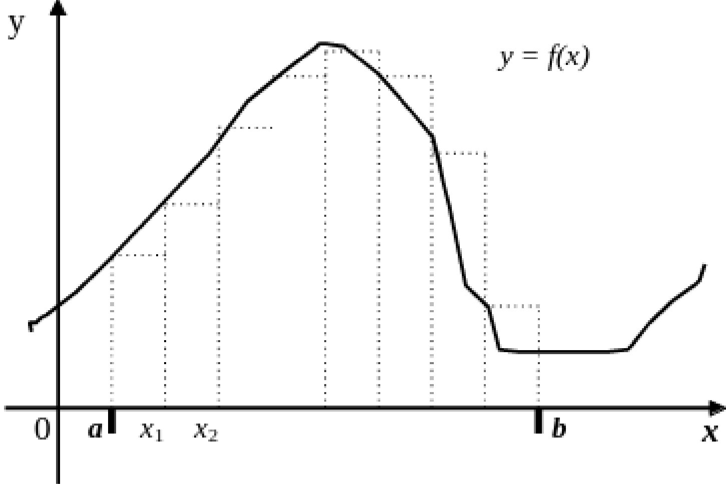


Рисунок 2.5 – Метод прямоугольников

Площадь каждого такого прямоугольника вычисляется по формуле: $S_i=f(x_i)(b\text{-}a)/n$. 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, 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 как в приведенном алгоритме), и вычисления площади повторяются вновь.

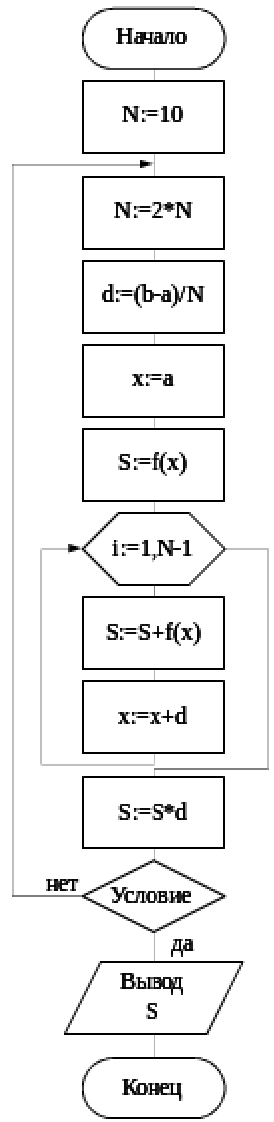


Рисунок 2.6 – Схема алгоритма вычисления интеграла методом прямоугольников

Условие точности, обозначенное на схеме алгоритма блоком решения "Условие", зависит от поставленной задачи. Будем пользоваться двумя способами оценки погрешности:

1) если задано точное значение интеграла, равное I, то условием выхода из цикла будет выполнение неравенства $| S - I | < \varepsilon$;

2) если точного значения интеграла не задано, 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. Отличие состоит в том, что криволинейная трапеция заменяется не прямоугольниками, а прямоугольными трапециями (см. рисунок 2.7).

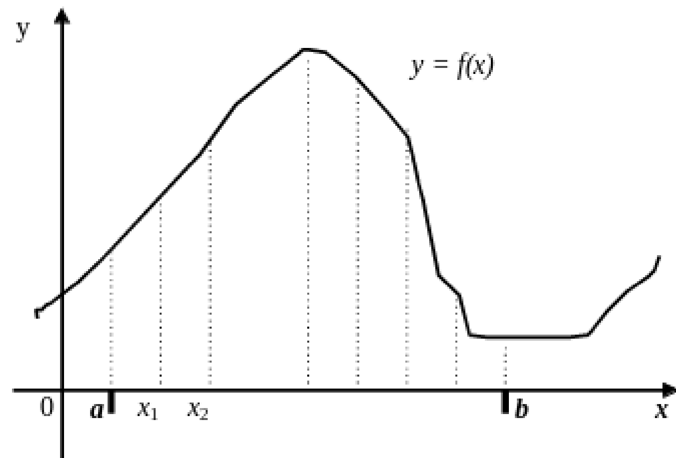


Рисунок 2.7 – Метод трапеций

Площадь i-той прямоугольной трапеции вычисляется следующим образом:

$$S_i = \frac{1}{2}\left(f(x_i) + f(x_{i+1})\right)\frac{(b-a)}{n}$$

Тогда формула приближенного вычисления определенного интеграла методом трапеций принимает 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

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

Схема алгоритма разрабатывается аналогично предыдущему методу.

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 данном отрезке непрерывна и на его концах принимает значения противоположных знаков (т.е. *f(a)f(b)*<0). Излагаемые ниже методы

предусматривают некоторую последовательность действий, приводящую к нахождению нового интервала [a_i, b_i] такого, что

$$a \le a_i < x_0 < b_i \le b,$$

где x_0 – корень уравнения, i – номер итерации. Очевидно, условие существования корня сохраняется: $f(a_i)f(b_i)<0$. Сокращение интервала [a_i, b_i] продолжается до достижения допустимой погрешности ε: $|f(x_i)|< \varepsilon$, где $x_i \in [a_i, b_i]$ вычисляется способом, реализованным в конкретном методе. За корень уравнения на отрезке [a, b] принимается абсцисса x_i: $x_i \approx x_0$ с точностью ε.

2.5.1. Метод половинного деления

Метод является одним из самых простых. Приближение к корню уравнения осуществляется путем нахождения средней точки *c* отрезка [*a*, *b*], т.е. деления его пополам: |a c| = |c b| (см. рисунок 2.8).

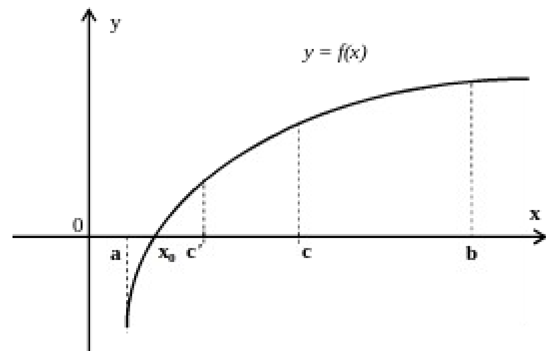


Рисунок 2.8 – Метод половинного деления

Далее проверяется, является ли точка ***c*** корнем уравнения $f(x)=0$. В случае положительного ответа задача решена. В случае отрицательного необходимо сравнить знак функции в точке *c* со знаками функции на концах отрезка и далее рассматривать тот из отрезков [*a*, *c*] и [*c*, *b*], на концах которого функция принимает значения разных знаков. Новый отрезок вновь делится пополам, и все действия повторяются. Таким образом, роль абсциссы x_i, являющейся приближением к корню уравнения на текущей итерации, играет координата с, которая вычисляется заново на каждой итерации алгоритма и становится левой или правой границей сокращаемого интервала. В процессе сокращения длины рабочего отрезка искомый корень уравнения остается внутри него. Вычисления прекращаются, когда будет достигнута заданная точность вычислений ε, т.е. $|f(c)|< \varepsilon$.

Оглавление

2.5.2. Метод хорд

Метод хорд аналогичен методу половинного деления, но обладает лучшей сходимостью, т.е. дает более быстрое приближение к корню уравнения. Новая абсцисса, являющаяся приближением к корню на текущей итерации, находится следующим образом. Точки с координатами (*a*, *f*(*a*)) и (*b*, *f*(*b*)) соединяются отрезком (хордой). Абсцисса ***c*** точки пересечения полученной хорды с осью Ox дает очередное значение корня (см. рисунок 2.9.)

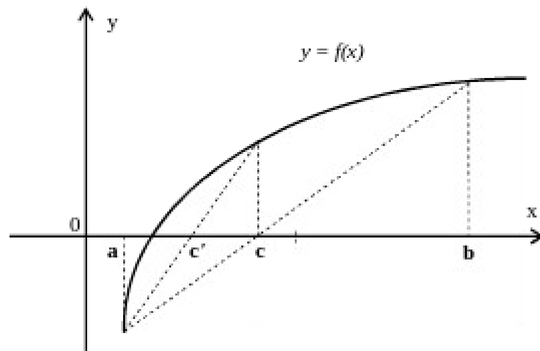


Рисунок 2.9 – Метод хорд

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

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

2.6. Нахождение длины кривой

Пусть функция f(x) непрерывна на отрезке [*a*, *b*]. Требуется определить длину кривой, заданной уравнением *y*=*f*(*x*), на этом отрезке. Задачу можно решить, если аппроксимировать кривую на данном отрезке ломаной, состоящей из n звеньев, и принять за длину кривой длину ломаной (см. рисунок 2.10).

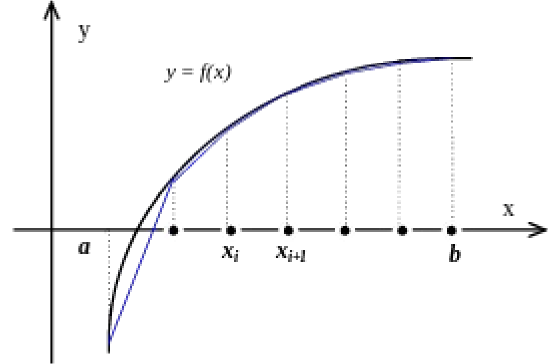


Рисунок 2.10 – Метод хорд нахождения длины кривой

Для удобства алгоритмизации следует предусмотреть, чтобы проекция каждого звена на ось Ox имела одну и ту же длину. Таким образом, отрезок [*a*, *b*] окажется разбит на n отрезков равной длины, вычисляемой как (b-a)/n, – так же, как в численных методах интегрирования. Найти длину *i*-го звена ломаной не представляет труда – это длина отрезка с координатами $(x_i, f(x_i))$ и $(x_{i+1}, f(x_{i+1}))$, где $x_{i+1} = x_i + (b-a)/n$, $x_i < b$.

Начать разбиение отрезка можно с n=2, на следующей итерации n удвоить и т.д. Очевидно, чем больше звеньев ломанной, тем ее длина ближе к искомой длине кривой. Вычисления необходимо продолжать до достижения требуемой точности ε, определяемой, например, как разность между длинами ломаной, полученными на текущей и предыдущей итерациях.

Контрольные вопросы

1. Какой вычислительный процесс называют циклическим?
2. Какие типы циклов вы знаете?
3. Какие основные этапы присутствуют в любом циклическом процессе?
4. Каков порядок проектирования циклического процесса?
5. Какой цикл называют вложенным? Какое правило соблюдается при построении вложенных циклов?
6. Каковы основные особенности алгоритма табулирования функции?
7. Какие можно привести примеры постановки задачи вычисления суммы ряда? Что меняется в схеме алгоритма зависимости от постановки задачи?

Оглавление

8. Чем отличаются численные методы прямоугольников и трапеций для вычисления определенных интегралов? Есть ли между этими методами существенное алгоритмическое различие?
9. Каковы алгоритмические особенности численных методов отыскания корня уравнения на заданном отрезке? В чем их сходство и различие?
10. Каковы особенности алгоритма определения длины кривой на заданном отрезке? С какими из ранее изученных алгоритмов данный метод имеет наибольшее сходство?

ГЛАВА 3. МАССИВЫ

Массив – это упорядоченная совокупность однотипных данных, называемых *элементами* массива. Порядок следования элементов в массиве задают так называемые *индексы*. Каждому элементу сопоставлен свой индекс в случае одномерных массивов или уникальный набор индексов в случае многомерных массивов. В большинстве случаев в качестве индексов используют конечный ряд целых чисел.

Обработка массива сводится к выполнению последовательности действий над его элементами. Для обращения к конкретному элементу массива необходимо указать имя массива и значение индекса (индексов) элемента. Можно выделить следующие классы задач обработки массивов:

1) последовательная обработка всех 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 элемента, отвечающего заданному условию, причем это может быть поиск единственного элемента, либо первого по порядку обработки элемента, либо всех элементов массива, отвечающих условию.

Каждому из перечисленных шести классов задач соответствует своя совокупность приемов программирования. В реальной практике тот или иной класс задач в чистом виде встречается редко. Однако при помощи метода пошаговой детализации любая сложная задача может быть разбита на более простые задачи, относящиеся к указанным выше классам. Следует также учитывать, что существуют особенности решения каждого из названных классов задач в зависимости от количества индексов массива.

3.1. Приемы обработки одномерных массивов

Одномерными называют массивы, для доступа к элементам которых необходим один индекс. Рассмотрим наиболее распространенные приемы обработки одномерных массивов, используемые в программировании.

3.1.1. Последовательная обработка всех элементов массива

Примерами подобного класса задач служат: нахождение суммы элементов, произведения элементов, их среднего арифметического и среднего геометрического, подсчет количества элементов, отвечающих определенному условию или обладающих некоторыми признаками, а также вычисление 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 типов циклов, менее эффективных в рассматриваемом случае.

Рассмотрим некоторые типовые алгоритмы задач.

Пример 3.1. Пусть необходимо ввести (вывести) массив. Для этого нужно последовательно перебрать все элементы массива. Как отмечено, при известном количестве вводимых элементов для этой операции удобно использовать счетный цикл. При неизвестном количестве элементов рациональнее использовать циклы с пред- или постусловиями. При выводе массива 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), представлена на рисунке 3.1, *а*. Заметим, что при решении реальных задач бывает полезно после ввода элементов массива перед началом обработки сразу вывести его, например, для удобства отладки. С этой целью можно использовать тот же цикл, что и для ввода массива, разместив в теле цикла как оператор ввода, так и оператор вывода элемента.

Поскольку ввод-вывод массивов всегда осуществляется однотипно, каждый раз детализировать эти операции нет необходимости. На схемах алгоритмов операции ввода и вывода массивов обычно показывают одним блоком (см. рисунок 3.1, *б*).

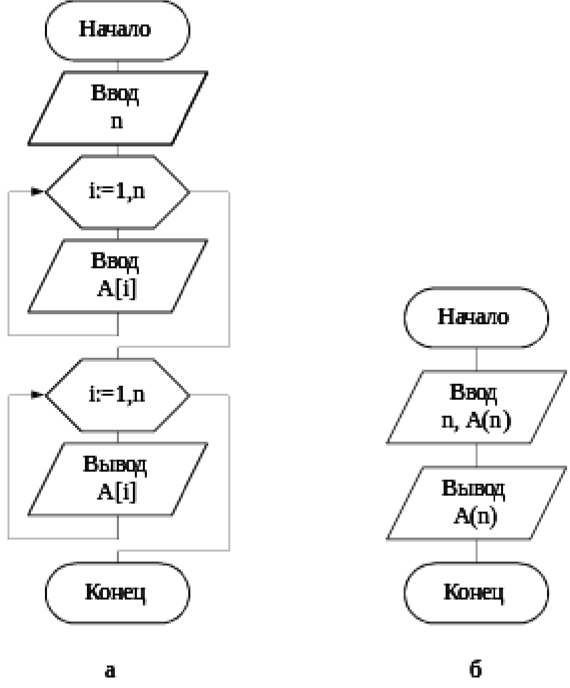


Рисунок 3.1 – Ввод-вывод одномерного массива

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 меньше, чем очередной элемент, то *Amax* присваивается новое значение (см. рисунок 3.2).

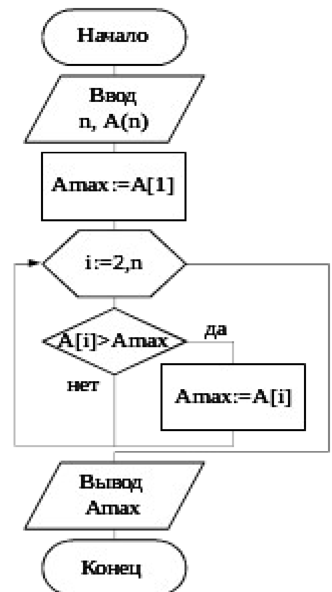


Рисунок 3.2 – Схема алгоритма нахождения максимального элемента массива

Оглавление

Пример 3.3. Пусть необходимо найти среднее арифметическое значение элементов целочисленного массива, кратных трем. Для решения этой задачи водятся две вспомогательные переменные – *Sum* и *Kol*, которые будут использоваться для накопления суммы требуемых элементов и их количества. Первоначально обеим переменным присваиваются нулевые значения. Алгоритм осуществляет полный перебор всех элементов и, если очередной удовлетворяет условию, его значение добавляется к переменной *Sum*, а значение переменной *Kol* увеличивается на единицу. После просмотра массива среднее арифметическое определяется делением накопленной суммы на количество найденных элементов. Схема алгоритма решения задачи приведена на рисунке 3.3.

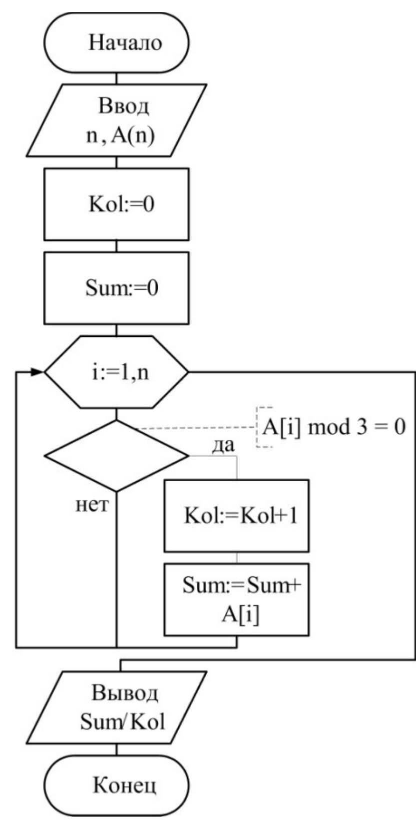


Рисунок 3.3 – Схема алгоритма нахождения среднего арифметического элементов, кратных 3

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

Несложно понять, что схема алгоритма решения этой задачи (см. рисунок 3.4) очень похожа на предыдущую, поскольку вновь требуется последовательная проверка всех элементов массива.

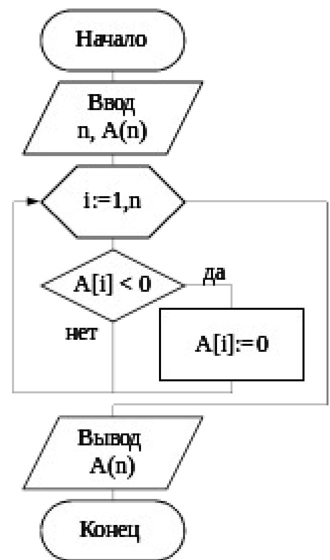


Рисунок 3.4 – Схема алгоритма замены отрицательных элементов массива нулями

3.1.2. Выборочная обработка элементов массива

К задачам данного типа относятся задачи, по формулировке схожие задачам первого класса, но обработка ведется не всех элементов массива, а только тех, которые имеют вполне определенное значение индексов. Таким образом, основной особенностью задач этого класса 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 выполнение. В зависимости от закона изменения индексов могут использоваться любые виды циклов, а также их сочетания.

Пример 3.5. В заданном целочисленном массиве определить максимальный среди элементов, стоящих на четных местах.

Очевидно, просматривать весь массив нет необходимости. Начинать обработку его элементов надо со второго, а индекс увеличивать на два после каждой проверки. Количество анализируемых элементов при этом уменьшается вдвое. Схема алгоритма решения задачи представлена на рисунке 3.5. Возможен вариант решения этой задачи без использования вспомогательной переменной k, когда в качестве индекса используется непосредственно параметр счетного цикла, который изменяется от двух с шагом два.

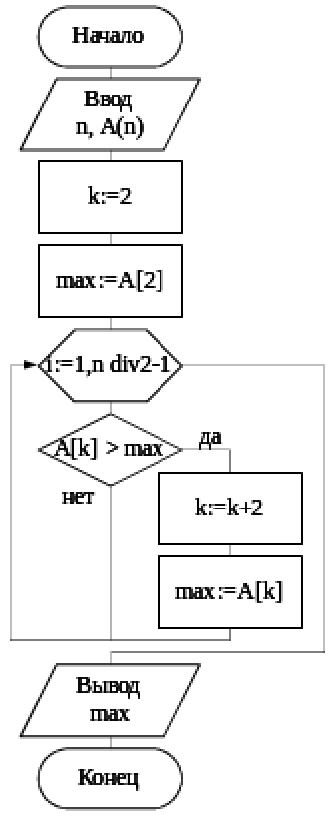


Рисунок 3.5 – Схема алгоритма поиска максимального элемента, записанного на четном месте

Пример 3.6. Заменить каждый третий элемент массива на ноль, если элемент четный, и остатком от деления элемента на пять, если он нечетный.

В данном случае нас интересует каждый третий элемент, поэтому количество просматриваемых элементов равно трети от числа элементов массива, а значение индекса k в цикле увеличивается на три (см. рисунок 3.6). Замечание относительно возможного использования переменной цикла непосредственно в качестве индекса аналогично предыдущему примеру, начальное значение равно трем, шаг изменения – плюс три.

В задачах данного класса изменяется только порядок следования элементов массива. Размер массива остается неизменным.

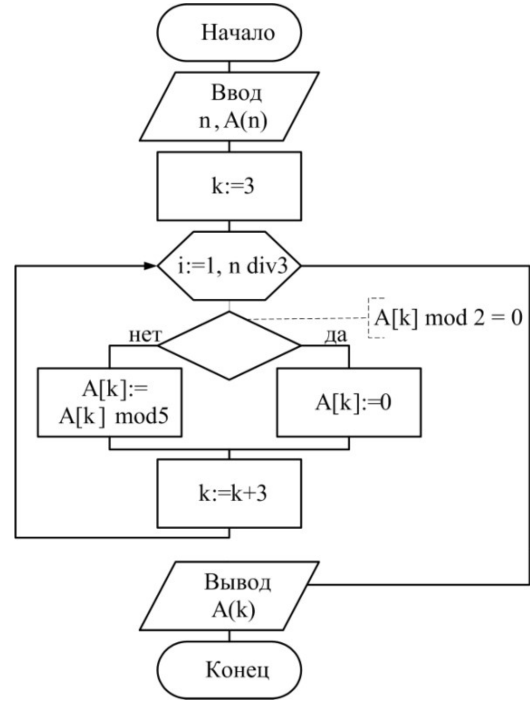


Рисунок 3.6 – Схема алгоритма замены каждого третьего элемента массива

3.1.3. Изменение порядка следования элементов массива. Сортировка

Примерами подобной обработки служат: замена значений некоторых элементов в соответствии с определенными правилами, сортировка массивов по неубыванию, невозрастанию и по любому другому признаку, различного рода перестановки, сдвиги элементов и другое. Особенностью данного класса задач является необходимость поиска элемента перед перестановкой или заменой. Таким образом, несмотря на постоянную размерность массива, для преобразования требуется несколько просмотров его элементов. Поэтому даже в случае одномерного массива требуется применение вложенных циклов. Рассмотрим некоторые типовые приемы.

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

Прежде всего, задачу сложно решить, не используя дополнительный массив, поскольку запись элементов происходит в неизвестном заранее 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 понадобится специальный индекс (индексы), который придется изменять с помощью отдельного оператора.

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 использование и 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.

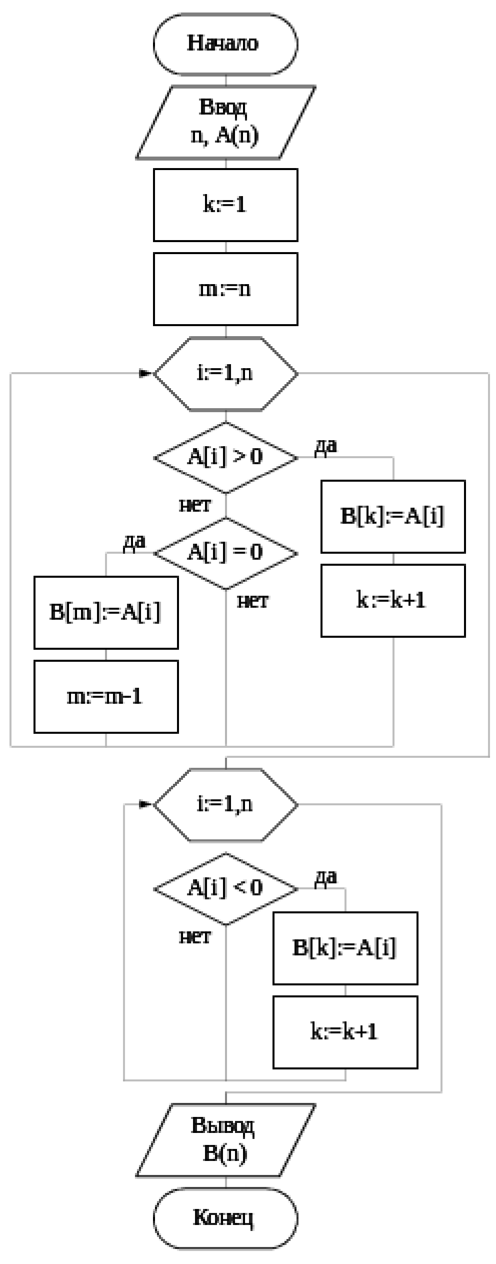


Рисунок 3.7 – Схема алгоритма перестановки элементов массива

Наиболее распространенными задачами второго класса являются сортировки массивов.

Сортировка – это изменение порядка следования элементов массива таким образом, что значения элементов образуют упорядоченную по некоторому закону совокупность. На практике сортируемые массивы обычно имеют большую размерность, поэтому для любого алгоритма сортировки оценивают *временну́ю вычислительную сложность*, которая определяется зависимостью времени работы алгоритма (числа него шагов) от размерности задачи, т.е. от количества сортируемых элементов. Чем меньше это время, тем более эффективен алгоритм. Эффективность алгоритма оценивают также *ёмкостной вычислительной сложностью*, с точки зрения которой существенно, какие ресурсы памяти требуется для выполнения алгоритма. Поэтому интерес представляют такие методы сортировки, которые позволяют экономно использовать оперативную память, например, обходиться без дополнительных массивов.

Классических методов сортировки три: выбором, обменом и вставками. Рассмотрим их особенности применительно к сортировке по неубыванию.

Сортировка выбором. Это один из самых простых методов сортировки. Он предполагает следующую последовательность действий:

- выбирается наименьший элемент всего массива;
- найденный наименьший элемент меняется местами с первым элементом;
- указанные два действия повторяются с оставшимися *n*-1 элементами, *n*-2 элементами и т.д. до тех пор, пока не останется n-ный, наибольший элемент массива.

Пример схемы алгоритма сортировки выбором приведен на рисунке 3.8.

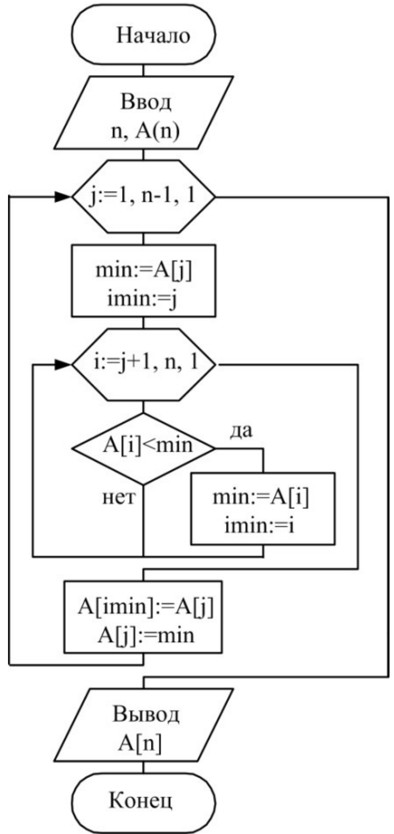


Рисунок 3.8 – Схема алгоритма сортировки выбором

Сортировка обменом (метод «пузырька»). Алгоритм прямого обмена основывается на попарном сравнении соседних элементов и смене их местами в соответствии с видом сортировки. Например, при сортировке по неубыванию обмен элементов местами происходит, если правый элемент в паре меньше левого. Сравнения продолжают до тех пор, пока не будут упорядочены все элементы. Название метода объясняется сходством процесса сортировки с всплыванием пузырьков газа в сосуде с жидкостью: элементы массива можно рассматривать как пузырьки, которые поднимаются к его левой либо правой границе.

Пусть, например, массив состоит из элементов (2, 5, 12, 1, 8, 4). Если сортировать по возрастанию, то после первого «прохода» по массиву мы получим его в следующем виде: (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 свое окончательное место, поэтому можно каждый раз сокращать количество просматриваемых элементов. Схема алгоритма сортировки обменом представлена на рисунке 3.9.

Оглавление

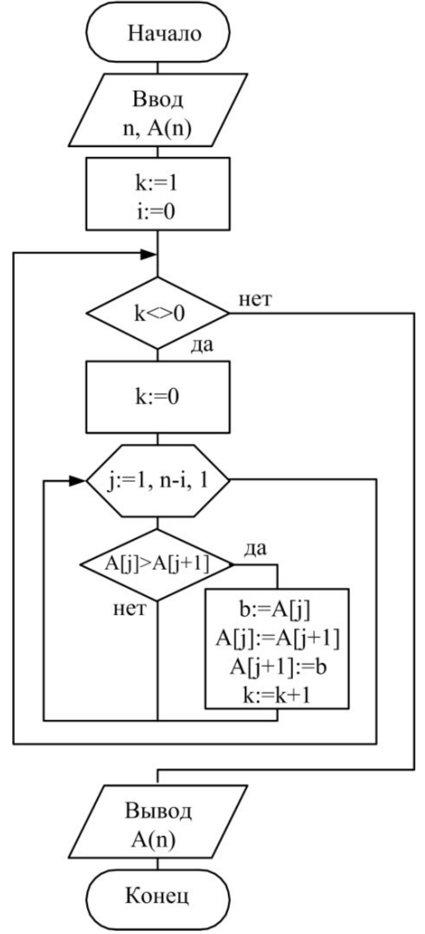


Рисунок 3.9 – Схема алгоритма сортировки обменом

Сортировка вставками. Алгоритм можно описать следующим образом. На каждом шаге, начиная с *i*=2, из исходной последовательности извлекается *i*-й элемент и перемещается 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 завершается при выполнении одного из двух условий: либо найден элемент a_j меньший, чем a_i, либо достигнута левая граница массива.

Данный случай повторяющегося процесса с двумя условиями окончания является типичным и позволяет воспользоваться хорошо известным приемом «барьера». Для этого необходимо расширить диапазон индекса в описании массива от 0 до *n*. Сортируются n элементов с индексами от 1 до n, а ячейка с нулевым индексом служит своеобразным «барьером» для срабатывания условий окончания поиска.

Схема алгоритма сортировки изложенным методом приводится на рисунке 3.10.

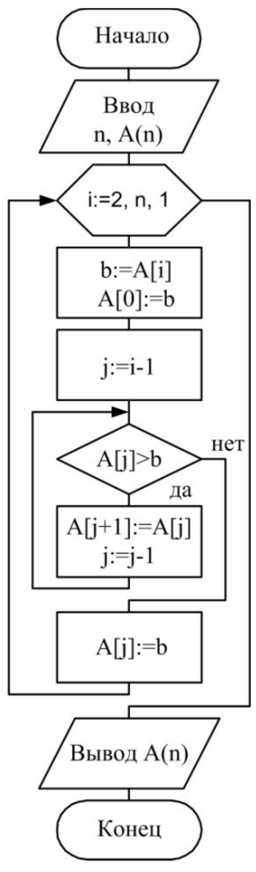


Рисунок 3.10 – Схема алгоритма сортировки вставками

Помимо указанных классических, существует ряд оптимизирующих методов сортировки. Одним из них является **метод Шелла.** Он 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 находились два элемента, т.е. *d*=[*n*/2].

3.1.4. Переформирование массива с изменением его размера

Примерами данного класса задач могут служить вычеркивание и вставка элементов, отвечающих определенным условиям или обладающих заданными свойствами (признаками). При этом для удаления или добавления элемента может производиться как полная, так и частичная обработка массива, когда просматриваются не все элементы, а некоторая их выборка.

Особенностью задач класса является изменение размеров исходного массива. Кроме того, вычеркивание или вставка элемента требует сдвига всех тех элементов, которые расположены после удаляемого или вставляемого. Для реализации алгоритмов решения подобных задач, как правило, требуется использование вложенных циклов. При этом внутренний цикл, в котором осуществляется сдвиг, может быть счетным.

Внешний цикл обхода элементов массива имеет переменную верхнюю границу, поэтому он должен быть итерационным – с пред- или постусловием. Если на соответствие условию проверяются все элементы, то параметр внешнего цикла меняется на 1. Однако после сдвига на месте прежнего анализируемого элемента может оказаться элемент, отвечающий требуемому условию. Поэтому для определения количества повторений цикла сдвигов необходим цикл с постусловием, а не оператор ветвления. Таким образом, в алгоритме задействуются три цикла, вложенные один в другой.

При выборочном анализе элементов массива параметр внешнего цикла меняется на разную величину в зависимости от наличия или отсутствия сдвига. Для принятия решения о выполнении сдвига можно использовать оператор условной передачи управления.

Если количество удаляемых или вставляемых элементов большое, то вычислительная сложность подобных алгоритмов достаточно велика.

Пример 3.7. Удалить из массива целых чисел все числа, кратные 5.

Для алгоритмизации задачи можно предложить два пути:

- просматривая массив, найти очередной удаляемый элемент, а затем путем сдвига перенести все элементы, расположенные после него, затирая удаляемый элемент. В полученном массиве окажется на один элемент меньше. Изменив размер массива, следует продолжить просмотр оставшихся элементов (см. рисунок 3.11, *а*);
- просматривая массив, переписывать остающиеся элементы на очередное новое место. В этом случае потребуются две переменные для индекса: в качестве первой будем использовать параметр цикла, а вторую будем изменять независимо после записи очередного остающегося элемента (см. рисунок 3.11, *б*).

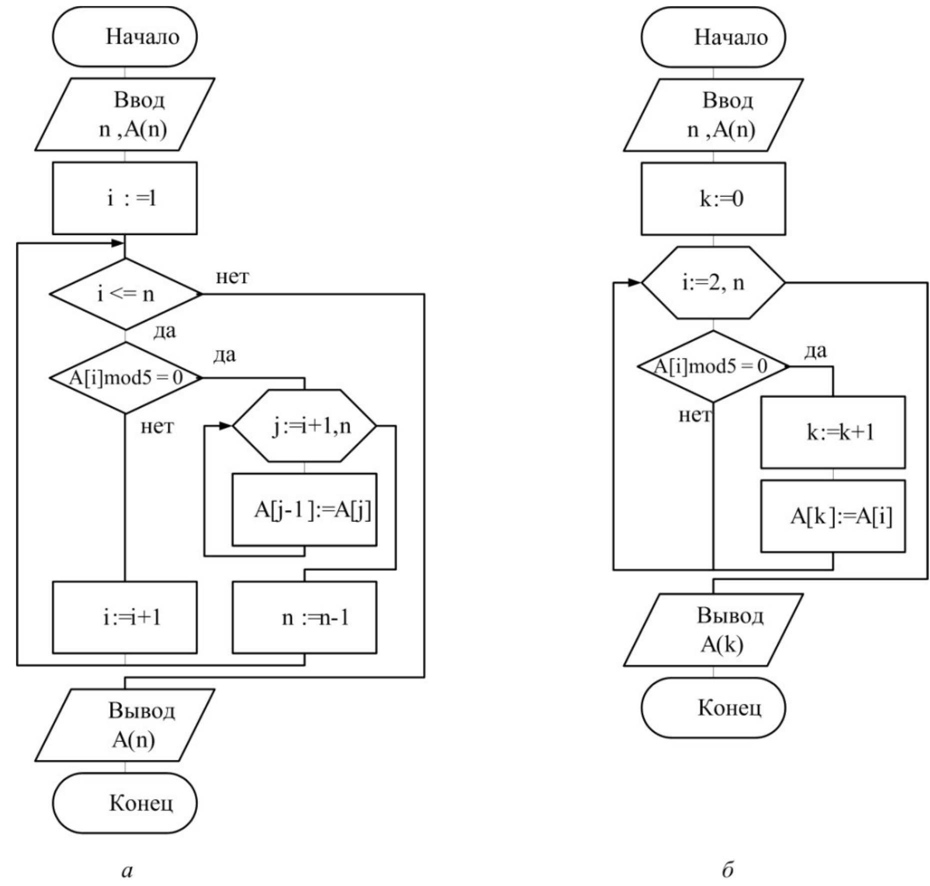


Рисунок 3.11 – Схемы алгоритма решения задачи:

а – с двумя циклами; *б* – с одним циклом

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

Нетрудно заметить, что вычислительная сложность второго алгоритма меньше, чем первого. Но первый алгоритм легче для восприятия. Практика показывает, что второй способ практически всегда применим, если элементы из массива удаляются. При добавлении элементов второй способ не применим, поскольку из-за отсутствия дополнительного массива переписываемые элементы рано или поздно начнут затирать еще не проверенную часть массива.

Пример 3.8. Переформировать массив вещественных чисел, добавив в него нулевые элементы после положительных элементов, стоящих на четных местах.

В данной задаче, очевидно, необходим второй цикл, причем следует иметь в виду, что при выполнении сдвига ни один из элементов не должен быть затерт, а значит, сдвигать элементы придется с последнего. Кроме того, необходимо обеспечить обход элементов, расположенных на четных местах. Следовательно, параметр цикла обхода необходимо менять на две единицы.

После обнаружения очередного положительного элемента, стоящего на четном месте, требуется сдвиг всех оставшихся элементов. При этом все элементы, стоявшие на четных местах, окажутся на нечетных. Значит, для правильного добавления нулевого элемента необходимо перейти через три элемента, а не через два. Для того, чтобы в результате работы алгоритма вывести весь массив, надо также подсчитать количество добавленных элементов. И, конечно, следует предусмотреть в массиве свободное место для размещения добавляемых элементов. Схема одного из возможных алгоритмов решения задачи представлена на рисунке 3.12.

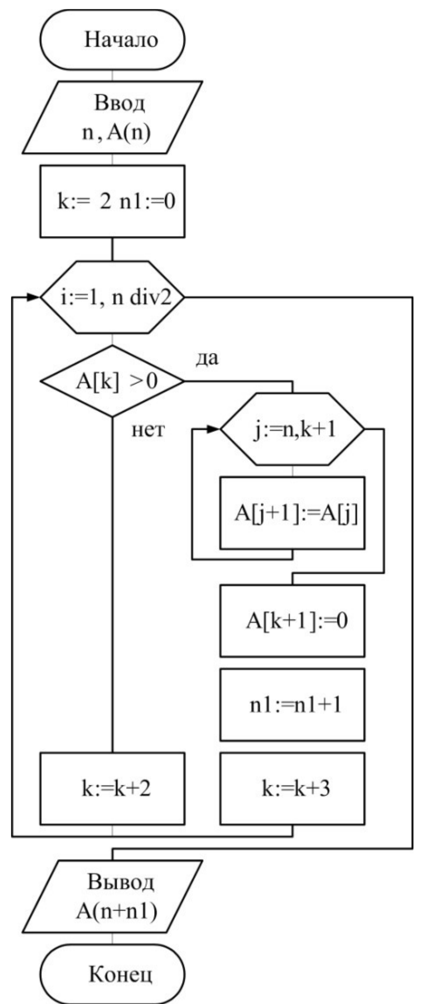


Рисунок 3.12 – Схема алгоритма добавления элементов в массив

Оглавление

3.1.5. Одновременная обработка нескольких массивов или подмассивов

К этому классу относятся такие задачи как слияние (объединение) массивов; переписывание элементов, отвечающих определенному условию или имеющих некоторый признак, из одного массива в другой; создание нового массива из элементов исходного, преобразованных по некоторой формуле или подчиняющихся определенному закону. Особенностью задач этого класса является наличие у каждого массива своего индекса, который меняется в своем диапазоне по своему закону. При алгоритмизации подобных задач можно использовать как счетные циклы, так и циклы с пред- или постусловием, причем выбор зависит от законов изменения индексов и от того, как грамотно обеспечить их изменение.

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

Поскольку задача относится к классу задач по одновременной обработке нескольких массивов, каждому массиву будет усвоен уникальный индекс для доступа к его элементам. Кроме того, размеры исходных массивов в общем случае различны, поэтому у каждого индекса будет свой диапазон изменения. Так как в результате работы программы должен быть сформирован новый массив, элементами которого станут элементы обоих исходных массивов, то размер нового массива должен быть равен сумме размеров исходных массивов. Соответственно, диапазон изменения его индекса будет определяться суммой диапазонов изменения индексов исходных массивов.

Примем следующую последовательность действий для решения поставленной задачи. Сравниваем первые элементы исходных массивов A и B и наименьший из них записываем первым элементом результирующего массива C. Увеличиваем на единицу индекс результирующего массива и индекс того из массивов A и B, элемент которого был переписан. Продолжаем аналогично, последовательно формируя массив C. Так как размеры исходных массивов различны, то просмотр одного из них закончится раньше. После того, как его элементы окажутся исчерпаны, в массив C дописываем оставшиеся элементы второго массива.

Схема алгоритма решения данной задачи показана на рисунке 3.13.

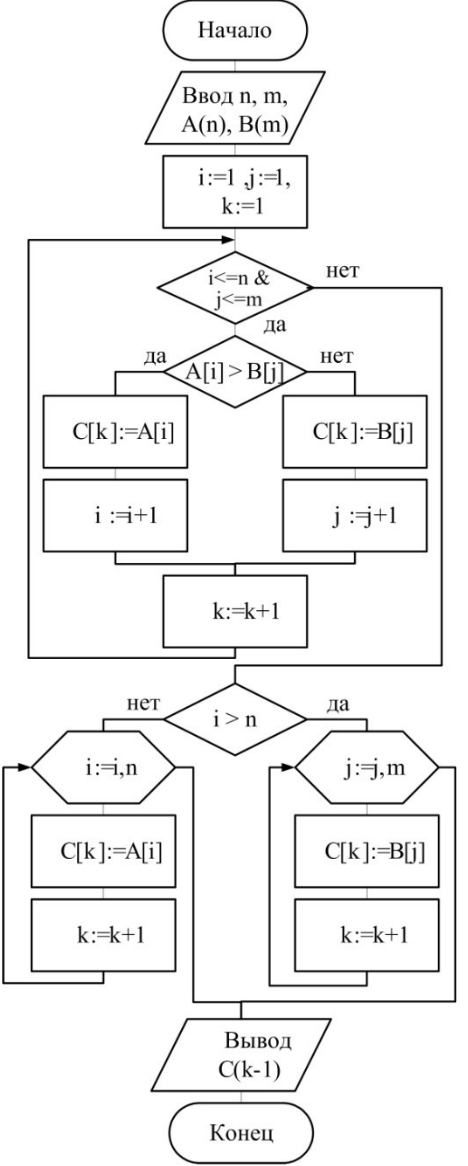


Рисунок 3.13 – Схема алгоритма слияния отсортированных массивов

3.1.6. Поиск в массиве элемента, отвечающего заданному условию

К этому классу относятся, например, следующие задачи: поиск первого отрицательного, первого положительного и любого первого элемента, отвечающего некоторому условию, поиск первого или единственного элемента, равного некоторому конкретному значению, а также поиск всех элементов массива, отвечающих заданному условию. В рамках настоящего учебного пособия ограничимся задачей поиска первого или единственного элемента, удовлетворяющего условию.

Особенностью задачи является отсутствие необходимости просмотра всего массива: перебор элементов нужно закончить сразу, как только требуемый элемент будет найден. При этом может выполнять как поэлементный просмотр, так и выборочную обработку массива. Однако в худшем случае для поиска элемента потребуется полный перебор всех элементов массива. Такой поиск называется *линейным*. Если размер массива невелик, то затратами времени на линейный поиск можно пренебречь. Однако в больших массивах полный перебор приводит к заметным затратам машинного времени. Следовательно, требуется оптимизация поискового алгоритма. Примерами оптимизирующих алгоритмов могут служить поиск с «барьером» и двоичный (бинарный) поиск.

Чаще всего при программировании поисковых задач используются циклы с пред- или постусловием, которое фактически представляет собой конъюнкцию двух условий: первое – пока искомый элемент не найден, второе – пока массив не исчерпан. После выхода из цикла осуществляется проверка, по какому из условий произошел выход.

Рассмотрим следующие примеры.

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

Задачу можно решить, если, например, последовательно сравнить каждый элемент массива с последующим. Так как в худшем случае пара одинаковых элементов может находиться в самом конце массива или вовсе не существовать, то одним из условий является достижение конца массива, а другим – обнаружение двух одинаковых соседних элементов. Схему алгоритма решения данной задачи см. на рисунке 3.14.

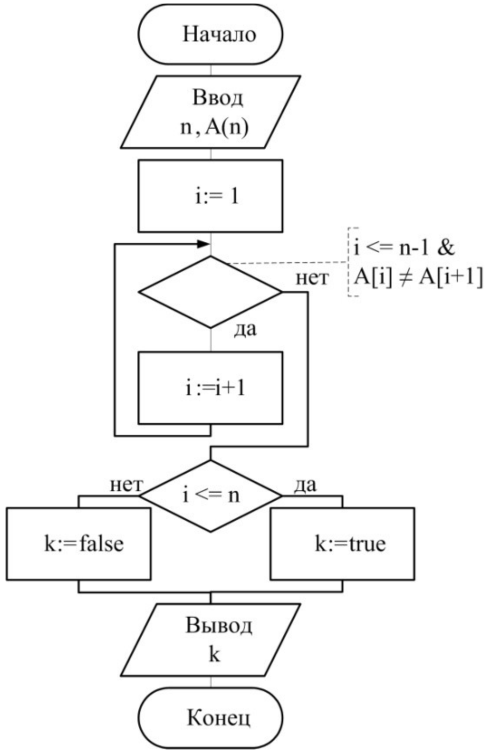


Рисунок 3.14 – Схема алгоритма поиска пары одинаковых элементов в массиве

Пример 3.11. Среди элементов массива, стоящих на четных местах, найти первый отрицательный элемент.

Логика решения задачи ясна из схемы алгоритма, показанной на рисунке 3.15.

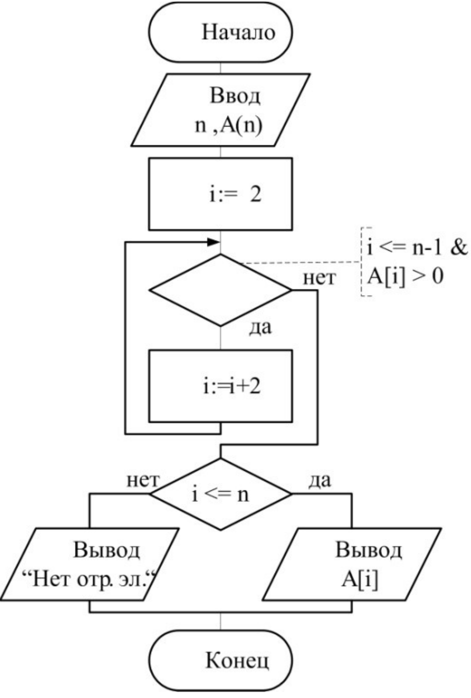


Рисунок 3.15 – Схема алгоритма поиска в массиве первого отрицательного элемента на четном месте

3.2. Приемы обработки матриц

Массивы, имеющие два индекса, называют двумерными или, по аналогии с математикой, *матрицами*. Для краткости будем в дальнейшем придерживаться именно этого термина.

Рассмотрим наиболее распространенные приемы программирования при обработке матриц. Следует отметить, что алгоритмизация задач всех классов применительно к матрицам имеет свою специфику, связанную с тем, что матрица фактически является массивом одномерных массивов. Это значит, что в каждом классе имеется гораздо больше различных вариантов решений, да и самих задач также. Каждая конкретно поставленная задача обработки матрицы содержит целый ряд подзадач, относящихся к разным классам.

3.2.1. Последовательная обработка элементов матрицы

К данному типу задач относятся не только те, при решении которых требуется обработать все элементы матрицы, но также и задачи по обработке отдельно строк или отдельно столбцов. Кроме того, к этому классу могут быть отнесены и задачи ввода-вывода матриц. Особенности программирования в данном случае, в основном, те же, что и для одномерных массивов (см. раздел 4.1.1).

Рассмотрим некоторые типовые алгоритмы задач первого класса.

Пример 3.12. Дана целочисленная квадратная матрица порядка *n*. Найти наибольший элемент матрицы и количество таких элементов.

Для решения задачи необходимо организовать цикл для перебора всех элементов матрицы. Так как матрица является двумерным массивом, то потребуются два цикла: один по строкам, второй – по столбцам. Один из циклов, обычно тот, в котором перебираются элементы столбцов, должен быть вложенным в другой, то есть для каждой строки матрицы перебираются принадлежащие ей элементы всех столбцов. В остальном алгоритм решения задачи (см. рисунок 3.16) схож с алгоритмом нахождения наибольшего элемента одномерного массива.

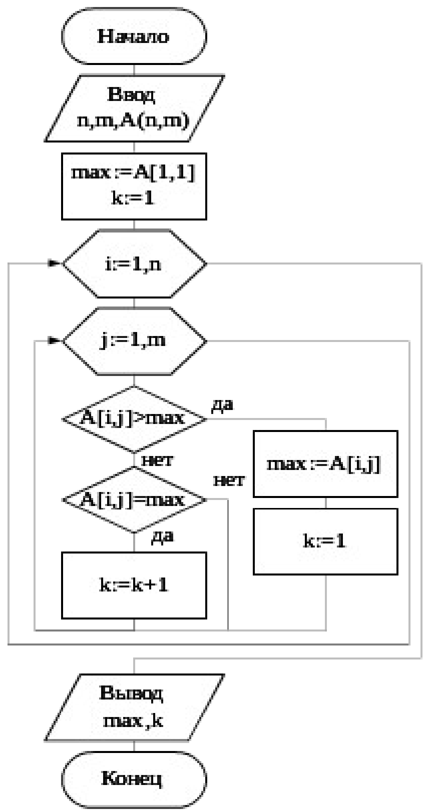


Рисунок 3.16 – Схема алгоритма поиска наибольших элементов матрицы

Пример 3.13. Дана целочисленная матрица *A*(*n*, *m*). Вычислить сумму каждой строки и результат записать в массив *B*(*n*).

Для нахождения суммы элементов строки с номером *i* необходимо организовать цикл, перебирающий все ее элементы. 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 элементу массива *B*(*i*). Для обеспечения обхода всех строк матрицы параметром внешнего цикла назначаем переменную *i* – индекс строки. Схема алгоритма показана на рисунке 3.17).

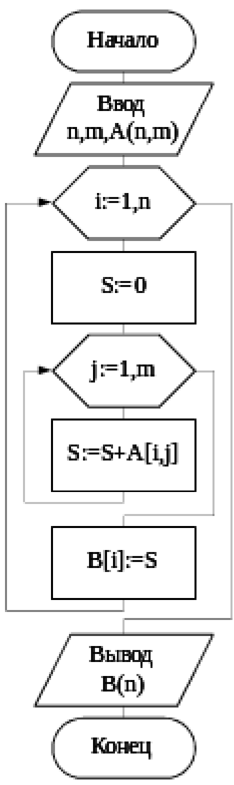
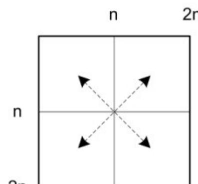


Рисунок 3.17 – Схема алгоритма вычисления суммы элементов каждой строки матрицы

3.2.2. Изменение порядка следования элементов матрицы

Здесь, как и в задачах, рассмотренных в разделе 3.1.3, размер исходной матрицы сохраняется, меняется лишь порядок следования ее элементов. В качестве примера задач данного класса рассмотрим задачу перемещения элементов матрицы.

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



Для решения данной задачи, как и других подобных задач, необходимо проанализировать, как будет происходить модификация индексов элементов матрицы в соответствии с поставленным условием изменения порядка их следования, и выработать правило преобразования индексов. Будем считать, что после преобразования исходной матрицы *A* получается новая квадратная матрица *B* того же порядка 2*n*. Изучив

соответствие между элементами матриц *A* и *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 задачи представлена на рисунке 3.18.

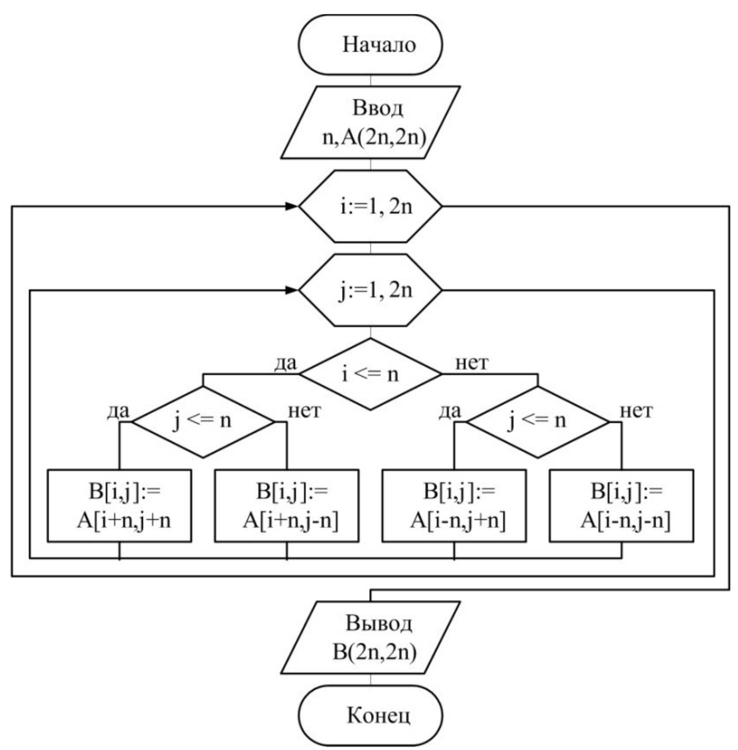


Рисунок 3.18 – Схема алгоритма перестановки элементов матрицы

Следует отметить, что приведенные примеры не охватывают всего многообразия задач на обработку матрицам. Как было отмечено выше, в процессе алгоритмического решения многих из них используются приемы, применяемые при обработке одномерных массивов. Однако существуют и специфические приемы, относящиеся именно к матрицам. Они требуют отдельного, более детального рассмотрения, что выходит за пределы задачи, которую ставили перед собой авторы настоящего учебного пособия.

Контрольные вопросы

1. Что называют массивом? Каким образом определяется место каждого элемента в массиве?
2. Какие классы задач обработки массивов можно выделить?
3. Какой тип цикла наиболее подходит для последовательной обработки всех элементов массива? Приведите примеры задач данного класса.
4. В чем состоит основная особенность алгоритмизации задач выборочной обработки элементов массива? Приведите примеры подобного рода задач.
5. В чем заключается задача сортировки одномерного массива? Какие методы сортировки вы знаете?
6. Сформулируйте основные алгоритмические особенности каждого из рассмотренных методов сортировки одномерных массивов.
7. Перечислите основные особенности алгоритмов решения задач переформирования массива с изменением его размера. Приведите примеры таких задач.
8. Каковы особенности алгоритмизации задач, связанных с одновременной обработкой нескольких массивов?
9. Какова особенность алгоритмического решения задач поиска в массиве элемента, отвечающего заданному условию? Какие типы циклов при этом чаще всего используются?
10. В чем состоит специфика обработки матриц по сравнению с приемами обработки одномерных массивов?
11. В чем состоит особенность организации циклического процесса при последовательной обработке всех элементов матрицы?
12. Какую аналитическую работу необходимо выполнить перед составлением алгоритма решения задачи, связанной с изменением порядка следования элементов в матрице?

Литература

1. Иванова Г.С. Программирование: учебник. – М.: КНОРУС, 2013. – 432 с. – (Бакалавриат).
2. Демидович Б.П., Марон И.А. Основы вычислительной математики. М.: Изд-во «Лань», 2009. – 672 с.