Туаева М.Т.

АЛГОРИТМЫ

основы алгоритмизации

учебное пособие

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РСО-АЛАНИЯ

ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ «ВЛАДИКАВКАЗСКИЙ ТОРГОВО-ЭКОНОМИЧЕСКИЙ ТЕХНИКУМ»

АЛГОРИТМЫ
ОСНОВЫ АЛГОРИТМИЗАЦИИ

УЧЕБНОЕ ПОСОБИЕ

Владикавказ
2020

Одобрено
предметной (цикловой)
комиссией компьютеризации,
физики, математики

Протокол №
От « »_____2020г.

Председатель _____
И.С. Пархоменко

Составитель _____
М.Т. Туаева

Согласовано
Методист _____
З.А. Дзантиева

Утверждаю
Заместитель директора по УР
_____ Т.В. Иванова
« » _____ 2020 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Учебное пособие предназначено для тех, кто только начинает изучать программирование.

В учебном пособии приводятся сведения о типовых алгоритмах обработки данных безотносительно к языку программирования. Приведены примеры решения некоторых классов задач, где для каждой задачи разработан алгоритм в виде блок-схемы с пояснениями, набор тестовых данных и таблица исполнения алгоритма. Приводимые способы решения задач по возможности являются рациональными, но не претендуют на то, чтобы быть наилучшими.

Материал пособия не требует никаких предварительных знаний об алгоритмах и языках программирования, поэтому пособие может быть рекомендовано для самостоятельного изучения.

1. ОПРЕДЕЛЕНИЕ АЛГОРИТМА И ЕГО СВОЙСТВА

Под алгоритмом понимается точное предписание, задающее последовательность действий, которая ведет от произвольного исходного данного (или от некоторой совокупности возможных для данного алгоритма исходных данных) к достижению полностью определяемого этим исходным данным результата.

Алгоритм должен обладать определенными свойствами, наличие которых гарантирует получение решения задачи исполнителем.

Дискретность. Решение задачи должно быть разбито на элементарные действия. Запись отдельных действий реализуется в виде упорядоченной последовательности отдельных команд, образующих дискретную структуру алгоритма. Это свойство непосредственно отражено в определении алгоритма.

Понятность. На практике любой алгоритм предназначен для определенного исполнителя, и любую команду алгоритма исполнитель должен уметь выполнить.

Определенность (детерминированность). Каждая команда алгоритма должна определять однозначные действия исполнителя. Результат их исполнения не должен зависеть от факторов, не учтенных в алгоритме явно. При одних и тех же исходных данных алгоритм должен давать стабильный результат.

Массовость. Разработанный алгоритм должен давать возможность получения результата при различных исходных данных для однотипных задач.

Например, пользуясь алгоритмом решения квадратного уравнения, можно находить его корни при любых значениях коэффициентов.

Свойство массовости полезное, но не обязательное свойство алгоритма, так как интерес представляют и алгоритмы, пригодные для решения единственной задачи.

Результативность (конечность). Это свойство предполагает обязательное получение результата решения задачи за конечное число шагов. Под решением задачи понимается и сообщение о том, что при заданных значениях исходных данных задача решения не имеет.

Если решить задачу при заданных исходных данных за конечное число шагов не удается, то говорят, что алгоритм

«зацикливается».

Смысл условий дискретности, понятности и определенности ясен: их нарушение ведет к невозможности выполнения алгоритма. Остальные условия не столь очевидны. Для сложных алгоритмов выполнить исчерпывающую проверку результативности и корректности невозможно. Это равносильно полному решению задачи, для которой создан алгоритм, вручную.

Можно сформулировать общие правила, руководствуясь которыми следует записывать алгоритм решения задачи.

1. Выделить величины, являющиеся исходными данными для задачи.
2. Разбить решение задачи на такие команды, каждую из которых исполнитель может выполнить однозначно.
3. Указать порядок выполнения команд.
4. Задать условие окончания процесса решения задачи.
5. Определить, что является результатом решения задачи в каждом из возможных случаев.

Хотя алгоритмы обычно предназначены для автоматического выполнения, они создаются и разрабатываются людьми. Поэтому первоначальная запись алгоритма обычно производится в форме, доступной для восприятия человеком.

Самой простой является *словесная* форма записи алгоритмов на естественном языке. В этом виде алгоритм представляет собой описание последовательности этапов обработки данных, изложенное в произвольной форме.

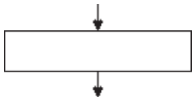
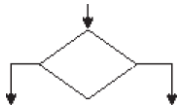
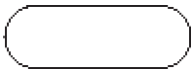
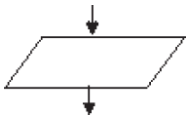
Словесная форма удобна для человеческого восприятия, но страдает многословностью и неоднозначностью.

Когда запись алгоритма формализована частично, то используется *псевдокод*. Он содержит как элементы естественного языка, так и формальные конструкции, описывающие базовые алгоритмические структуры. Эти конструкции называются *служебными словами*. Формального определения псевдокода или строгих правил записи алгоритмов в таком формате не существует.

Графическая форма представления алгоритма является более компактной. Алгоритм изображается в виде последовательности связанных между собой блоков, каждый из которых соответствует выполнению одного или нескольких действий. Графическое представление алгоритма называется *блок-схемой*. Блок-схема определяет структуру алгоритма.

Графические обозначения блоков стандартизованы. Некоторые из часто используемых блоков представлены в таблице 1.

Таблица 1. Изображение основных блоков на блок-схеме

Обозначение блока	**Пояснение**
	Процесс (вычислительное действие, реализованное операцией присваивания)
	Решение (проверка условия, реализующая условный переход)
	Начало, конец алгоритма
	Ввод-вывод в общем виде
	Модификация (начало цикла с параметром)

Отдельные блоки соединяются линиями переходов, которые определяют очередность выполнения действий. Направление линий сверху вниз или слева направо принимается за основное.

Алгоритм, записанный на языке программирования, называется *программой*. При использовании этих языков запись алгоритма абсолютно формальна и пригодна для выполнения на ЭВМ. Отдельная конструкция языка программирования называется *оператором*. Программа — это упорядоченная последовательность операторов.

2. БАЗОВЫЕ АЛГОРИТМИЧЕСКИЕ КОНСТРУКЦИИ

Число реализованных конструкций конечно в любом языке программирования. Структурной элементарной единицей алгоритма является команда, обозначающая один элементарный шаг обработки или отображения информации. Простая команда на языке блок-схем изображается в виде функционального блока «процесс», который имеет один вход и один выход. Из команд проверки условий и простых команд образуются составные команды, имеющие более сложную структуру, но тоже один вход и один выход.

Алгоритм любой сложности может быть представлен комбинацией трех базовых структур:

- следование;
- ветвление (в полной и сокращенной форме);
- цикл (с предусловием или постусловием).

Характерной особенностью этих структур является наличие у них одного входа и одного выхода.

2.1 Линейные алгоритмы

Базовая структура «следование» означает, что не сколько операторов выполняются последовательно друг за другом, и только один раз за время выполнения программы. Структура «следование» используется для реализации задач, имеющих *линейный* алгоритм решения. Это означает, что такой алгоритм не содержит проверок условий и повторений, действия в нем выполняются последовательно, одно за другим.

Пример. Построить блок-схему алгоритма вычисления высот треугольника со сторонами a, b, c по формулам:

$$h_a = \frac{2}{a}\sqrt{p(p-a)(p-b)(p-c)}$$

$$h_b = \frac{2}{b}\sqrt{p(p-a)(p-b)(p-c)}$$

$$h_c = \frac{2}{c}\sqrt{p(p-a)(p-b)(p-c)}$$

$p = \frac{(a+b+c)}{2}$ — полупериметр.

Для того чтобы не вычислять три раза одно и то же значение,

введем вспомогательную величину:

$$t = 2\sqrt{p(p-a)(p-b)(p-c)}$$

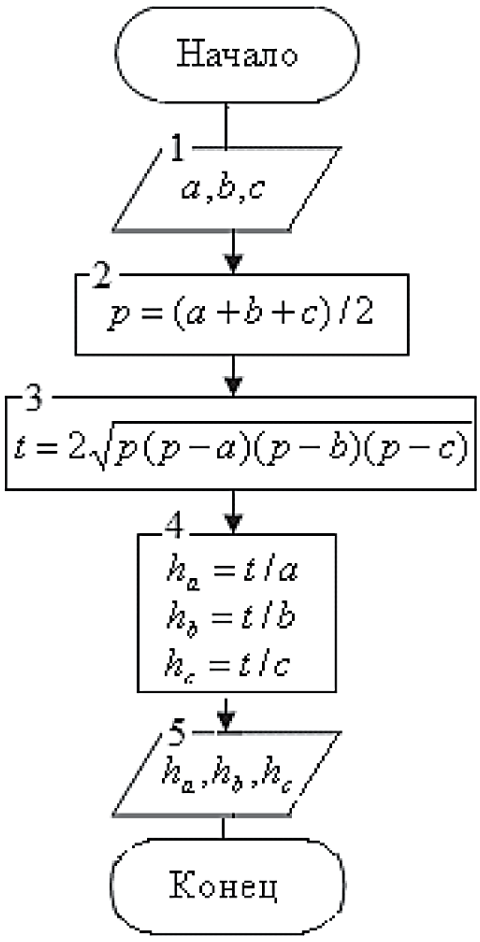
Блок 1. Ввод значений сторон треугольника.

Блок 2. Вычисление полупериметра.

Блок 3. Вычисление вспомогательной величины t.

Блок 4. Вычисление высот, опущенных на стороны a, b, c.

Блок 5. Вывод результатов.



2.2 Разветвляющиеся алгоритмы

Второй базовой структурой является «ветвление». Эта структура обеспечивает, в зависимости от результата проверки условия, выбор одного из альтернативных путей работы алгоритма, причем каждый из путей ведет к общему выходу, так что работа алгоритма будет продолжаться независимо от того, какой путь будет выбран.

Существует структура с полным и неполным ветвлением.

Структура с полным ветвлением (*если — то — иначе*) записывается так:

Если < условие >
то <действия 1>
иначе <действия 2>
Все если

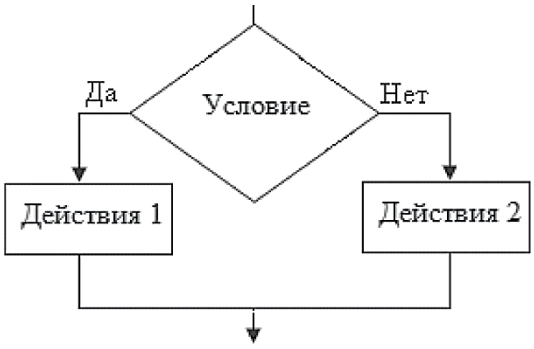
Команда выполняется так: если <условие> является истинным, то выполняются <действия 1>, записанные после ключевого слова *то*, если <условие> является ложным, то выполняются <действия 2>, записанные после слова *иначе*.

Структура с неполным ветвлением (*если — то*) не содержит части, начинающейся со слова *иначе*:

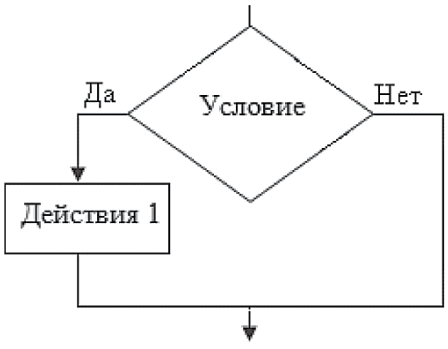
Если <условие>
то <действия 1>
Все если

Команда выполняется так: если <условие> является истинным, то выполняются <действия 1>, записанные после ключевого слова *то*.

Блок-схема алгоритма с ветвлением выглядит так:



Полное ветвление.
Структура *Если — То — Иначе*



Неполное ветвление.
Структура *Если — То*

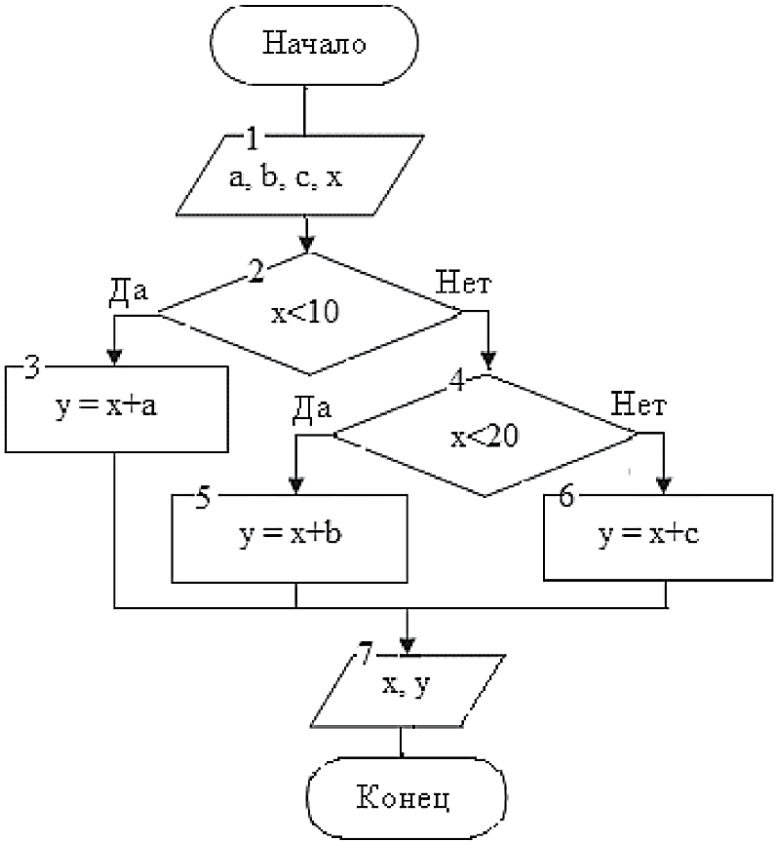
Пример. Вычислить значение функции

$$y = \begin{cases} x + a, \text{ при } x < 10; \\ x + b, \text{ при } 10 \le x \le 20; \\ x + c, \text{ при } x > 20. \end{cases}$$

Дано: *x*, *a*, *b*, *c* — произвольные числа.
Найти: *y*.
Представим задачу графически на числовой оси

$x < 10$	$10 \le x \le 20$	$x > 20$
10		20
$y = x + a$	$y = x + b$	$y = x + c$
1 интервал	2 интервал	3 интервал

Так как значение переменной *x* вводится произвольно, то оно может оказаться в любом из трех интервалов.
Приведем блок-схему.



Блок 1. Ввод исходных данных.

Блок 2. Проверка введенного значения. Если $x < 10$ (выход «Да»), то точка находится в первом интервале. В противном случае $x \ge 10$ (выход «Нет»), и точка может оказаться во втором или третьем интервале.

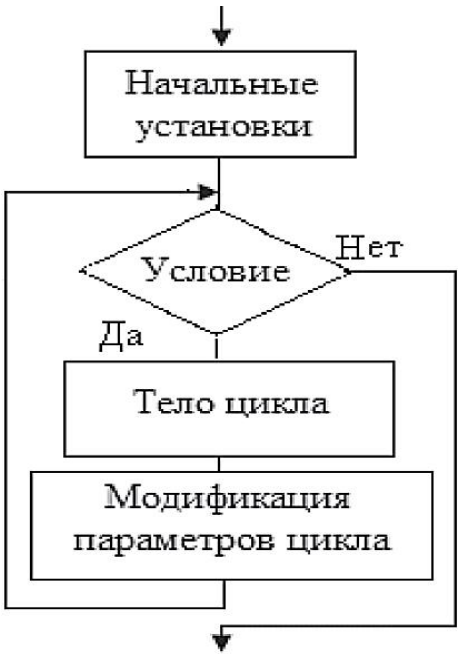
Блок 4. Проверка ограничения значения *x* справа ($x < 20$). Если условие выполняется (выход «Да»), то *x* находится во втором интервале, иначе $x \ge 20$, и точка находится в третьем интервале.

Блоки 3, 5 и 6. Вычисление значения *y*.

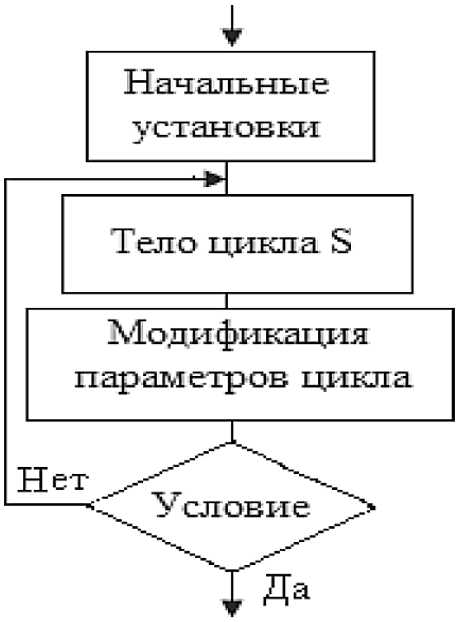
2.3 Циклические алгоритмы

При составлении алгоритмов решения большинства задач возникает необходимость в неоднократном повторении одних и тех же команд. Алгоритм, составленный с использованием многократных повторений одних и тех же действий (циклов), называется *циклическим*. Однако слово «неоднократно» не означает «до бесконечности». Организация циклов, никогда не приводящая к остановке в выполнении алгоритма («зацикливание» алгоритма), нарушает требование его результативности — получения результата за конечное число шагов.

Блок, для выполнения которого организуется цикл, называется *телом цикла*. Остальные операторы служат для управления процессом повторения вычислений: это начальные установки, проверка условия продолжения цикла и модификация параметра цикла. Один проход цикла называется *итерацией*.



Цикл с предусловием



Цикл с постусловием

Начальные установки служат для того, чтобы до входа в цикл задать значения переменных, которые в нем используются.

Проверка условия продолжения цикла выполняется на каждой итерации либо до тела цикла (*цикл с предусловием*), либо после тела цикла (*цикл с постусловием*). Тело цикла с постусловием всегда выполняется хотя бы один раз. Проверка необходимости выполнения цикла с предусловием делается до начала цикла, поэтому возможно, что он не выполнится ни разу.

При конструировании циклов следует соблюдать обязательное условие результативности алгоритма (т. е. его окончания за конечное число шагов). Практически это означает, что в условии должна быть переменная, значение которой изменяется в теле цикла. Причем, изменяется таким образом, чтобы условие в конечном итоге перестало выполняться. Такая переменная называется управляющей переменной цикла или *параметром* цикла.

Еще один вид циклов — цикл с *параметром*, или *арифметический цикл*. Тело цикла выполняется, пока параметр цикла i пробегает множество значений от начального (I_n) до конечного (I_k).

Переменная i определяет количество повторений тела цикла S. Если шаг изменения значения параметра цикла обозначить через ΔI, то количество повторений тела цикла n можно вычислить по формуле

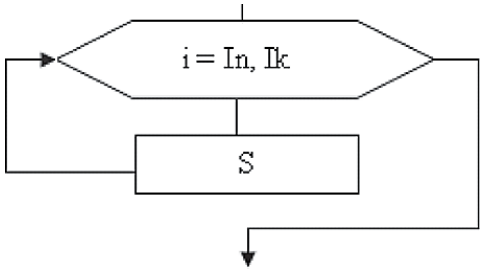
$$n = \frac{Ik - In}{\Delta I} + 1.$$

Если параметр цикла i изменяется с шагом 1, то шаг может не указываться.

Цикл выполняется так: начальное значение параметра цикла i равно I_n. Если $i \le I_k$, выполняется тело цикла S, после чего параметр цикла увеличивается на 1 с помощью опера- тора присваивания $i = i + 1$, и снова проверяется условие $i \le I_k$.

Пример. Дано целое положительное число n. Вычислить факториал этого числа. Известно, что факториал любого целого положительного числа n определяется как произведение чисел от 1 до заданного числа n:

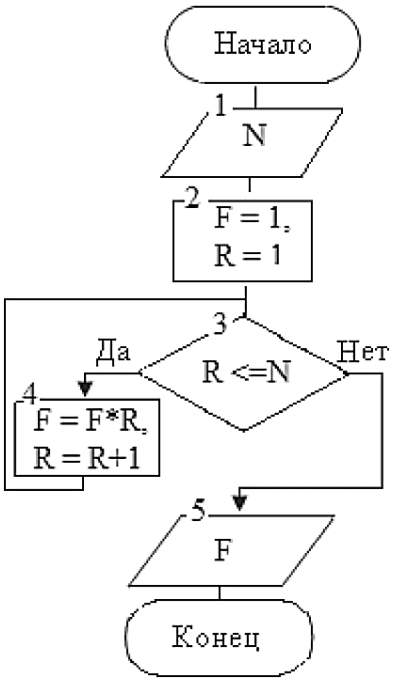
$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n.$$



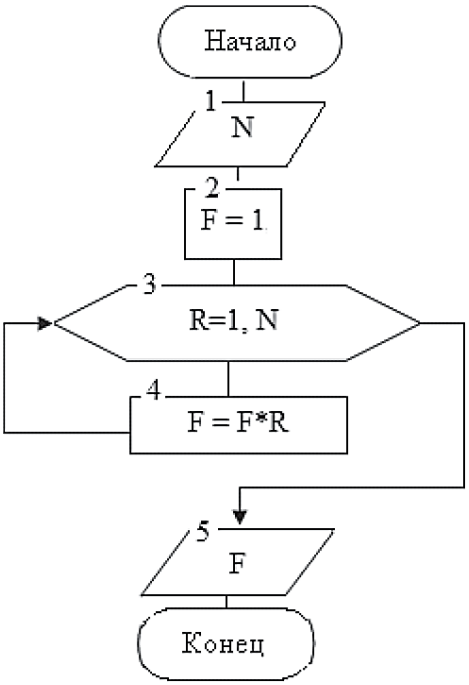
По определению 0! = 1 и 1! = 1.

Задача решается с помощью циклического алгоритма. Введем следующие обозначения: N — заданное число, F — факториал числа, R — параметр цикла. Составим два варианта алгоритма: с использованием цикла с предусловием и цикла с параметром.

Правильность алгоритма можно проверить, если выполнить его формально «вручную». Выполним алгоритм при $n = 4$.



Цикл с предусловием



Цикл с параметром

При решении данной задачи выполнение цикла с предусловием ничем не отличается от выполнения цикла с параметром. При $R = 5$ произойдет выход из цикла и окончательное значение $4! = 24$.

В чем же отличие? Посмотрим на структуру этих циклов. В цикле *пока* начальное значение параметра цикла R задается перед входом в цикл, в теле цикла организовано изменение параметра цикла командой $R = R + 1$, условие $R \le N$ определяет условие продолжение цикла. В цикле с заданным числом повторений эти же команды неявно заданы в операторе заголовка цикла.

Пример. Пусть $a_0 = 1$; $a_k = k \cdot a_{k-1} + 1/k$, $k = 1, 2, \ldots$ Дано натуральное число n. Получить a_n.

Дано: a_0 — первый член последовательности; n — номер члена последовательности, значение которого требуется найти.

Найти: a_n — n-й член последовательности.

Математическая модель. Посмотрим, как изменяется значение члена последовательности при изменении значения k. При

$k = 1$ $a_1 = 1 \cdot a_0 + 1$. При $k = 2$ $a_2 = 2 \cdot a_1 + 1/2$. При $k = 3$ $a_3 = 3 \cdot a_2 + 1/3$. Выполнив указанные вычисления n раз, получим искомое значение a_n. В задачах такого типа не требуется хранить результаты вычислений на каждом шаге. Поэтому можно использовать простые переменные.

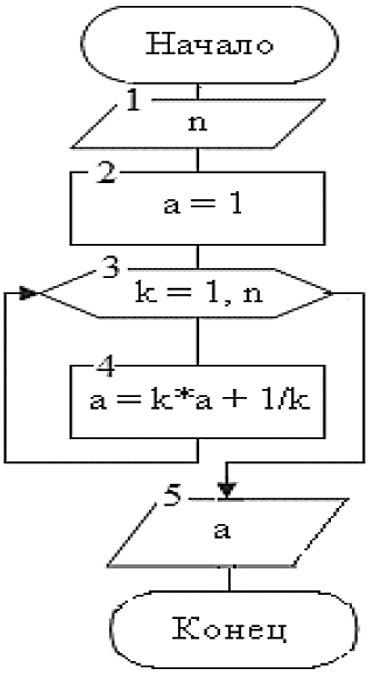
Обозначим через a — произвольный член последовательности. Тогда формула для вычисления члена последовательности будет выглядеть так:

$$a = k \cdot a + 1/k.$$

В этой формуле значение a, стоящее справа от знака «=», определяется на предыдущем шаге вычисления, а значение a, стоящее в левой части выражения, определяется на данном шаге и заменяет в памяти предыдущее значение.

Переменная a — вещественного типа; переменные k и n — целого типа.

Приведем словесное описание и блок-схему алгоритма.



Блок 1. Ввод количества членов последовательности n. Блок 2. Присваивание начального значения $a = 1$.

Блок 3. Арифметический цикл. Начальное значение параметра цикла k равно 1, конечное значение равно n, шаг изменения параметра — 1.

Блок 4. Выполнение тела цикла. Для каждого значения k вычисляется выражение $a = k * a + 1/ k$.

Блок 5. Вывод значения a.

3. АЛГОРИТМЫ, ИСПОЛЬЗУЮЩИЕ ОДНОМЕРНЫЕ МАССИВЫ

Массив — это упорядоченный набор однотипных значений (элементов массива). Каждый массив имеет имя, что дает возможность различать массивы между собой и обращаться к ним по имени.

Каждый элемент массива имеет три характеристики:

1) *имя*, совпадающее с именем массива;

2) *индекс* — целое число или множество целых чисел, однозначно определяющее местоположение элемента в массиве. В качестве индекса может использоваться также переменная или арифметическое выражение целого типа. Примеры индексов: 3, 15, i, j, $i - 1$, $j + 2$;

3) *значение* — фактическое значение элемента, определенное его типом.

Элементы массива могут использоваться произвольно и являются одинаково доступными. Доступ к элементам массива производится по его индексу.

Массивы могут быть *одномерными* и *многомерными.* В этом параграфе рассмотрим некоторые алгоритмы на одномерных массивах.

3.1 Ввод и вывод элементов массива

Одномерный массив определяется именем и числом элементов (размером), и мы обозначим его $a[n]$, где a — имя массива; n — число элементов массива.

Например, $a[10]$, где a — имя массива; 10 — число элементов в массиве.

Каждый элемент одномерного массива имеет один индекс, равный порядковому номеру элемента в массиве. Например, массив из 10 элементов выглядит так:

Индекс	1	2	3	4	5	6	7	8	9	10
Значение	3	0	15	4	6	–2	11	0	–9	7

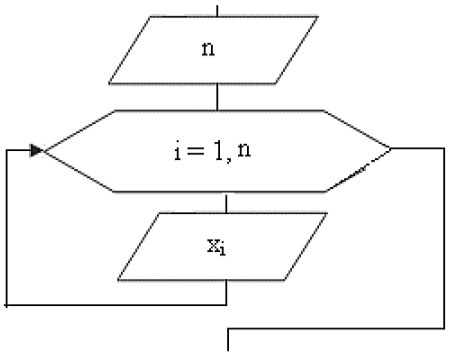
a[1] = 3; a[5] = 6; a[7] = 11; a[9] = –9; a[10] = 7.

Так как всегда известно количество элементов в массиве, то

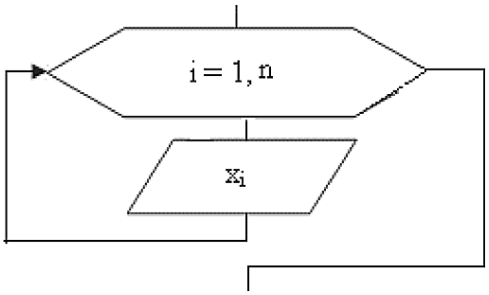
для ввода и вывода его элементов используется цикл с заданным числом повторений.

Мы рассматриваем массив, состоящий из произвольного числа элементов. Поэтому, прежде чем задать значения элементов массива, требуется ввести количество элементов массива *n*.

Вывод элементов также производится с использованием цикла с заданным числом повторений.



Ввод элементов массива



Вывод элементов массива

Здесь и далее будем обозначать через *i* — текущий индекс элемента массива. Он же будет являться параметром цикла, так как количество повторений цикла зависит от количества элементов в массиве.

Далее во всех задачах будем считать, что элементы массива заданы тем или иным способом, и показывать только реализацию алгоритма решения задачи, опуская команды ввода данных и вывода результата.

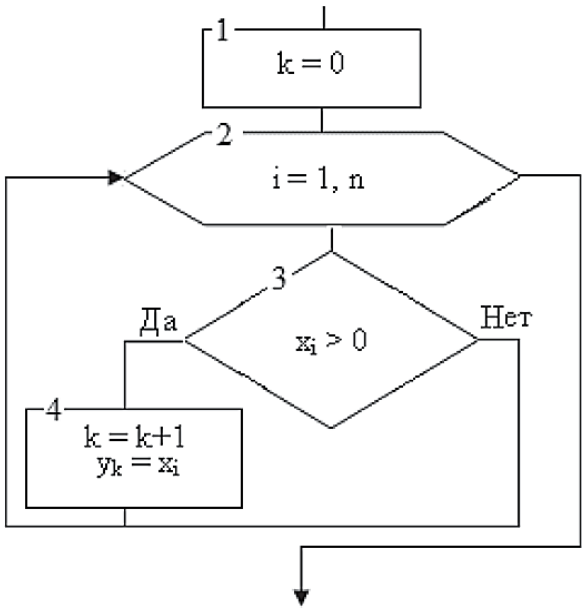
Также не будем обозначать блоки начала и конца выполнения алгоритма. Подразумевается, что все перечисленные блоки должны всегда присутствовать в алгоритме.

Пример. Сформировать новый одномерный массив из положительных элементов заданного массива.

Дано: *n* — размер массива произвольный; массив *X*[*n*]. *Найти*: массив *Y*[*k*].

Обозначим: *i* — текущий индекс элементов массива *X*, *k* — текущий индекс элементов массива *Y*.

Ясно, что для реализации алгоритма необходимо использовать цикл с заданным числом повторений, так как количество элементов в массиве *X* известно. Приведем фрагмент блок-схемы алгоритма и ее словесное описание.



Блок 1. $k = 0$.

Блок 2. В цикле для всех значений параметра i от 1 до n выполняется тело цикла.

Блок 3. Если условие $X[i] > 0$ выполняется (выход «Да»), то вычисляется группа операторов в блоке 4:

$k = k + 1$; $Y[k] = X[i]$.

Тест

Данные	**Результат**
$n = 5$ $X = (-1, 2, 0, 4, -3)$	$k = 2$ $Y = (2, 4)$

Выполнение алгоритма.

k	i	$x_i > 0$?	$y_k = x_i$
0	1	$x_1 > 0$? «Нет»	
	2	$x_2 > 0$? «Да»	
1			$y_1 = x_2 = 2$
	3	$x_3 > 0$? «Нет»	
	4	$x_4 > 0$? «Да»	
2			$y_2 = x_4 = 4$
	5	$x_5 > 0$? «Нет»	
	6	Конец цикла	

Мы используем цикл с заданным числом повторений. Напомним, что такой цикл (арифметический цикл) применяется, когда число повторений цикла известно к началу его выполнения.

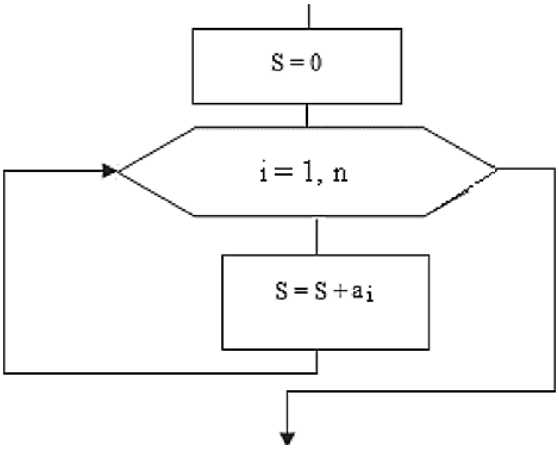
3.2 Вычисление суммы и количества элементов массива

Пример. Вычислить сумму элементов одномерного массива.

Дано: n — размер массива; массив $A = (a_1, a_2, \ldots, a_n)$.

Найти: S — сумму элементов массива.

Начальное значение суммы равно нулю. В предыдущих параграфах мы говорили о том, что значение суммы накапливается с каждым шагом выполнения алгоритма. Вычисляем сумму по формуле $S = S + a_i$.

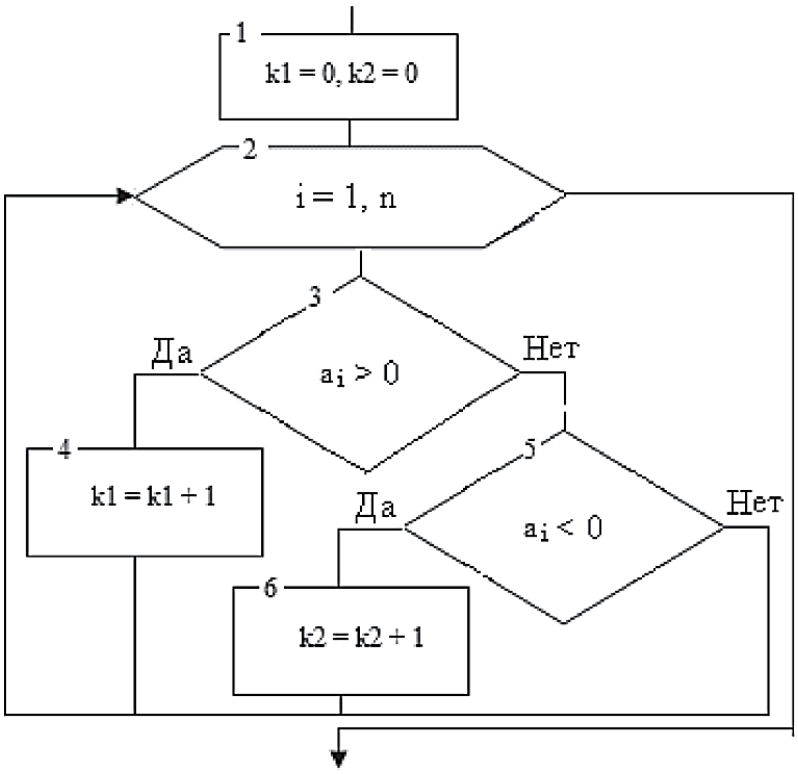


Пример. Найти количество положительных и отрицательных чисел в данном массиве.

Дано: n — размер массива; массив $A = (a_1, a_2, \ldots, a_n)$. Обозначим $k1$ — количество положительных чисел, $k2$ — количество отрицательных чисел.

Найти: $k1$, $k2$ — количество положительных и отрицательных чисел массива.

Математическая модель. Пусть $k1 = 0$ и $k2 = 0$. Если $a[i] > 0$, то $k1 = k1 + 1$. Если $a[i] < 0$, то $k2 = k2 + 1$. Процесс повторяется до окончания просмотра всех чисел массива. Приведем фрагмент блок-схемы алгоритма и ее словесное описание.



Блок 1. Задание начальных значений переменным $k1$ и $k2$.

Блок 2. Заголовок арифметического цикла.

Блок 3. Проверка условия. Если очередное значение элемента массива положительное (выход «Да» блока 3), то увеличиваем количество положительных чисел (блок 4).

Блок 5. Если очередное значение элемента массива отрицательное (выход «Да» блока 5), то увеличиваем количество отрицательных чисел массива (блок 6).

Указанные вычисления выполняем для всех n чисел массива.

В примере нам понадобилось два условных оператора, так как в массиве могут встретиться нулевые элементы, количество которых нам считать не надо.

3.3 Определение наибольшего элемента массива

Пример. Дан массив произвольной длины. Найти наибольший элемент массива и определить его номер.

Дано: n — размер массива; массив $A = (a_1, a_2, \ldots, a_n)$.

Найти: A_{max} — наибольший элемент массива, k — его номер.

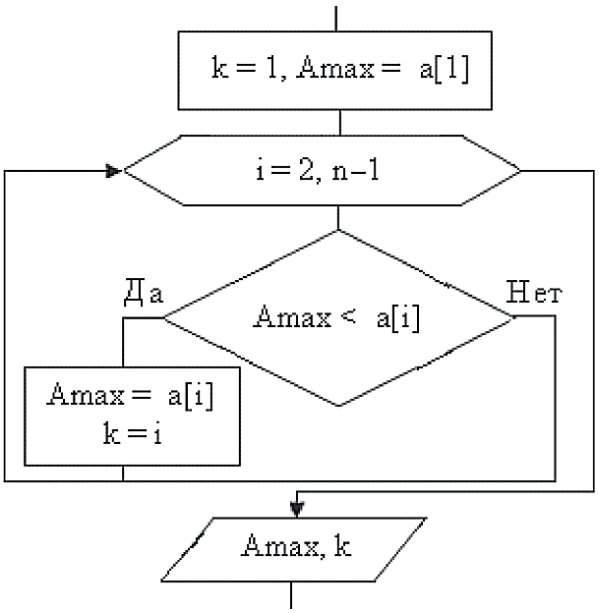
Математическая модель. Пусть $A_{max} = a[1]$; $k = 1$.

Если $A_{max} < a[i]$, то $A_{max} = a[i]$, $k = i$, для $i = 2, 3, \ldots, n$.

Тест

Данные		Результат	
$n = 5$	$A = (3, -1, 10, 1, 6)$	$A\max = 10$	$k = 3$

Приведем фрагмент блок-схемы алгоритма и его выполнение



для тестового примера.

i	$i \le n$?	$A_{\max} < A[i]$?	$A\max$	k
			3	1
2	2 ≤ 5? «Да»	–1 < 3? «Да»		
3	3 ≤ 5? «Да»	10 < 3? «Нет»	10	3
4	3 ≤ 5? «Да»	1 < 10? «Да»		
5	5 ≤ 5? «Да»	6 < 10? «Да»		
6	6 ≤ 5? «Нет» (кц)			

3.4 Определение первого элемента массива, имеющего заданное свойство

Пример. Определить, является ли заданная последовательность различных чисел $a_1, a_2, \ldots, a_n$ монотонно убывающей.

По определению последовательность монотонно убывает, если $a[i] \ge a[i + 1]$. Если хотя бы для одной пары чисел это условие нарушается, то последовательность не является монотонно убывающей. Следовательно, при построении алгоритма мы должны зафиксировать именно этот факт.

Пусть переменная *Flag* принимает значение равное 1, если

последовательность является монотонно убывающей, и 0 — в противном случае.

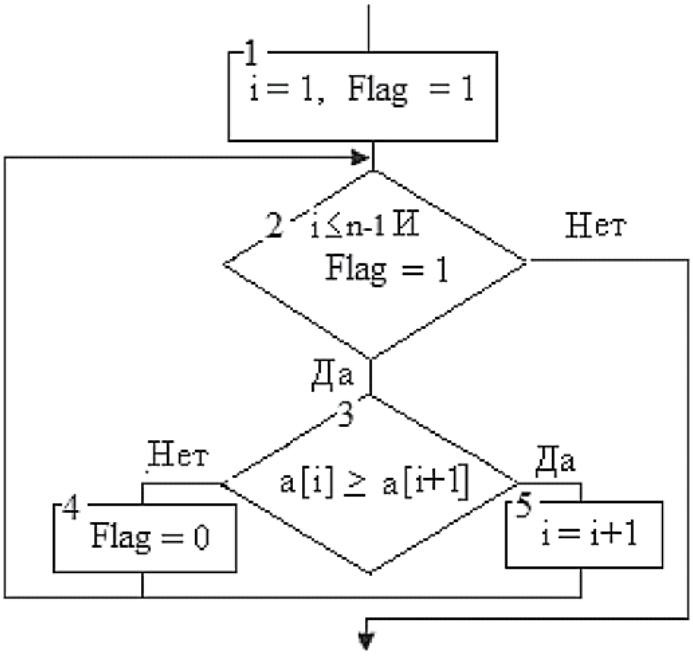
Дано: n — размер массива; $A[n]$ — числовой вещественный массив.

Найти: $Flag = 1$, если последовательность монотонно убывает; $Flag = 0$ в противном случае.

Словесное описание алгоритма. Предположим, что последовательность монотонно убывает, и присвоим переменной *Flag* начальное значение, равное 1. В цикле последовательно сравниваем значения двух соседних элементов. Выход из цикла возможен в двух случаях:

- просмотрены все элементы заданной последовательности. Это означает, что условие $a[i] \geq a[i + 1]$ выполняется для всех пар соединений элементов, и последовательность является монотонно убывающей. Тогда $Flag = 1$;
- условие $a[i] \geq a[i + 1]$ не выполнилось для пары сосед- них элементов, тогда $Flag = 0$. Цикл прерывается. Следовательно, последовательность не является монотонно убывающей.

Приведем фрагмент блок-схемы алгоритма и ее словесное описание.



Блок 1. Берем первое число последовательности $i = 1$. Предполагаем, что последовательность монотонно убывает ($Flag = 1$).

Блок 2. Цикл с предусловием. Пока не проверены все элементы

и пока последовательность монотонно убывает (проверка условия в блоке 2), выполняется тело цикла (блоки 3–5).

Блок 3. Если последовательность не монотонная (вы- ход «Нет»), то фиксируем это и $Flag = 0$ (блок 4).

Если условие не нарушается (выход «Да» блока 3), то присваиваем номеру элемента следующее значение (блок 5) и возвращаемся в цикл.

Система тестов

№ теста	**Проверяемый случай**	**Данные**		**Результат**
		n	**Массив *A***	***Flag***
1	Является	3	(3, 2, 1)	1
2	Не является	3	(2, 3, 1)	0

Выполнение алгоритма.

№ теста	***i***	***Flag***	**(*i* <= n – 1) и (*Flag* = 1)?**	***a*[*i*] > *a*[*i* + 1]?**
1	1	1	«Да»	*a*[1] > *a*[2]? «Да»
	2		«Да»	*a*[2] > *a*[3]? «Да»
	3		«Нет»	
			Последовательность монотонно убывает	
2	1	1	«Да»	*a*[1] > *a*[2]? «Нет»
		0	«Нет»	
			Последовательность не монотонно убывающая	

Пример. Определить, есть ли в одномерном массиве хотя бы один отрицательный элемент.

Пусть переменная $Flag = 1$, если в массиве есть отрицательный элемент, и $Flag = 0$ — в противном случае.

Дано: n — размер массива; $A[n]$ — числовой вещественный массив.

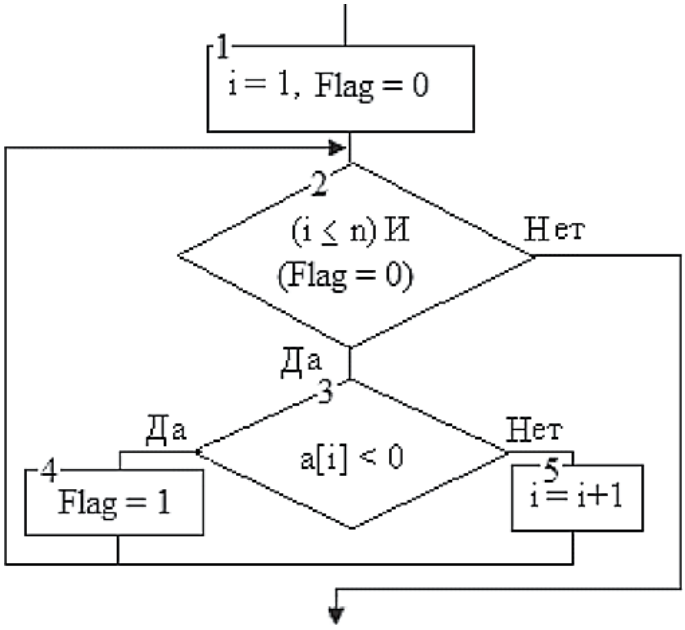
Найти: $Flag = 1$, если отрицательный элемент есть;

$Flag = 0$ — в противном случае.

Словесное описание алгоритма. Начинаем просматривать массив с первого элемента ($i = 1$). Пока не просмотрен последний элемент ($i \le n$) и не найден отрицательный эле- мент ($a[i] >= 0$), будем переходить к следующему элементу ($i = i + 1$). Таким образом, мы закончим просмотр массива в одном из двух случаев:

1) просмотрели все элементы и не нашли отрицательного, тогда $Flag = 0$ и $i > n$;

2) нашли нужный элемент, при этом $Flag = 1$. Приведем фрагмент блок-схемы алгоритма и ее словесное описание.



Блок 1. Берем первый элемент массива. Предполагаем, что в массиве нет отрицательных чисел и $Flag = 0$.

Блок 2. Пока не просмотрены все числа и пока не встретилось отрицательное число (выход «Да» блока 2) выполняем тело цикла (блоки 3–5).

Блок 3. Если встретилось отрицательное число (выход

«Да» блока 3), запоминаем это и $Flag = 1$. Если очередное число положительное или равно нулю (выход «Нет» блока 3), то увеличиваем номер числа (блок 5) и переходим к следующему числу.

Система тестов

№ теста	**Проверяемый случай**	**Данные**		**Результат**
		n	**Массив *A***	***Flag***
1	Есть	3	(3, –2, 1)	1
2	Нет	3	(2, 3, 1)	0

Исполнение алгоритма.

№ теста	i	$Flag$	$(i \le n)$ и $(Flag = 0)$?	$a[i] < 0$?
1	1	0	$(1 \le 3)$ и $(Flag = 0)$? «Да»	$a[1] < 0$? «Нет»
	2	0	$(2 \le 3)$ и $(Flag = 0)$? «Да»	$a[2] < 0$? «Да»
		1	Найдено отрицательное число (кц)	
2	1	0	$(1 \le 3)$ и $(Flag = 0)$? «Да»	$a[1] < 0$? «Нет»
	2	0	$(2 \le 3)$ и $(Flag = 0)$? «Да»	$a[2] < 0$? «Нет»
	3	0	$(3 \le 3)$ и $(Flag = 0)$? «Да»	$a[3] < 0$? «Нет»
	4	0	$(4 \le 3)$ и $(Flag = 0)$? «Нет» (кц)	
			Отрицательных чисел нет	

4. АЛГОРИТМЫ, ИСПОЛЬЗУЮЩИЕ ДВУМЕРНЫЕ МАССИВЫ

4.1 Понятие двумерного массива

Понятие «двумерный массив» определим на примере. Пусть имеются данные о продажах некоторого товара по месяцам:

Месяц	Объем продаж, пар	Цена продажи, руб.	Себестоимость, руб.
1	4500	100	50
2	3900	110	55
3	3100	120	60

Таблица представляет собой множество из двенадцати однородных величин — это массив. Ее элементы расположены в 3 строки по 4 столбца в каждой.

Подобного рода таблицы из нескольких строк с равным числом элементов в каждой называют в информатике двумерными массивами или матрицей.

Двумерный массив определяется именем, числом строк и столбцов и обозначается, например, так: $A[n, m]$, где A — произвольное имя массива; n — число строк; m — число столбцов. Обратите внимание на то, что сначала всегда указывается количество строк, а потом — количество столбцов массива.

Если имеются данные о продажах за 3 мес., то нашу таблицу можно обозначить так: $A[3, 4]$, т. е. массив состоит из 3 строк и 4 столбцов.

Строки двумерных массивов нумеруются по порядку сверху вниз, а столбцы — слева направо.

Элементы таблицы, представленной в примере, получат такие обозначения:

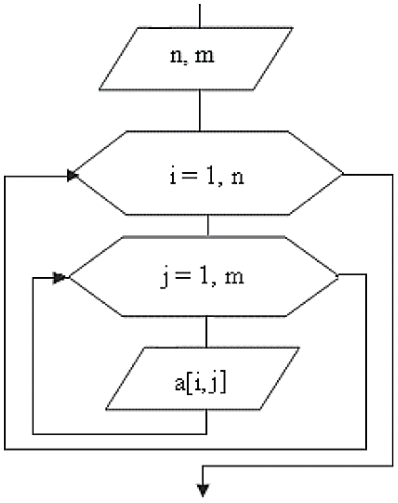
$$A[3,4] = \begin{pmatrix} A[1,1] & A[1,2] & A[1,3] & A[1,4] \\ A[2,1] & A[2,2] & A[2,3] & A[2,4] \\ A[3,1] & A[3,2] & A[3,3] & A[3,4] \end{pmatrix} = \begin{pmatrix} 1 & 4500 & 100 & 50 \\ 2 & 3900 & 110 & 55 \\ 3 & 3100 & 120 & 60 \end{pmatrix}.$$

Каждый элемент двумерного массива определяется номерами строки и столбца, на пересечении которых он находится, и в соответствии с этим обозначается именем массива с двумя индексами: первый — номер строки, второй — номер столбца. Например, $A[1, 3]$ — элемент находится в первой строке и третьем

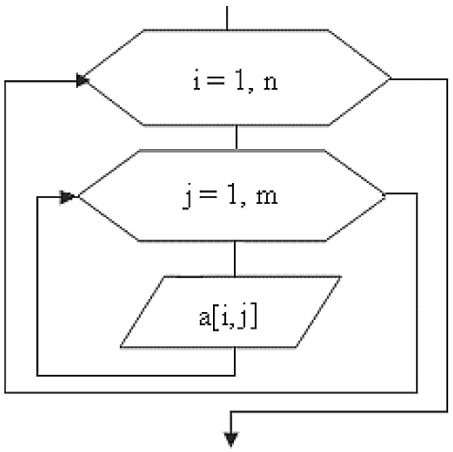
столбце.

Таким образом, $A[1, 1] = 1$, $A[2, 3] = 110$ и т. д. Произвольный элемент двумерного массива мы будем обозначать $A[i, j]$, где i — номер строки, j — номер столбца.

Поскольку положение элемента в двумерном массиве описывается двумя индексами, алгоритмы для решения большинства задач с их использованием строятся на основе вложенных циклов. Обычно внешний цикл организуется по строкам массива, т. е. в нем выбирается требуемая строка, а внутренний — по столбцам, в котором выбирается элемент внутри строки.



Ввод двумерного массива



Вывод двумерного массива

В отличие от одномерных массивов для ввода и вывода данных в двумерные массивы необходимо использовать вложенные циклы. Циклы являются арифметическими, так как количество строк и столбцов известно.

Мы ввели и вывели элементы произвольного массива, имеющего n строк и m столбцов. Понятно, что при вводе необходимо задать количество строк и столбцов.

Напомним, что во всех задачах будем считать, что элементы массива заданы тем или иным способом, и показывать только реализацию алгоритма решения задачи, опуская команды ввода данных и вывода результата.

Также не будем обозначать блоки начала и конца выполнения алгоритма. Подразумевается, что все перечисленные блоки должны всегда присутствовать в алгоритме.

Следует понимать, что все алгоритмы, приведенные ранее для

одномерных массивов, можно использовать и для двумерных массивов, применив вложенный цикл. Приведем алгоритмы решения некоторых задач.

4.2 Вычисление наибольшего элемента двумерного массива

Пример. Найти наибольший элемент двумерного массива и его индексы.

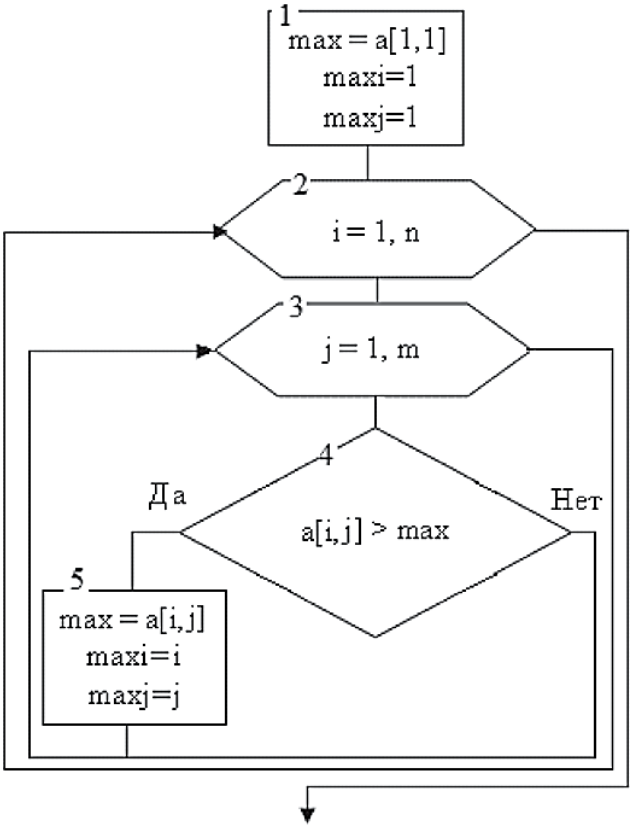
Дано: массив $A[n, m]$, где n, m — количество строк и столбцов массива соответственно.

Найти: max — наибольший элемент массива, а также max*i* и max*j* — номер строки и столбца соответственно, на пересечении которых находится искомый элемент.

Словесное описание алгоритма. Пусть первый элемент двумерного массива является наибольшим. Запоминаем его значение и индексы. Сравниваем наибольшее значение со всеми оставшимися элементами. Если запомненное значение меньше очередного элемента массива, то запоминаем новое значение и его индексы.

Так как значения элементов в массиве могут повторяться, то договоримся, что будем запоминать только индексы первого наибольшего элемента.

Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Присваиваем начальные значения переменным max, max*i*, max*j*.

Блок 2. Берем очередную строку матрицы.

Блок 3. На *i*-й строке матрицы последовательно выбираем каждый элемент.

Блок 4. Сравниваем выбранный элемент со значением переменной max.

Блок 5. Если значение очередного элемента больше max (выход «Да» блока 4), то запоминаем его значение и индексы. Возвращаемся в блок 3 и выбираем следующий элемент строки.

Блок 6. Если в строке нет элементов, то выбираем новую строку и повторяем вычисления (блоки 3–5).

Данные			**Результат**
n	*m*	*A*	
2	3	$\begin{pmatrix} 1 & 3 & 5 \\ 4 & 3 & 5 \end{pmatrix}$	$\max = 5$ $\max i = 1$ $\max j = 3$

Выполните тестирование алгоритма самостоятельно.

4.3 Нахождение строк или столбцов двумерного массива, обладающих заданным свойством

Пример. Дан двумерный массив *A*[*n*, *m*]. Найти количество строк, содержащих хотя бы один нуль.

Обозначим: *k* — количество строк, содержащих хотя бы один нуль; *Flag* — признак наличия нулей в строке. *Flag* = 1 означает, что нули в строке есть; $Flag = 0$ — нулей в строке нет.

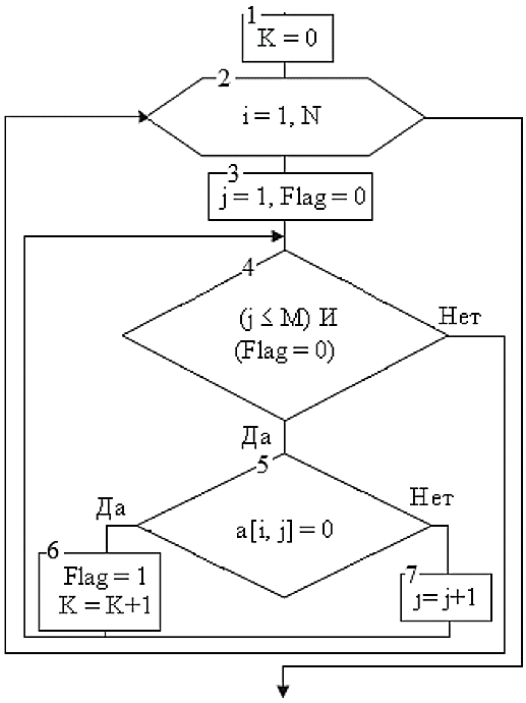
Словесное описание алгоритма. Начинаем просматривать массив с первой строки. Берем первый элемент столбца ($j = 1$). Пока не просмотрен последний элемент столбца ($j \le m$) и не найден отрицательный элемент ($a[i, j] >= 0$), будем переходить к следующему элементу, увеличивая индекс элемента *j* ($j = j + 1$). Таким образом, мы закончим просмотр строки массива в одном из двух случаев:

1) просмотрели все элементы и не нашли отрицательного, тогда $Flag = 0$;

2) нашли нужный элемент, при этом *Flag* = 1. Увеличиваем

значение количества строк *k*, содержащих нули.

Фрагмент блок-схемы алгоритма.



Блок 1. Присваиваем переменной *k* начальное значение.

Блок 2. Берем очередную строку *i*.

Блок 3. Для выбранной строки задаем первый элемент строки и устанавливаем $Flag = 0$.

Блок 4. Выполняем итерационный цикл для всех элементов строки (блоки 5–7).

Блок 5. Сравниваем очередной элемент строки с нулем.

Блок 6. Если нулевой элемент найден (выход «Да» блока 5), то $Flag = 1$, увеличиваем значение *k* и заканчиваем просмотр строки (выход «Нет» блока 4).

Блок 7. Если нулевой элемент не найден (выход «Нет» блока 5), то продолжаем просмотр оставшихся элементов в строке, увеличивая индекс элемента *j*.

Покажем выполнение алгоритма на тестовом примере.

Данные			**Результат**
N	*M*	Матрица *A*	*K*
3	3	$\begin{pmatrix} 1 & 0 & 1 \\ 2 & 4 & 7 \\ 0 & 1 & 0 \end{pmatrix}$	2

Исполнение алгоритма.

i	***Flag***	***j***	**(*j* <= *M*) и (*Flag* = 0)?**	***A*[*i*, *j*] = 0?**	**K**
1	0				0
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[1, 1] = 0? «Нет»	
		2	(2 <= 3) и (*Flag* = 0)? «Да»	*A*[1, 2] = 0? «Да»	
	1				1
			(2 <= 3) и (*Flag* = 0)? «Нет» (кц)		
2	0				
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 1] = 0? «Нет»	
		2	(2 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 2] = 0? «Нет»	
		3	(3 <= 3) и (*Flag* = 0)? «Да»	*A*[2, 3] = 0? «Нет»	
	0	4	(4 <= 3) и (*Flag* = 0)? «Нет» (кц)		
3	0				
		1	(1 <= 3) и (*Flag* = 0)? «Да»	*A*[3, 1] = 0? «Да»	2
			(4 <= 3) и (*Flag* = 0)? «Нет» (кц)		

Итак, количество строк, в которых встречается хотя бы один нуль, равно 2, что соответствует условию задачи.

5. АЛГОРИТМЫ СОРТИРОВКИ

Сортировка — это процесс перестановки элементов некоторого заданного множества в определенном порядке. Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве.

В общем случае задача сортировки ставится следующим образом. Имеется массив, тип данных которого позволяет использовать операции сравнения («=», «>», «<», «>=» и

«<=»). Задачей сортировки является преобразование исходного массива в массив, содержащий те же элементы, но в порядке возрастания (или убывания) значений.

Методы сортировки не должны требовать дополнительной памяти: все перестановки с целью упорядочения элементов массива должны производиться в пределах того же массива.

Будем выполнять сортировку по возрастанию.

Методы, сортирующие массив «на месте», делятся на три основных класса в зависимости от лежащего в их основе приема.

1. Сортировка включениями (вставкой).
2. Сортировка выбором.
3. Сортировка обменом.
4.

5.1 Сортировки включениями (вставкой)

Сортировка простой вставкой

Идея алгоритма очень проста. Пусть имеется массив $a[1]$, $a[2]$, ..., $a[n]$. Пусть элементы $a[1]$, $a[2]$, ..., $a[i-1]$ уже отсортированы, и пусть имеем входную последовательность $a[i]$, $a[i+1]$, ..., $a[n]$. На каждом шаге, начиная с $i = 2$ и увеличивая i на единицу, берем i-й элемент вход- ной последовательности и вставляем его на подходящее место в уже отсортированную часть последовательности.

Обозначим вставляемый элемент через x. Пусть начальный массив 32 64 9 30 87 14 2 76.

$i = 2$ $x = 64$. Ищем для x подходящее место, считая, что $a[1] = 32$ — это уже отсортированная часть последовательности. Получаем *32 64* **9** 30 87 14 2 76.

$i = 3$ $x = 9$. Часть последовательности $a[1]$, $a[2]$ уже

отсортирована. Ищем для *x* подходящее место: *9 32 63* **30** 87 14 2 76.

Аналогично выполняем последующие шаги сортировки.

$i = 4$ $x = 30$. Ищем для *x* подходящее место:

9 30 32 64 **87** 14 2 76.

$i = 5$ $x = 87$. Ищем для *x* подходящее место:

9 30 32 64 87 **14** 2 76.

$i = 6$ $x = 14$. Ищем для *x* подходящее место:

9 14 30 32 64 87 **2** 76.

$i = 7$ $x = 2$. Ищем для *x* подходящее место:

2 9 14 30 32 64 87 **76.**

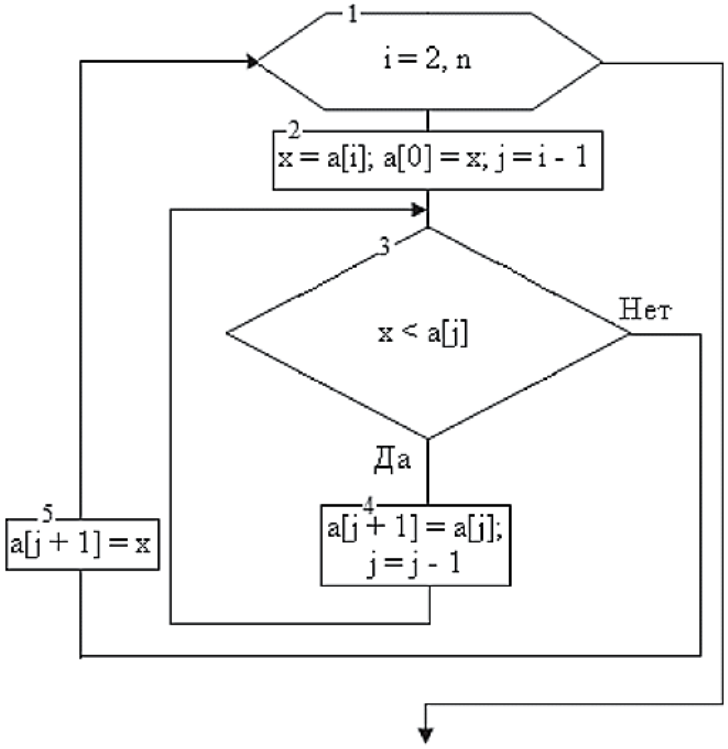
$i = 8$ $x = 76$. Ищем для *x* подходящее место:

2 9 14 30 32 64 76 **87.**

При поиске подходящего места для элемента *x* мы чередовали сравнения и пересылки, т. е. как бы «просеивали» *x*, сравнивая его с очередным элементом *a*[*i*] и либо вставляя *x*, либо пересылая *a*[*i*] направо и продвигаясь влево. Заметим, что «просеивание» может закончиться при двух различных условиях:

1) найден элемент, значение которого больше, чем *x*;
2) достигнут конец последовательности.

Это типичный пример цикла с двумя условиями окончания. При реализации алгоритма с целью окончания выполнения внутреннего цикла используют прием фиктивного элемента (барьера). Его можно установить, приняв *a*[0] = *x*. Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Начинаем цикл для перебора всех элементов исходного массива.

Блок 2. Присваиваем x значение очередного элемента массива. Устанавливаем барьер и продвигаемся назад по отсортированной части массива.

Блок 3. Начинам цикл для поиска подходящего места для значения x.

Блок 4. Пока x меньше очередного элемента (выход

«Да» блока 3), выполняем сдвиг и продвигаемся к началу отсортированной части массива.

Блок 5. По выходу «Нет» блока 3 вставляем элемент x на нужное место.

Покажем исполнение алгоритма для рассмотренного примера.

i	x	$a[0]$	j	$x < a[j]$?	$a[j+1]$	Массив
2	64	64	1	64 < 32? «Нет»	$a[2] = x = 64$	32 64 9 30 87 14 2 76
3	9	9	2	9 < 64? «Да»	$a[3] = a[2] = 64$	32 64 64 30 87 14 2 76
			1	9 < 32? «Да»	$a[2] = a[1] = 32$	32 32 64 30 87 14 2 76
			0	9 < 9? «Нет»	$a[1] = x = 9$	9 32 64 30 87 14 2 76
4	30	30	3	30 < 64? «Да»	$a[4] = a[3] = 64$	9 32 64 64 87 14 2 76
			2	30 < 32? «Да»	$a[3] = a[2] = 32$	9 32 32 64 87 14 2 76
			1	30 < 9? «Нет»		$a[2] = x = 30$

После этого шага массив выглядит так: 9 30 32 64 87 14 2 76. Сортировка продолжается далее для оставшихся значений i.

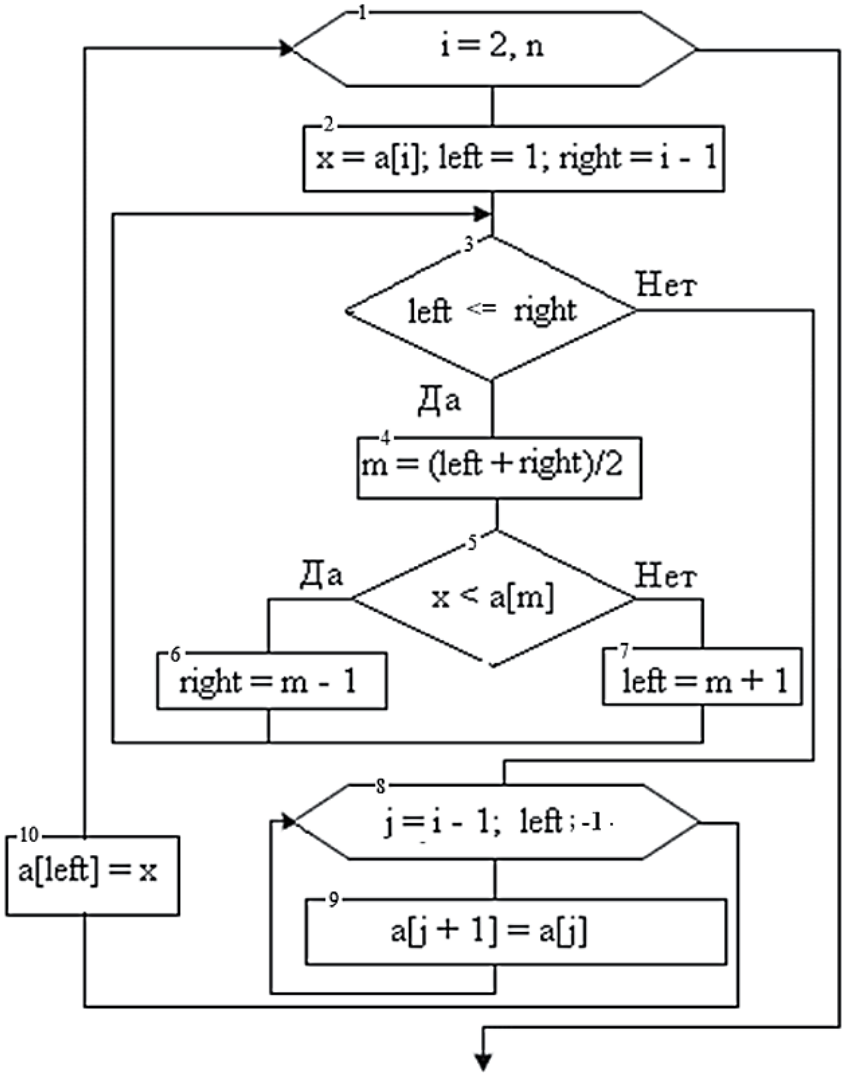
Сортировка бинарными включениями

Алгоритм сортировки простыми включениями имеет слабые места. Это, во-первых, необходимость перемещения данных, причем при вставке элементов, близких к концу массива, приходится перемещать почти весь массив. Второй недостаток — это необходимость поиска места для вставки, на что также тратится много ресурсов. Эту часть алгоритма можно улучшить, применив так называемый бинарный поиск. Этот метод на каждом шаге сравнивает x со средним (по положению) элементом отсортированной последовательности до тех пор, пока не будет найдено место включения.

Модифицированный алгоритм называется сортировкой бинарными включениями.

Обозначим *left* — левая граница отсортированного массива, *right* — его правая граница.

Приведем фрагмент блок-схемы алгоритма и ее описание.



Блок 1. Начинаем цикл для перебора всех элементов исходного массива.

Блок 2. Присваиваем *x* значение очередного элемента массива. Устанавливаем левую и правую границы отсортированной части массива.

Блок 3. Пока левая граница отсортированного массива не превосходит правую, выполняется тело цикла, состоящее из блоков 4–7.

Блок 4. Находится средний по положению элемент отсортированной части массива.

Блок 5. Значение *x* сравнивается с найденным элементом. По выходу «Да» блока 5 корректируется правая граница отсортированной части массива, по выходу «Нет» — левая.

При выходе из цикла с предусловием будет найдено положение вставляемого элемента.

Блоки 8–9 реализуют сдвиг элементов массива для вставки значения x.

Блок 10. Вставка значения в отсортированную часть массива.

Приведем выполнение алгоритма при сортировке массива 32 64 9 30 87 14 2 76.

i	x	$left$	$right$	$left <= right$?	m	$x < a[m]$?	j	$a[j+1]$	$a[left]$
2	64	1	1	«Да»	1	32 < 64? «Да»			
			0	«Нет»			1	$a[2] = 64$	$a[1] = 32$
3	9	1	2	«Да»	1	9 < 64? «Да»			
			1	«Нет»			2	$a[3] = a[2] = 64$	
							1	$a[2] = a[1] = 32$	
									$a[1] = 9$
4	30	1	3	«Да»	2	30 < 32? «Да»			
			2	«Да»	1	30 < 9? «Нет»			
		2		«Нет»			3	$a[4] = a[3] = 64$	
							2	$a[3] = a[2] = 32$	$a[2] = 30$

После этого шага массив выглядит так: 9 30 32 64 87 14 2 76. Сортировка массива продолжается для следующих значений i.

5.2 Сортировка простым выбором

Сортировка простым выбором основана на следующем правиле.

В неупорядоченном массиве выбирается и отделяется от остальных элементов наименьший элемент. Наименьший элемент записывается на i-е место исходного массива, а элемент с i-го места — на место выбранного. Уже упорядоченные элементы (а они будут расположены начиная с первого места) исключаются из дальнейшей сортировки, поэтому длина оставшегося неупорядоченного массива должна быть на один элемент меньше предыдущего.

Улучшенный алгоритм — пирамидальная сортировка требует организации массива виде дерева.

Сортировку простым выбором продемонстрируем на массиве 32 64 9 30 87 14 2 76.

$i = 1$, наименьшее значение 2, меняем местами 2 и 32.

Массив после этого шага:

2 64 9 30 87 14 32 76.

i = 2, наименьшее значение в оставшейся части массива 9, меняем местами 9 и 64. Массив после этого шага:

2 9 64 30 87 14 32 76.

i = 3, наименьшее значение в оставшейся части массива 14, меняем местами 14 и 64. Массив после этого шага:

2 9 14 30 87 64 32 76.

i = 4, наименьшее значение в оставшейся части массива 30. Обмен не происходит, так как 30 стоит на своем месте.

i = 5, наименьшее значение в оставшейся части массива 32, меняем местами 32 и 87. Массив после этого шага:

2 9 14 30 32 64 87 76.

i = 6, наименьшее значение в оставшейся части массива 64. Обмен не происходит, так как 64 стоит на своем месте.

i = 7, наименьшее значение в оставшейся части массива 76, меняем местами 76 и 87. Массив после этого шага:

2 9 14 30 32 64 76 87.

Массив отсортирован, но выполняется еще один шаг.

i = 8, наименьшее значение в оставшейся части массива 87. Обмен не происходит, так как 87 стоит на своем месте.

Таким образом, мы видим, что метод сортировки простым выбором основан на повторном выборе наименьшего элемента сначала среди *n* элементов, затем среди (n – 1)-го элемента и т. д. Возможна ситуация, когда обмен значений элементов не происходит.

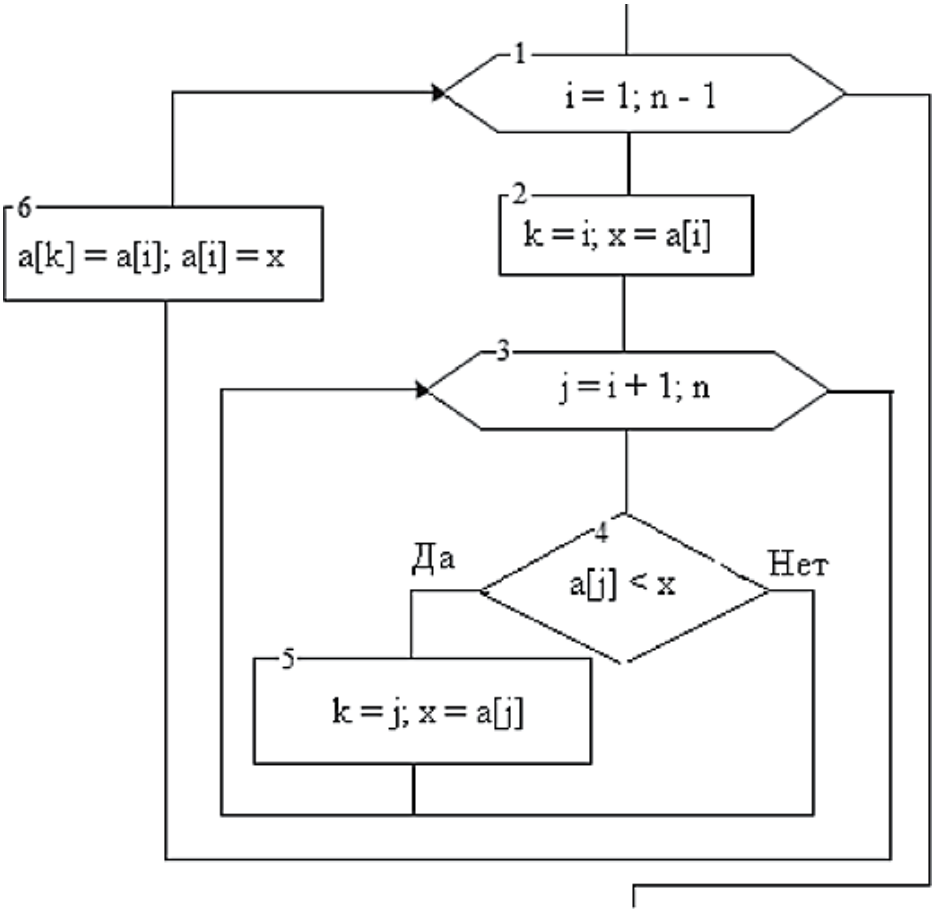
Приведем блок-схему и описание алгоритма сортировки простым выбором.

Блок 1. Организует внешний цикл для просмотра элементов неупорядоченной части массива.

Блок 2. Запоминание индекса и значения *i*-го элемента, которые принимаем за начальные при поиске наименьшего элемента.

Блоки 3–5. Поиск значения наименьшего элемента и его номера в еще неупорядоченной части массива.

Блок 6. Обмен наименьшего и *i*-го элементов.



Приведем исполнение алгоритма для массива: 32 64 9 30 87 14 2 76; $n = 8$.

i	***k***	***x***	***j***	***a*[*j*] < *x*?**	***a*[*k*] = *a*[*i*]; *a*[*i*] = *x***
1	1	32	2	*a*[2] < 32? «Нет»	
	3	9	3	*a*[3] < 32? «Да»	
			4	a[4] < 9? «Нет»	
			5	a[5] < 9? «Нет»	
			6	a[6] < 9? «Нет»	
	7	2	7	a[7] < 9? «Да»	*A*[7] = *a*[1] = 32; *a*[1] = 2
			кц		

После окончания внутреннего цикла массив имеет вид 2 64 9 30 87 14 32 76.

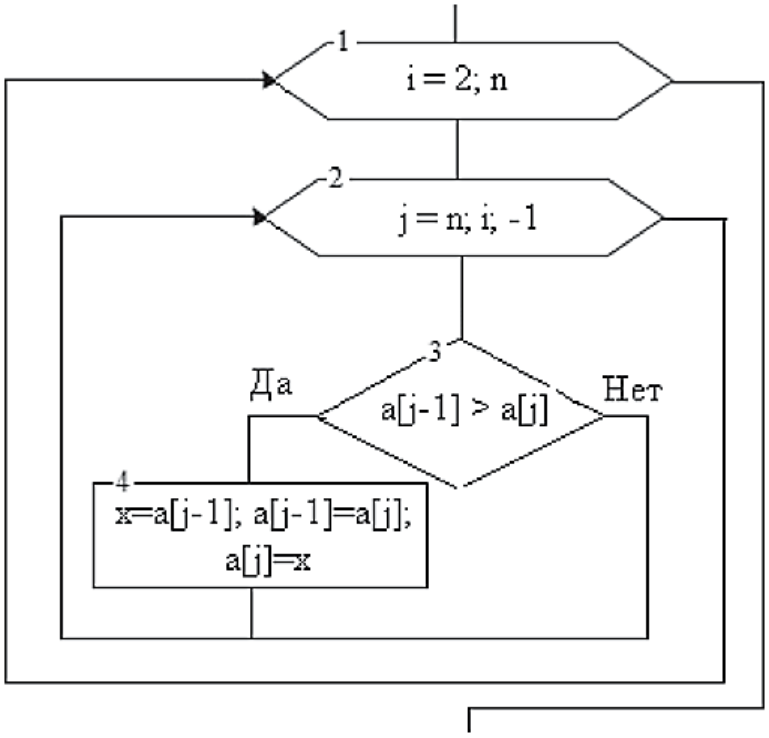
Сортировка продолжается для всех оставшихся значений параметра внешнего цикла *i*.

5.3 Обменные сортировки

Сортировка простым обменом

Простая обменная сортировка (называемая также «методом пузырька») для массива $a[1]$, $a[2]$, ..., $a[n]$ работает следующим образом. Начиная с конца массива, сравниваем два соседних элемента ($a[n]$ и $a[n-1]$). Если выполняется условие $a[n-1] > a[n]$, то значения элементов меняются местами. Процесс продолжается для $a[n-1]$ и $a[n-2]$ и т. д., пока не будет произведено сравнение $a[2]$ и $a[1]$. Понятно, что после этого на месте $a[1]$ окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются $a[3]$ и $a[2]$. На последнем шаге будут сравниваться только значения $a[n]$ и $a[n-1]$. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые «легкие») постепенно «всплывают» к верхней границе массива.

Простая реализация алгоритма.



Блок 1. Арифметический цикл. Значение параметра цикла i определяет количество сравниваемых элементов во внутреннем цикле.

Блок 2. Задание номеров элементов для сравнения.

Блок 3. Сравнение двух соседних элементов.

Блок 4. Выполняется обмен элементов массива при выходе

«Да» блока 3.

Сортируем массив 32 64 9 30 87 14 2.

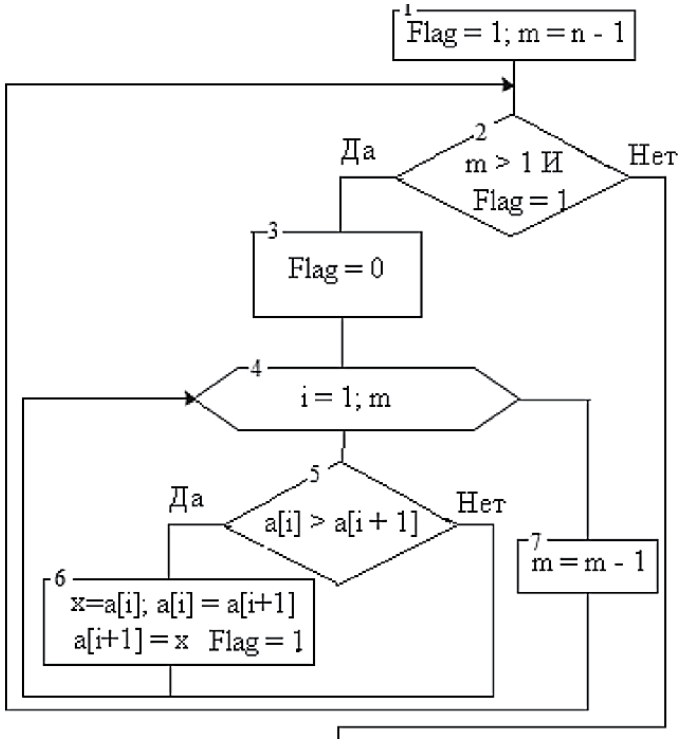
Выполнение алгоритма для $i = 2$ приведено ниже.

i	j	$a[j-1] > a[j]$?	$a[j-1], a[j]$	Массив
2	8	2 > 76? «Нет»		32 64 9 30 87 14 2 76
	7	14 > 2? «Да»	$a[6] = 2$, $a[7] = 14$	32 64 9 30 87 **2** 14 76
	6	87 > 2? «Да»	$a[5] = 2$, $a[6] = 87$	32 64 9 30 **2** 87 14 76
	5	30 > 2? «Да»	$a[4] = 2$, $a[5] = 30$	32 64 9 **2** 30 87 14 76
	4	9 > 2? «Да»	$a[3] = 9$, $a[4] = 2$	32 64 **2** 9 30 87 14 76
	3	64 > 2? «Да»	$a[2] = 2$, $a[3] = 64$	32 **2** 64 9 30 87 14 76
	2	32 > 2? «Да»	$a[1] = 2$, $a[2] = 32$	**2** 32 64 9 30 87 14 76

После этого шага значение 2 встало на свое место, и алгоритм повторяется для $i = 3, 4, ..., n$.

Очевидный способ улучшить алгоритм — это запоминать, производился ли на данном проходе какой-либо обмен. Если нет, то алгоритм может закончить свою работу. Используем переменную *Flag*, и пусть переменная $Flag = 1$, если на очередном шаге был выполнен обмен элементов, и $Flag = 0$, если обмена не было.

Приведем фрагмент блок-схемы и ее описание.



Блок 1. Задаем начальные значения $Flag = 1$ и $m = n - 1$.

Блок 2. Заголовок цикла. Пока не просмотрен весь массив и пока был обмен, выполняем внутренний цикл.

Блок 3. Предположим, что элементы массива уже отсортированы, тогда $Flag = 0$.

Блок 4. Организация арифметического цикла для сравнения и обмена соседних элементов массива.

Блоки 5–6. Тело внутреннего цикла. В блоке 5 выполняется сравнение соседних элементов.

При выходе «Да» блока 5 меняем местами элементы и фиксируем факт обмена в переменной *Flag*.

Блок 7. По окончании внутреннего цикла очередной элемент встал на свое место, уменьшается размер еще не отсортированной части массива.

Если был хотя бы один обмен, алгоритм повторяется. Выполним алгоритм и отсортируем массив 32 64 9 30 87 14 2.

Flag	***m***	***m* > 1 и *Flag* = 1?**	***i***	***a*[*i*] > *a*[*i* + 1]**	***a*[*i*]**	***a*[*i* + 1]**
1	6	«Да»				
0			1	32 > 64? «Нет»		
1			2	64 > 9? «Да»	*a*[2] = 9	*a*[3] = 64
1			3	64 > 30? «Да»	*a*[3] = 30	*a*[4] = 64
			4	64 > 87? «Нет»		
1			5	87 > 14? «Да»	*a*[5] = 14	*a*[6] = 87
1			6	87 > 2? «Да»	*a*[6] = 2	*a*[7] = 87

В результате получим следующий массив: 32 9 30 64 14 2 87.

По выходе из внутреннего цикла $m = 6$ и $Flag = 1$, так как выполнялись обмены элементов массива. Начинаем внутренний цикл для нового значения переменной m.

Последний элемент находится на своем месте. Продолжим сортировку для $m = 5$.

Flag	***m***	***m* > 1 и *Flag* = 1?**	***i***	***a*[*i*] > *a*[*i* + 1]?**	***a*[*i*]**	**a[i+ 1]**
1	5	«Да»				
0			1	32 > 9? «Да»	*a*[1] = 9	a[2] = 32
1			2	32 > 30? «Да»	*a*[2] = 30	a[3] = 32
1			3	32 > 64? «Нет»		
			4	64 > 14? «Да»	*a*[4] = 14	a[5] = 64
1			5	64 > 2? «Да»	*a*[5] = 2	a[6] = 64
1			6	64 > 87? «Нет»		

Теперь массив имеет вид 9 30 32 14 2 64 87, и два последних элемента находятся на своих местах.

Так как были перестановки элементов, то алгоритм продолжается до полной сортировки массива.

Второе улучшение алгоритма связано с тем, что можно установить барьер: запоминать наименьшее значение индекса массива, для которого на текущем шаге выполнялись перестановки. Очевидно, что верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса.

В-третьих, метод пузырька работает неравноправно для так называемых «легких» и «тяжелых» значений. Один неправильно расположенный «пузырек» в «тяжелом» конце рассортированного массива всплывет на место за один проход, а неправильно расположенный элемент в «легком» конце будет опускаться на правильное место только за один шаг на каждом проходе. Например, массив 12 18 42 44 55 67 94 6 будет рассортирован за один проход, а сортировка массива 94 6 12 18 43 44 55 76 потребует семи проходов. Эта асимметрия подсказывает третье улучшение: менять местами направление следующих один за другим проходов. Этот улучшенный алгоритм называется шейкер-сортировкой.

Шейкер-сортировка

При ее применении на каждом следующем шаге меняется направление последовательного просмотра. В результате на одном шаге «всплывает» очередной наиболее легкий элемент, а на другом «тонет» очередной самый тяжелый.

Обозначим *left* — номер левой границы сортируемой части массива, *right* — номер его правой границы.

Блок 1. Установка номеров начальных границ сортируемого массива.

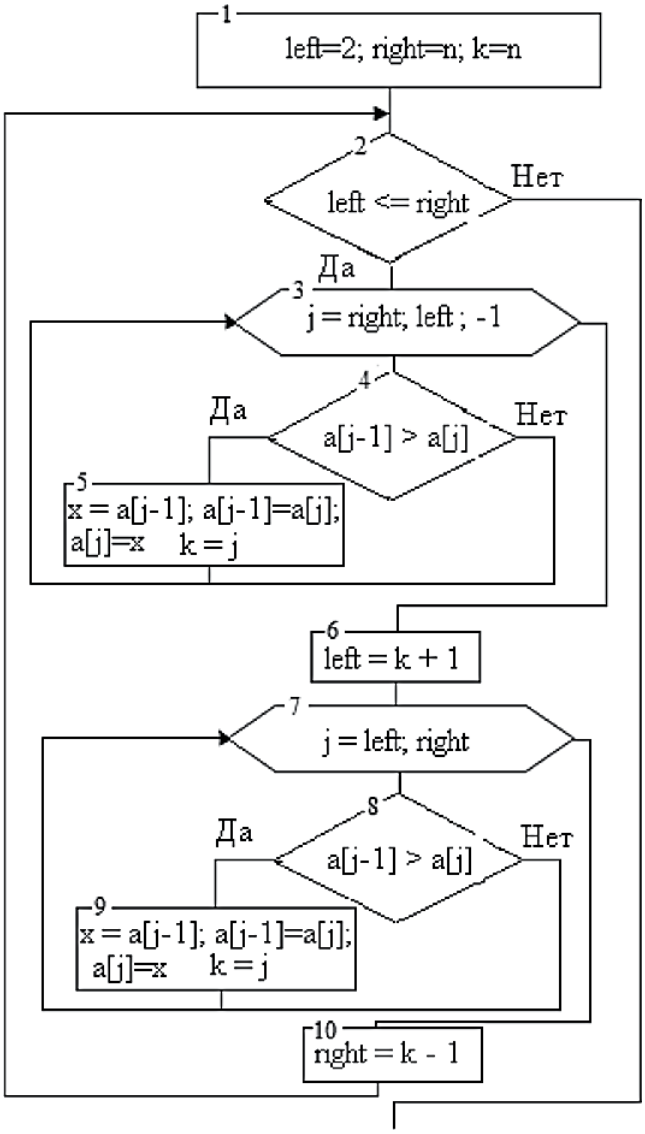
Блок 2. Вход в цикл. Пока левая граница не превосходит правую границу (выход «Да» блока 2) выполняем цикл.

Блок 3. Выполнение прохода массива вниз.

Блок 4. Сравнение соседних элементов.

Блок 5. Если $a[j-1] > a[j]$ (выход «Да» блока 4), производим обмен этих элементов и фиксируем номер элемента j, с которым производился обмен.

По окончании прохода вниз (выход блока 3) сдвигаем левую границу массива (блок 6) и выполняем проход вверх (блоки 7–9).



Покажем выполнение алгоритма для массива, состоящего из 7 элементов: 32 9 30 64 14 2 87.

Выполняем проход сверху вниз.

left	***right***	***k***	***left* <= *right*?**	***j***	***j* > *left*?**	***a*[*j* – 1] > *a*[*j*]?**	**Обмен**
2	7	7	«Да»	7	«Да»	2 > 87 «Нет»	
		6		6	«Да»	14 > 2 «Да»	$x = 14$; $a[5] = 2$; $a[6] = 14$
		5		5	«Да»	64 > 2 «Да»	$x = 64$; $a[4] = 2$; $a[5] = 64$
		4		4	«Да»	30 > 2 «Да»	$x = 30$; $a[3] = 2$; $a[4] = 30$
		3		3	«Да»	9 > 2 «Да»	$x = 9$; $a[2] = 2$; $a[3] = 9$
		2		2	«Да»	32 > 2 «Да»	$x = 32$; $a[1] = 2$; $a[2] = 32$
				1	«Нет»		

Массив после этого прохода имеет вид 2 32 9 30 64 14 87.

Меняем направление движения и выполняем проход снизу вверх.

left	***right***	***k***	***j***	***j* <= *right***	***a*[*j* – 1] > *a*[*j*]?**	
3	7	3	3	«Да»	32 > 9? «Да»	$x = 32$; $a[2] = 9$; $a[3] = 32$
		4	4	«Да»	32 > 30? «Да»	$x = 32$; $a[3] = 30$; $a[4] = 32$
			5	«Да»	32 > 64? «Нет»	
		6	6	«Да»	64 > 14? «Да»	$x = 64$; $a[5] = 14$; $a[6] = 64$
			7	«Да»	64 > 87? «Нет»	
			8	«Нет»		

Теперь массив имеет вид 2 9 30 32 14 64 87.

Опять меняем направление движения. Смена направления движения выполняется до тех пор, пока выполняется условие, записанное в блоке 2.

Анализ показывает, что сортировка обменом и ее небольшие улучшения хуже, чем сортировка включениями и выбором. Алгоритм шейкер-сортировки рекомендуется использовать в тех случаях, когда известно, что массив «почти упорядочен».

5.4 Сравнение методов сортировки

Можно сделать следующие выводы об эффективности рассмотренных алгоритмов сортировки.

1. Преимущество сортировки бинарными включениями по сравнению с сортировкой простыми включениями мало, а в случае уже имеющегося порядка вообще отсутствует.

2. Сортировка методом «пузырька» является наихудшей среди сравниваемых методов. Ее улучшенная версия — шейкер-сортировка — все-таки хуже, чем сортировка простыми включениями и простым выбором.

3. Сортировка простым выбором является лучшим из простых методов.

6. АЛГОРИТМЫ ПОИСКА

Алгоритмы поиска занимают очень важное место среди прикладных алгоритмов, и это утверждение не нуждается в доказательстве.

Все алгоритмы поиска разбиваются на две большие группы в зависимости от того, упорядочен или нет массив данных, в котором проводится поиск. Рассмотрим простые алгоритмы поиска заданного элемента в одномерном массиве данных.

6.1 Последовательный поиск

Наиболее примитивный, а значит, наименее эффективный способ поиска — это обычный последовательный просмотр массива.

Пусть требуется найти элемент X в массиве из n элементов. Значение элемента X вводится с клавиатуры.

В данном случае известно только значение разыскиваемого элемента, никакой дополнительной информации о нем или о массиве, в котором его надо искать, нет. Поэтому для решения задачи разумно применить очевидный метод — последовательный перебор элементов массива и сравнение значения очередного элемента с заданным образцом.

Пусть $Flag = 1$, если значение, равное X, в массиве найдено, в противном случае $Flag = 0$. Обозначим k — индекс найденного элемента. Элементы массива — целого типа.

Элементы массива

Присвоим начальные значения переменным $Flag$ и k

и возьмем первый элемент массива (блок 1).

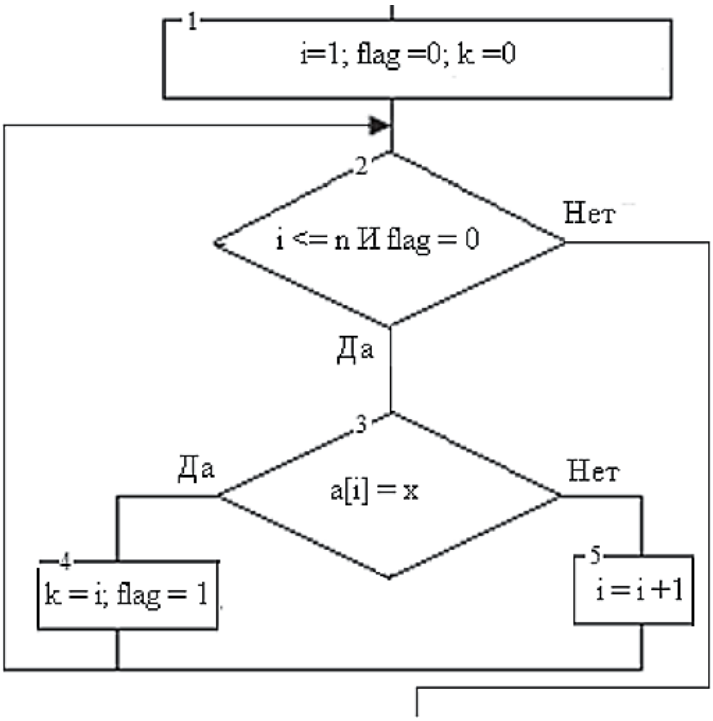
Пока не просмотрены все элементы и пока элемент, равный X, не найден (выход «Да» блока 2), выполняем цикл.

Сравниваем очередной элемент массива со значением X, и если они равны (выход «Да» блока 3), то запоминаем его индекс и поднимаем $Flag = 1$. Если элемент еще не найден (выход «Нет» блока 2), продолжаем просмотр элементов массива.

Выход из цикла (блок 2) возможен в двух случаях:

1) элемент найден, тогда $Flag = 1$, а в переменной k сохранен индекс найденного элемента;

2) элемент не найден, тогда $Flag = 0$, а k также равен 0.



6.2 Бинарный поиск

Если известно, что массив упорядочен, то можно использовать бинарный поиск.

Пусть даны целое число X и массив размера n, отсортированный в порядке неубывания, т. е. для любого k ($1 <= k < n$) выполняется условие $a[k - 1] <= a[k]$. Требуется найти такое i, что $a[i] = X$, или сообщить, что элемента X нет в массиве.

Идея бинарного поиска состоит в том, чтобы проверить, является ли X средним по положению элементом массива. Если да, то ответ получен. Если нет, то возможны два случая.

1. X меньше среднего по положению элемента, следовательно, в силу упорядоченности массива можно исключить из рассмотрения все элементы массива, расположенные правее этого элемента, так как они заведомо больше его, который, в свою очередь, больше X, и применить этот метод к левой половине массива.

2. X больше среднего по положению элемента, следовательно, рассуждая аналогично, можно исключить из рассмотрения левую половину массива и применить этот метод к его правой части.

Средний по положению элемент и в том, и в другом случае в дальнейшем не рассматривается. Таким образом, на каждом шаге отсекается та часть массива, где заведомо не может быть обнаружен

элемент *X*.

Пусть *X* = 6, а массив состоит из 10 элементов: 3 5 6 8 12 15 17 18 20 25.

1-й шаг. Найдем номер среднего по положению эле- мента: *m* = [(1 + 10) / 2] = 5. Так как 6 < *a*[5], то далее можем рассматривать только элементы, индексы которых меньше 5. Об остальных элементах можно сразу сказать, что они больше, чем *X*, вследствие упорядоченности массива и среди них искомого элемента нет.

3 5 6 8 **12** 15 17 18 20 25.

2-й шаг. Сравниваем лишь первые 4 элемента массива; значение *m* = [(1 + 4) / 2] = 2. 6 > *a*[2], следовательно, рассматриваем правую часть подмассива, а первый и второй элементы из рассмотрения исключаются:

3 **5** *6 8* 12 15 17 18 20 25.

3-й шаг. Сравниваем два элемента; значение

m = [(3 + 4) / 2] = 3:

3 5 **6** *8* 12 15 17 18 20 25.

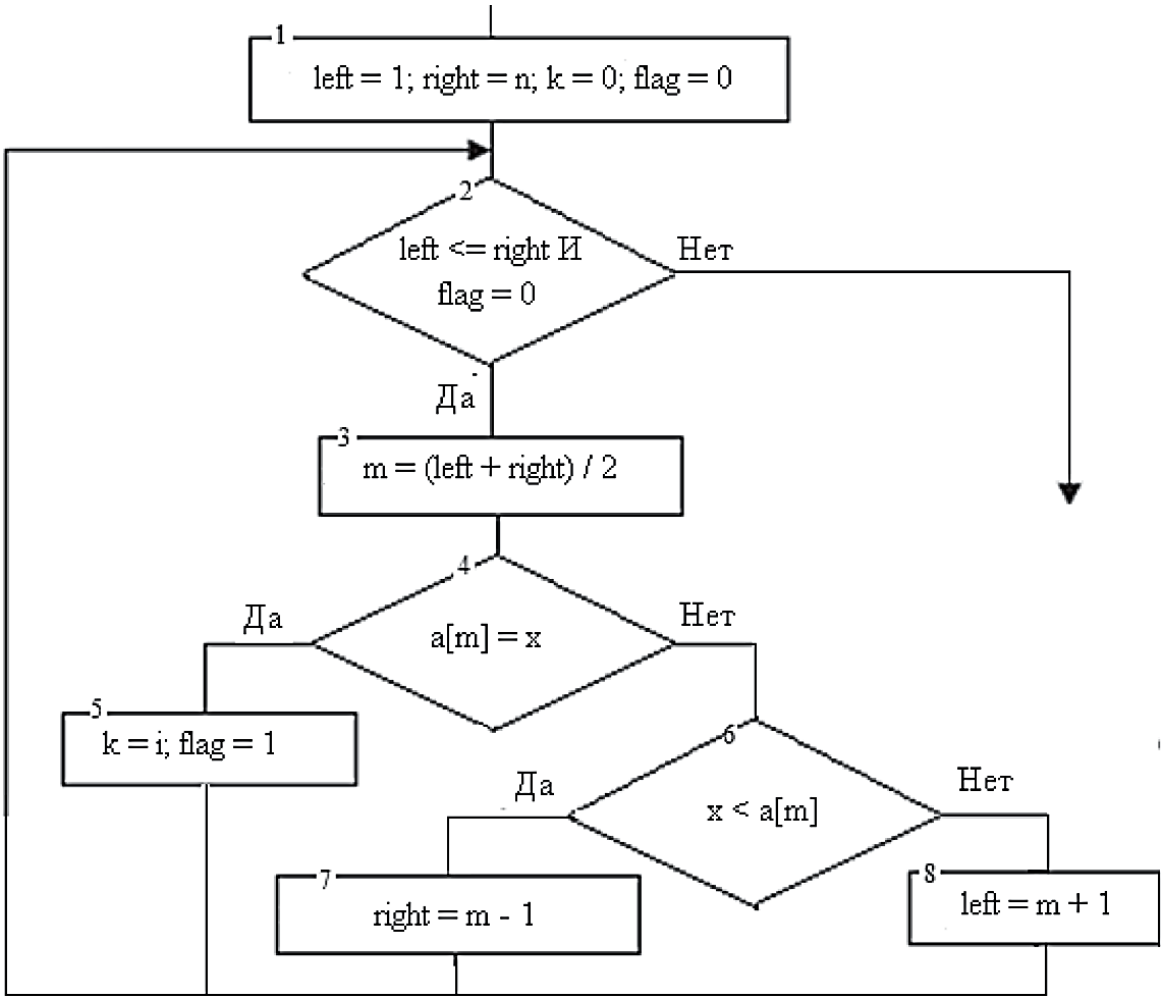
a[3] = 6. Элемент найден, его номер — 3.

В общем случае значение *m* равно целой части от дроби [(*left* + *right*) / 2], где *left* — индекс первого, а *right* — индекс последнего элемента рассматриваемой части массива.

Если *Flag* = 1, то нужное значение в массиве найдено, а если *Flag* = 0, то значения, равного *X*, в массиве нет.

Алгоритм бинарного поиска можно применять только для упорядоченного массива. Это происходит потому, что данный алгоритм использует тот факт, что индексами элементов массива являются последовательные целые числа. По этой причине бинарный поиск практически бесполезен в ситуациях, когда массив постоянно меняется в процессе решения задачи. Алгоритм также используется при сортировке массивов методом бинарного включения.

Ниже приведен фрагмент блок-схемы алгоритма бинарного поиска.


1
left = 1; right = n; k = 0; flag = 0
2
left <= right И
flag = 0
Нет
Да
3
m = (left + right) / 2
4
Да
a[m] = x
Нет
5
k = i; flag = 1
6
Да
x < a[m]
Нет
7
right = m - 1
8
left = m + 1

ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Дайте определение алгоритма. Перечислите свойства алгоритма.
2. Назовите базовые алгоритмические структуры и дайте им краткую характеристику.
3. Дайте определение цикла с заданным числом повторений. Когда целесообразно применять циклы этого вида?
4. Что такое итерационные циклы? Когда возникает необходимость в их использовании?
5. Определите основные отличия между циклами с постусловием и предусловием. Как они выполняются?
6. Дайте определение массива. Поясните, почему для хранения его элементов используется непрерывная память.
7. Можно ли при вводе или выводе элементов массива использовать цикл с предусловием или с постусловием?
8. Укажите, как изменится алгоритм нахождения наибольшего значения, если все элементы массива — отрицательные числа.
9. Дайте определение двумерного массива. Поясните особенности хранения элементов двумерного массива.
10. Почему при составлении алгоритмов для решения задач с использованием двумерного массива применяется вложенный цикл?
11. Перечислите простые алгоритмы сортировки и укажите их основные отличия.
12. Почему сортировка включениями является неэкономным методом?
13. Какими характеристиками должен обладать массив, чтобы применение шейкер-сортировки было эффективным?
14. Назовите метод сортировки, который является лучшим среди простых методов. Поясните, за счет чего это достигается.
15. Почему алгоритм бинарного поиска превосходит «слепой» поиск? Какими характеристиками должен обладать массив, в котором применяется алгоритм бинарного поиска?

БИБЛИОГРАФИЯ

1. Жданова Т.А. Основы алгоритмизации и программирования: учеб. пособие / Т.А. Жданова, Ю.С. Бузыкова. – Хабаровск : Изд-во Тихоокеан. гос. ун-та, 2016. – 56 с.

2. Кадырова, Г. Р. Основы алгоритмизации и программирования: учебное пособие / Г. Р. Кадырова. – Ульяновск: УлГТУ, 2016. – 95 с.

3. Панова, Т.В. Основы алгоритмизации и программирования на языке высокого уровня Си: учебно-практическое пособие / Т.В. Панова, Н.Д. Николаева; Балт. гос. техн. ун-т. – СПб., 2015. – 176 с.

4. Петров Вадим Юрьевич. Информатика. Алгоритмизация и программирование. Учебное пособие. Часть 1. – СПб: Университет ИТМО, 2016. – 91с.

5. Семакин И. Г. Основы алгоритмизации и программирования: учебник для студ. учреждений сред. проф. образования / И. Г. Семакин, А.П. Шестаков. 3-е изд., стер. — М: Издательский центр «Академия», 2016. — 304 с.

6. Трофимов, В. В. Основы алгоритмизации и программирования: учебник для СПО / В. В. Трофимов, Т. А. Павловская; под ред. В. В. Трофимова. — М: Издательство Юрайт, 2018. — 137 с. — (Серия: Профессиональное образование).