Министерство образования и науки Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

ФИЛИПОВСКИЙ ВЛАДИМИР МИХАЙЛОВИЧ

Основы программирования и алгоритмизации. Практикум по алгоритмизации

Санкт-Петербург
2022

УДК 519.6

Филиповский В.М. Основы программирования и алгоритмизации. Ч. 2. Практикум по алгоритмизации: Учебное пособие.- СПб.: СПбПУ, 2022.- 71 с., ил.

Пособие соответствует ФГОС ВО по направлению подготовки 27.03.04 «Управление в технических системах» (уровень бакалавриата).

Рассматриваются основные свойства и способы представления алгоритмов. Подробно изложен структурный подход к разработке алгоритмов с применением типовых схем. Приведено множество примеров с демонстрацией рационального использования основных приемов и типовых схем при решении многочисленных алгоритмических задач. Особое внимание уделено возможным ошибкам и рекомендациям, как этих ошибок избежать и спроектировать высокоэффективный алгоритм.

Содержание настоящего пособия опирается на традиционные методы и подходы, накопленный богатый опыт разработки грамотных алгоритмов с использованием стандартных (типовых) схем.

Учебное пособие предназначено студентам Высшей Школы Киберфизических систем и Управления (программы «Управление в технических системах») в качестве пособия для изучения лекционных курсов и выполнении курсовых и лабораторных работ по курсам "Программирование", "Информатика", "Вычислительная математика", "Математические модели технических систем" и других, связанных с составлением программ для ЭВМ.

Данное пособие является продолжением учебного пособия "Основы программирования и алгоритмизации. Ч. 1. Технология создания программ".

Ключевые слова: Алгоритм, оператор, переменная, метод пошаговой детализации, модульный принцип программирования, структурный подход, типовые схемы алгоритмов, ветвление, циклы, рекурсии, тестирование алгоритма, трассировочная таблица.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компьютер — многозначный термин в современной литературе, наиболее часто употребляемый в качестве обозначения программно управляемого электронного устройства обработки информации (электронно-вычислительной машины).

Жизнь каждого человека тесно связана с таким явлением технического прогресса, как компьютер. Электронно-вычислительная техника стала привычной не только в производстве и научных лабораториях, но и в студенческих аудиториях и школьных классах.

Любой компьютер представляет собой автоматическое устройство, работающее по заложенным в него программам. Программы для компьютера пишутся человеком на основании правил, определяемых языками программирования, на которых указываются действия, которые должен выполнять компьютер.

Компьютерная программа представляет собой последовательность команд, выполняемых процессором автоматически друг за другом в определенном порядке. Данная последовательность называется алгоритмом. Понятие алгоритма такое же основополагающее для информатики, как и понятие информации.

Слово АЛГОРИТМ происходит от имени великого узбекского учёного Мухаммеда аль-Хорезми́, жившего в первой половине IX века. «Аль-Хорезми» означает «из Хорезма».

Около 825 года аль-Хорезми написал сочинение, в котором впервые дал описание придуманной в Индии позиционной десятичной системы счисления. В первой половине XII века книга аль-Хорезми в латинском переводе проникла в Европу. Переводчик, имя которого до нас не дошло, дал ей название «Algoritmi de numero Indorum» («Индийское искусство счёта, сочинение Аль-Хорезми»). В течение последующих нескольких столетий в Европе сочинения по искусству счёта назывались Алгоритмами. Но уже в 18 веке впервые в литературе стали появляться современные понятия алгоритма. К началу XX века для математиков слово «алгоритм» уже означало любой арифметический или алгебраический процесс, выполняемый по строго определённым правилам. С появлением компьютеров понятие «алгоритм» обрело новую жизнь.

В настоящее время существует несколько определений алгоритма.

- *Алгоритм* — это конечный набор правил, который определяет последовательность операций для решения конкретного множества задач и обладает пятью важными чертами: конечность, определённость, ввод, вывод, эффективность (*Д. Э. Кнут*).
- *Алгоритм* — это всякая система вычислений, выполняемых по строго определённым правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи (*А. Н. Колмогоров*).
- *Алгоритм* — это последовательность действий, либо приводящая к решению задачи, либо поясняющая, почему это решение получить нельзя.

- *Алгоритм* или операционное правило обработки данных — это конструктивное описание, состоящее из конечного множества правил и определяющее процесс переработки данных.
- *Алгоритм* — это точный набор инструкций, описывающих порядок действий некоторого исполнителя для достижения результата, решения некоторой задачи за конечное время.
- *Алгоритм* — заранее заданное понятное и точное предписание возможному исполнителю совершить определенную последовательность действий для получения решения задачи за конечное число шагов.

Следовательно, алгоритм задает некоторый свод правил, описывающих процесс, протекающий во времени и определяющий последовательность действий для перехода от исходной ситуации к желаемому новому состоянию.

Описание такого свода правил может быть выполнено на обычном языке с помощью математических формул или же с использованием специальных символов.

Разработка алгоритма для решения любой задачи является наиболее важным и ответственным процессом, так как именно алгоритм определяет ту последовательность действий, которая выполняется машиной. Ошибки, полученные при записи алгоритма, обычно приводят к неверному ходу вычислительного процесса (процесса обработки данных) и, следовательно, к ошибочному результату. Результат может быть и верным, но полученным не оптимальным путем, если используется алгоритм, не учитывающий индивидуальных особенностей задачи. Основная проблема при эффективном использовании ЭВМ - построение ХОРОШЕГО алгоритма.

Алгоритмизация - это раздел информатики, изучающий свойства, методы и приемы построения алгоритмов (иногда также называют алгоритмикой).

Целью настоящего пособия является:

- изучение алгоритмизации и алгоритмов, на основании которых может быть построена эффективная программа для вычислительной машины (компьютера);
- рассмотрение типовых структур, по которым строится грамотный (и правильный) алгоритм;
- анализ возможных ошибок, возникающих при проектировании алгоритмов для решения различных задач;
- выработка рекомендаций, позволяющих разработать рациональный алгоритм, и, как следствие, построить эффективную задачу.

Данное пособие является продолжением учебного пособия: "Филиповский В.М. Основы программирования и алгоритмизации. Ч. 1. Технология создания программ" [4].

ЧАСТЬ ВТОРАЯ. ПРАКТИКУМ ПО АЛГОРИТМИЗАЦИИ

Разработка алгоритма является наиболее важным и ответственным процессом при решении любой задачи, так как именно алгоритм определяет ту последовательность действий, которая выполняется машиной. Алгоритмизация - ответственный этап разработки программного обеспечения. Создание ХОРОШЕГО алгоритма позволяет уже на этапе его построения исключить все возможные ошибки в будущей программе, обеспечить ее высокоэффективную безотказную работу.

1. ОСНОВНЫЕ СВОЙСТВА АЛГОРИТМА

Алгоритм — это точный набор инструкций, описывающих порядок действий исполнителя для достижения результата, решения некоторой задачи за конечное время. Именно алгоритм определяет ту последовательность действий, которая выполняется машиной.

Любой алгоритм определяется следующими атрибутами: именем, входными и выходными данными, началом, концом, командами (действиями, операторами).

Имя (наименование) алгоритма раскрывает его смысл и определяется решаемой задачей.

Входные и выходные данные необходимо описать до создания алгоритма, на этапе осмысления задачи. Это как в математике: Вопросы "Что дано?", "Что надо получить?" определяют отправную точку и цель любого процесса. Не исключением является и процесс программирования (построения алгоритма).

Рекомендуется перед разработкой алгоритма уделить особое внимание формулированию имени, набора входных и выходных данных. Лучше записать их на листе бумаги.

1.1. Свойства алгоритма

Можно указать следующие основные СВОЙСТВА алгоритма:

- ДИСКРЕТНОСТЬ (от лат. discretus – разделенный, прерывистый) указывает, что любой алгоритм должен состоять из конкретных действий, следующих в определенном порядке. Данное свойство состоит в том, что алгоритм должен представлять процесс решения задачи как последовательное выполнение простых (или ранее определенных) шагов (этапов). При этом для выполнения каждого этапа требуется некоторый отрезок времени. То есть преобразование исходных данных в результат осуществляется во времени дискретно.
- ОПРЕДЕЛЕННОСТЬ или детерминированность (от лат. determinate – определенность, точность) указывает, что любое действие алгоритма должно быть строго и недвусмысленно определено в каждом отдельном

случае. Это свойство заключается в том, что каждое правило алгоритма должно быть четким, однозначным и не оставлять места для произвола. Благодаря этому свойству выполнение алгоритма носит механический характер и не требует никаких дополнительных сведений о решаемой задаче.

- РЕЗУЛЬТАТИВНОСТЬ (или конечность). Это свойство означает, что алгоритм должен приводить к решению задачи за конечное число шагов. Результативность требует, чтобы в алгоритме не было ошибок, то есть при точном исполнении всех команд процесс решения задачи должен прекратиться за конечное число шагов (завершиться!) и при этом должен быть получен ответ.
- МАССОВОСТЬ. Смысл этого свойства заключается в том, что алгоритм решения задачи разрабатывается в общем виде, и он должен быть применим для некоторого класса однотипных задач, различающихся конкретными значениями исходных данных. При этом исходные данные могут выбираться из некоторой области, называемой ОБЛАСТЬЮ ПРИМЕНИМОСТИ алгоритма.

1.2. Способы представления алгоритма

Разработанный алгоритм фиксируется несколькими способами:

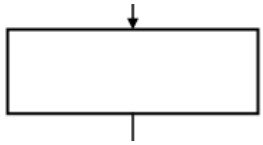
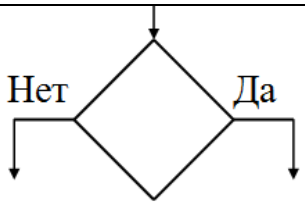
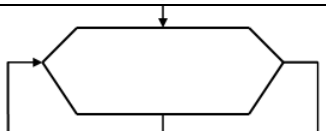
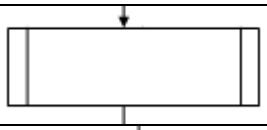
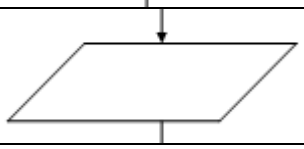
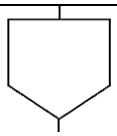
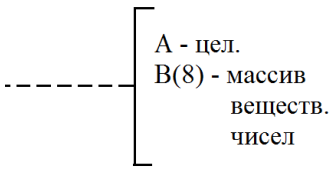
1. **На естественном языке**. Хотя естественный язык не требует детальных разъяснений и полной формализации, существуют некоторые стандартные языковые конструкции, которые используются при данном способе представления алгоритма. Применяются ключевые слова, например:

   ЕСЛИ, ИДТИ, ВЫПОЛНИТЬ и т.п.

2. **В виде схемы** (наиболее распространенный способ изображения алгоритма). СХЕМОЙ АЛГОРИТМА называется наглядное графическое изображение последовательности необходимых действий. Схема алгоритма составляется из элементов в виде геометрических фигур, имеющих по предварительному соглашению вполне определенный однозначный смысл и обозначающих различные действия. Эти элементы соединяются между собой стрелками, указывающими направление переходов от одних действий (операций) к другим. Направления сверху вниз и слева направо принимаются за основные и могут (если не имеют изломов) не обозначаться стрелками. Внутри фигур, как правило, указываются те операции, которые в данном блоке выполняются. Для указания конкретных операций, выполняемых в блоке, обычно используются математические символы или записи в виде текста. Употребляются также некоторые специальные символы, например символ := обозначает, что указанное в блоке значение должно быть заменено новым (присвоено новое значение).

В таблице 1 приведены наиболее часто употребляемые блоки и даны пояснения к ним.

Таблица 1

Наименование символа	Изображение символа	Примечание
Процесс		Вычислительное действие или последовательность вычислительных действий. Арифметический блок
Принятие решения		Проверка условий. Логический блок
Модификация		Начало и конец арифметического цикла
Предопределенный процесс		Вычисления по подпрограмме (функции)
Передача данных		Ввод данных или вывод данных; печать результатов
Прерывание		Начало, конец, останов
Соединитель		Разрыв линий потока информации
Межстраничный соединитель		Указатель соединения между частями схемы, расположенными на разных листах
Комментарий		Пояснительные записи в целях объяснения или примечания к какому-нибудь блоку схемы

Блоки соединяются линиями потока информации. Внутри блоков записываются выполняемые действия. Линии определяют направление вычислений, причем по умолчанию сверху вниз и слева направо. Если необходимо отразить другое направление (снизу вверх и справа налево), то необходимо на линиях ставить стрелки. Блоки на схеме могут нумероваться цифрами, которые ставятся в разрыве верхней линии слева.

Основными достоинствами графического представления алгоритма является наглядность, особенно, если схема помещается на одной странице. Но наряду с достоинствами данный способ имеет и недостатки: большая трудоем-

кость вычерчивания схем алгоритмов, при внесении изменений в алгоритм схему приходится перечерчивать заново.

Полный перечень установленных правил и условных графических обозначений определен ГОСТ 19.701-90 [1].

СХЕМА - графическое представление определения, анализа или метода решения задачи, в котором используются символы для отображения операций, данных, потока, оборудования и т.д.

ГОСТ устанавливает следующие виды схем.

- Схема данных - отображает путь данных при решении задач и определяет этапы обработки, а также различные применяемые носители данных. Примером может служить рис.2.
- Схема программы - отображает последовательность операций в программе. Именно схему программы часто называют АЛГОРИТМОМ ПРОГРАММЫ, либо БЛОК-СХЕМОЙ ПРОГРАММЫ.
- Схема работы системы - отображает управление операциями и поток данных в системе.
- Схема взаимодействия программ - отображает путь активаций программ и взаимодействий с соответствующими данными. Каждая программа в схеме взаимодействия программ показывается только один раз. Эту схему часто называют структурой программы.
- Схема ресурсов системы - отображает конфигурацию блоков данных и обрабатывающих блоков, которая требуется для решения задачи или набора задач.

Наиболее часто применяются схема данных, схема программы и схема взаимодействия программ.

3. **На специальном языке** для записи алгоритмов (АЛГОРИТМИЧЕСКОМ ЯЗЫКЕ). Алгоритмические языки близки к естественному языку. Однако правила построения конструкций более "жесткие". Небольшие ошибки или описки, допускаемые в предложениях естественного языка и не искажающие смысла, в алгоритмическом языке совершенно недопустимы. Программа, составленная на алгоритмическом языке, не может быть непосредственно выполнена ЭВМ. Поэтому необходимо промежуточное звено, которое выполняло бы работу по расчленению отдельных действий программы и записи их на машинном языке. Данная процедура выполняется в ЭВМ с помощью специальной программы - ТРАНСЛЯТОРА.

Главный недостаток такого способа представления алгоритма: современные алгоритмические языки не обладают наглядными средствами для описания алгоритмов и требуют использования многих второстепенных языковых конструкций, затрудняющих понимание общих принципов построения алгоритмов и программ. Поэтому представление алгоритма на алгоритмическом языке

можно рекомендовать лишь после построения его схемы. Такой подход из-за соображений наглядности графической формы алгоритма позволяет получить как более рациональный (в общем случае) алгоритм, так и избежать значительного количества ошибок в программе (в основном логического характера) еще на этапе разработки ее алгоритма.

1.3. Структурный подход к разработке алгоритмов

На практике при разработке алгоритмов и программ используется так называемый структурный подход.

Принцип структурного программирования: алгоритм любой сложности может быть представлен комбинацией нескольких базовых структур. Основные структуры алгоритмов - это ограниченный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий.

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

К основным структурам относятся следующие (приводятся примеры описания на естественном языке и схемы):

1. СЛЕДОВАНИЕ. Последовательное размещение блоков и групп блоков. В программе реализуется последовательным размещением операторов.

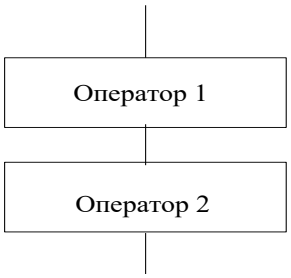


Рис. 1. Структура типа "СЛЕДОВАНИЕ"

2. ВЕТВЛЕНИЕ. Применяется, когда в зависимости от условия нужно выполнить либо одно, либо другое действие. Каждое из действий может в свою очередь содержать несколько этапов.

Пример:

1. Если УСЛОВИЕ , идти к п.2
2. Операторы действия "1"
3. Идти к п.5

иначе, идти к п. 3

4. Операторы действия "2"
5....

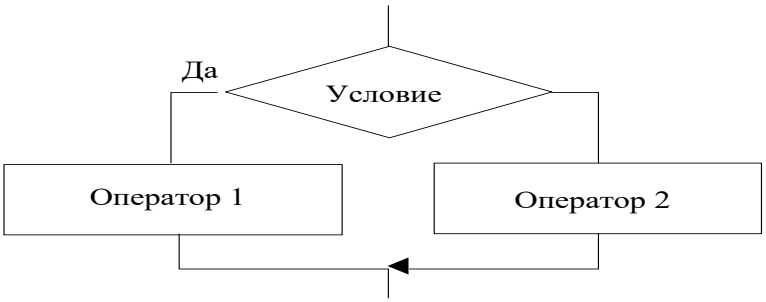


Рис. 2. Структура типа "ВЕТВЛЕНИЕ"

Внимание! Обе ветви структуры сходятся в одной точке (Таким образом реализуется один выход).

3. ОБХОД. Частный случай ВЕТВЛЕНИЯ, когда одна ветвь не содержит никаких действий.

Пример:

1. Если УСЛОВИЕ , идти к п.2
2. Операторы действия
3. Идти к п. 4

иначе, идти к п. 4

4....

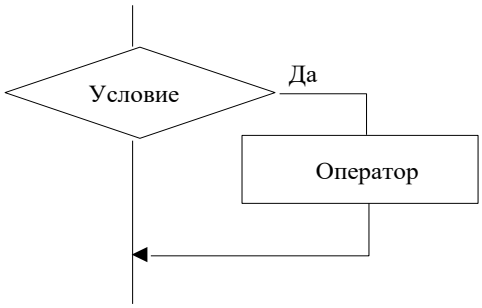


Рис. 3. Структура типа "ОБХОД"

Внимание! Обе ветви структуры сходятся в одной точке!

Рекомендация: При использовании структур ВЕТВЛЕНИЕ либо ОБХОД оператор ПРОВЕРКА УСЛОВИЯ желательно писать простым. В случае сложных условий целесообразно разбивать их на простые, используя вложенные структуры типа ВЕТВЛЕНИЯ.

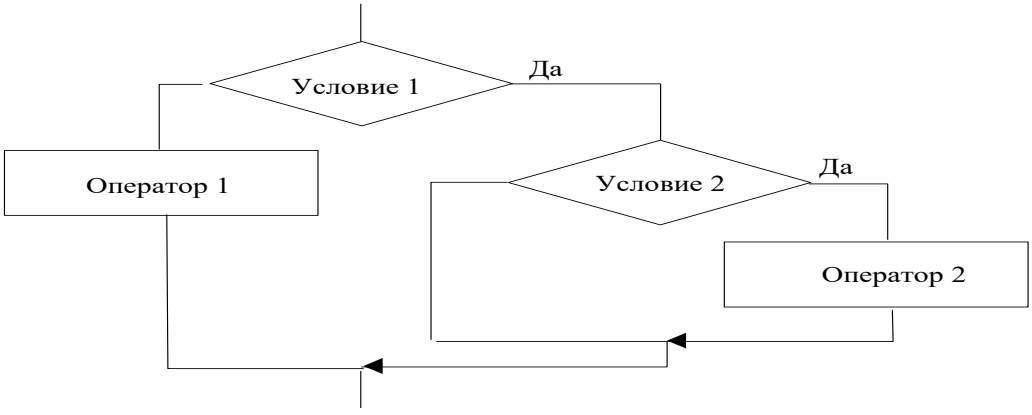


Рис. 4. Структура типа "ВЛОЖЕННОЕ ВЕТВЛЕНИЕ"

4. МНОЖЕСТВЕННЫЙ ВЫБОР. Является обобщением структуры типа ВЕТВЛЕНИЕ, когда в зависимости от значения управляющей переменной выполняется одно из нескольких действий (Д1, Д2, Д3,...)

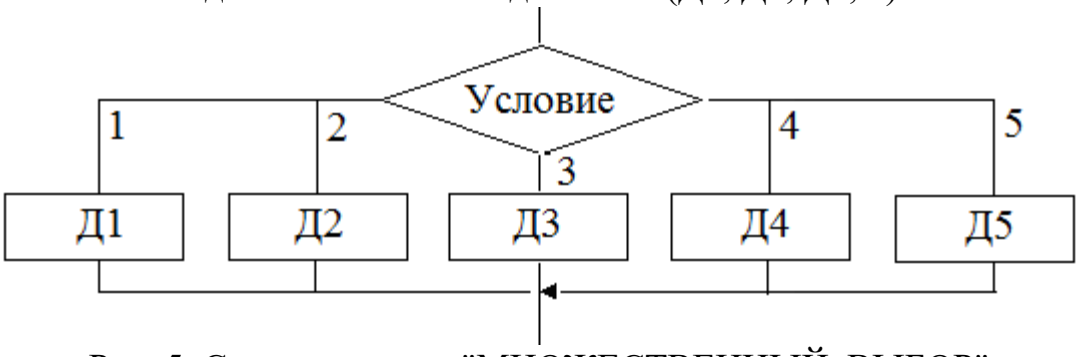


Рис. 5. Структура типа "МНОЖЕСТВЕННЫЙ ВЫБОР"

Внимание! Все ветви структуры сходятся в одной точке!

5. ЦИКЛ "ДО". Применяется при необходимости выполнить какие-либо вычисления несколько раз до выполнения некоторого условия.

Пример:

....
1. Операторы начальных присваиваний
2. Операторы тела цикла
3. Если УСЛОВИЕ идти к п.2.
иначе, идти к п.4.
4....

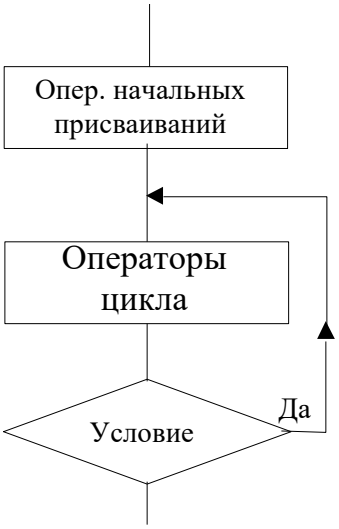


Рис. 6. Структура типа "ЦИКЛ "ДО"

Особенность этого цикла в том, что операторы цикла выполняются хотя бы один раз, так как первая проверка условия выхода из цикла происходит после того, как тело цикла выполнено. ТЕЛО ЦИКЛА - это та последовательность действий, которая выполняется многократно. НАЧАЛЬНЫЕ ПРИСВАИВАНИЯ - задание начальных значений тем переменным, которые используются в теле цикла.

Внимание! Операторы начальных присваиваний размещаются непосредственно перед операторами цикла (в самом начале цикла)

6. ЦИКЛ "ПОКА". Цикл "ПОКА" отличается от цикла "ДО" тем, что проверка условия проводится до выполнения тела цикла, и если при первой проверке условие выхода из цикла выполняется, то тело цикла не выполняется ни разу.

Пример:

....
1. Оператор начальных присваиваний
2. Если УСЛОВИЕ, идти к п.3,
   иначе - к п.5 (выход из цикла)
3. Операторы тела цикла
4. Идти к п.2
5....

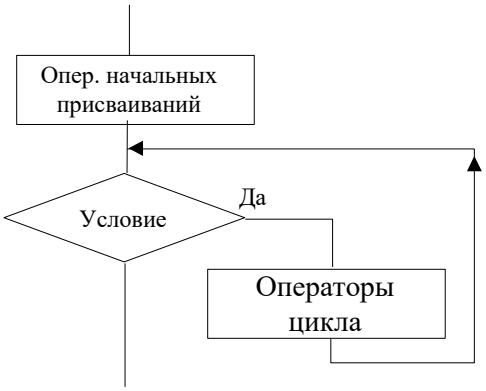


Рис. 7. Структура типа "ЦИКЛ "ПОКА"

7. АРИФМЕТИЧЕСКИЙ ЦИКЛ. Цикл "АРИФМЕТИЧЕСКИЙ" отличается от цикла "ПОКА" тем, что его заголовке указывается три параметра: начальное значение переменной (от), конечно значение (до) и ее изменение с помощью арифметической операции на каждом «обороте» цикла (шаг).

Пример:

....
1. Заголовок цикла, указывающий начальное значение переменной, ее конечное значение и шаг изменения,
2. Операторы тела цикла
3. .....

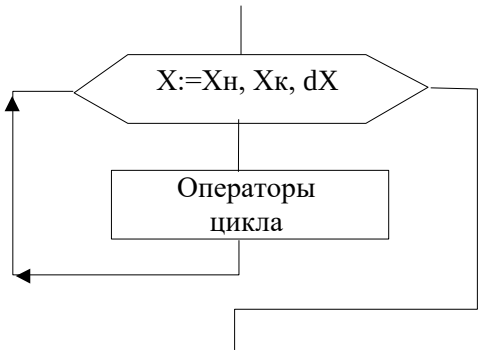


Рис. 8. Структура типа "АРИФМЕТИЧЕСКИЙ ЦИКЛ"

Особенностью всех приведенных структур является то, что они имеют один вход и один выход, и их можно соединять друг с другом в любой после-

довательности. В частности, каждая структура в качестве одного из блоков может содержать любую другую структуру (вложение структур).

Обычно при составлении схемы блоки размещаются друг под другом в порядке их выполнения. Возврат назад осуществляется только в циклах. Это дает простую и наглядную структуру алгоритма, по которой легко далее составить программу.

Структурный подход к программированию естественно не является догмой. В некоторых случаях применение структурного метода приводит к необоснованному усложнению программы и потере ее наглядности и естественности. Поэтому иногда оказывается целесообразным отказаться от структурного программирования, отдав предпочтение ясности и естественности программы.

Внимание! Следует иметь ввиду, что применение стандартных структур позволяет уменьшить количество ошибок на следующем этапе разработки программы - кодировании.

1.4. Основные приемы разработки алгоритма

Одним из приемов разработки алгоритма решения сложных задач является метод ДЕКОМПОЗИЦИИ (метод ПОШАГОВОЙ ДЕТАЛИЗАЦИИ).

Декомпозиция — разделение целого на части. Декомпозиция — это научный метод, использующий структуру задачи и позволяющий заменить решение одной большой задачи решением серии меньших задач, пусть и взаимосвязанных, но более простых, которые, в свою очередь, также могут быть расчленены на части.

При методе пошаговой детализации первоначально продумывается и фиксируется общая структура алгоритма без детальной проработки отдельных его частей. Блоки, требующие дальнейшей детализации, обозначаются пунктирной линией. Далее прорабатываются (детализируются) отдельные блоки, не детализированные на предыдущем шаге. То есть на каждом шаге разработки уточняется реализация фрагмента алгоритма (программы), и, таким образом, на каждом шаге имеют дело с более простой задачей. Полностью закончив детализацию всех блоков, получают решение всей задачи в целом. Необходимо внимательно следить за тем, чтобы на каждом шаге структура алгоритма оставалась простой и ясной.

Описанный метод пошаговой детализации называется также программированием СВЕРХУ ВНИЗ.

Противоположным рассмотренному подходу является принцип программирования СНИЗУ ВВЕРХ. Альтернативное название данного подхода - метод ВОСХОДЯЩЕГО ПРОГРАММИРОВАНИЯ.

Данная методика разработки программ предполагает, что крупные блоки собираются из ранее созданных мелких блоков. Восходящее программирование начинается с разработки ключевых процедур и подпрограмм, которые за-

тем постоянно модифицируются. Иными словами, идея этого принципа заключается в том, что решаются конкретные небольшие задачи, а их результаты объединяются в более крупное решение.

Положительной стороной такого подхода является то, что конкретные, маленькие задачи решаются проще и быстрее.

Отрицательной — качество решения основной, глобальной задачи зависит от профессионализма разработчиков, от их способностей собрать небольшие решения в конкретное целое.

Принято считать метод пошаговой детализации (программирование СВЕРХУ ВНИЗ) более естественным. Он дает возможность руководству лучше оценивать состояние работ и зачастую исключает болезненный процесс объединения модулей, необходимый при разработке методом СНИЗУ ВВЕРХ.

2. РАЗРАБОТКА АЛГОРИТМА. ПРИМЕНЕНИЕ ТИПОВЫХ СХЕМ

Важно при разработке алгоритма решения конкретной задачи использовать язык блок-схем. Решение одной и той же задачи может быть реализовано с помощью разных алгоритмов, отличающихся друг от друга как по времени счета и объему вычислений, так и по своей сложности. Запись этих алгоритмов с помощью блок-схем позволяет сравнивать их, выбирать наилучший алгоритм, упрощать его, находить и устранять ошибки.

Отказ от языка блок-схем при разработке алгоритма и запись алгоритма сразу на языке программирования может приводить к неоптимальному решению задачи, к значительным потерям времени. Поэтому сначала разрабатывается алгоритм решения конкретной задачи на языке блок-схем, после чего он будет переведен на язык программирования (этап кодирования).

Рекомендуется при создании алгоритма решения любой задачи использовать стандартные (типовые) блок-схемы. Их применение позволяет уже на этапе разработки алгоритма устранить большинство возможных ошибок.

При разработке алгоритма сложной задачи используется метод пошаговой детализации. Первоначально разрабатывается общая структура алгоритма без детальной проработки отдельных его частей. Блоки, требующие детализации, продумываются и детализируются на последующих шагах разработки алгоритма.

В данном разделе показано применение типовых схем для создания алгоритмов решения конкретных задач.

2.1. Общие правила по составлению алгоритма. Схема "следование"

Наиболее простым видом алгоритма является ***линейный*** алгоритм, при котором действия выполняются последовательно, одно за другим, без разветвлений и возвратов. На его примере показано, как должен быть оформлен (изображен) любой алгоритм.

Пример 1. Вычислить площадь треугольника по трем сторонам a, b, c помощью формулы Герона:

$$\mathbf{S = \sqrt{p(p-a)(p-b)(p-c)}}\text{, где } \mathbf{p = \frac{a+b+c}{2}}.$$

Блок-схема алгоритма вычисления площади треугольника изображена на Рис. 9. Как следует из рисунка, на схеме обозначены начало и конец алгоритма. Надписи "Начало" и "Конец" обозначать необязательно. В комментарии, приведенном в схеме после Начала алгоритма, описаны переменные, используемые для решения данной задачи.

Ниже, на Рис. 10, изображена схема алгоритма решения той же задачи с применением специальной функции, непосредственно вычисляющей площадь данной фигуры. Вызов функции в основном алгоритме обозначен блоком "Предопределенный процесс". Параметры и локальные переменные функции описаны в соответствующих комментариях.

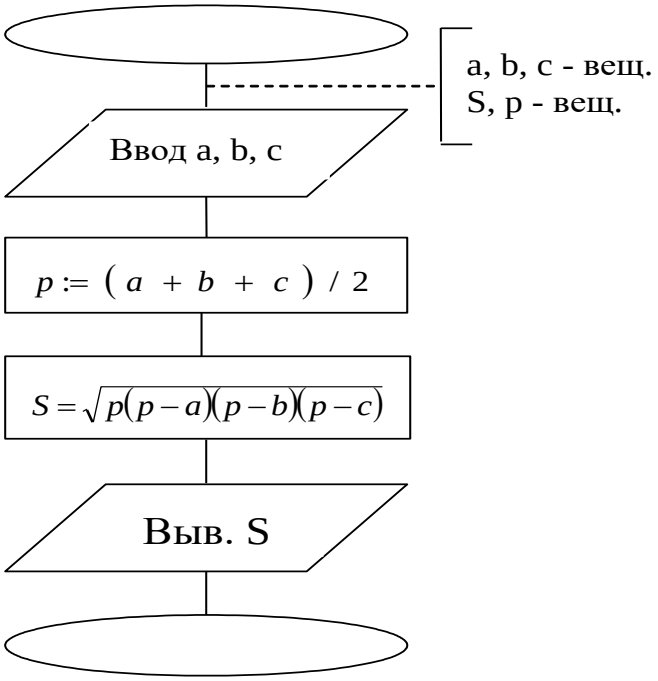


Рис. 9. Схема алгоритма вычисления площади треугольника

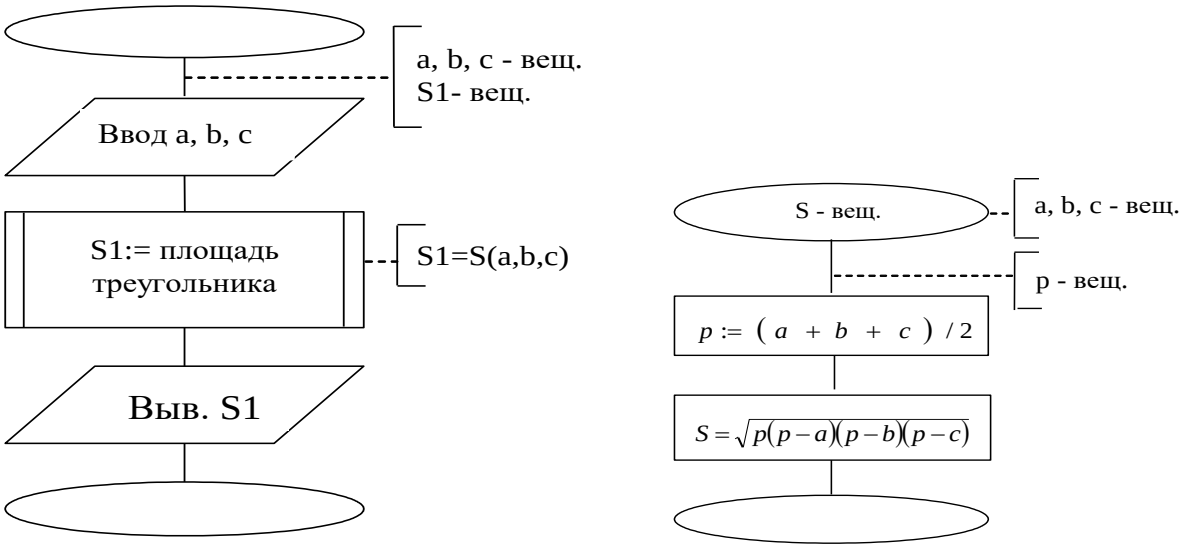


Рис. 10. Алгоритм вычисления площади треугольника с помощью функции

Во всех последующих примерах блок-схем алгоритмов комментарии с описанием локальных переменных, а также блоки "Начало" и "Конец" схемы будут опущены.

2.2. Типовая структура "Ветвление"

В процессе решения многих задач возникает необходимость в зависимости от исходных данных или получающихся промежуточных результатов проводить вычисления либо по одним, либо по другим формулам. В алгоритме этот

факт отображается разными направлениями, ветвями. Такой вычислительный алгоритм называется ВЕТВЛЕНИЕМ. Несмотря на относительную простоту этих задач разработка алгоритма имеет свои особенности и, как следствие, некоторые стандартные ошибки, рассмотренные в следующих примерах.

Пример 2. Вычислить $y(x)=\begin{cases} x+1, & \text{если } x \ge 1; \\ x^2, & \text{если } x < 1. \end{cases}$

Первый вариант алгоритма, изображенный на рис. 11., выполнен с использованием двух последовательных структур типа "ВЕТВЛЕНИЕ". Налицо избыточная сложность разработанного алгоритма.

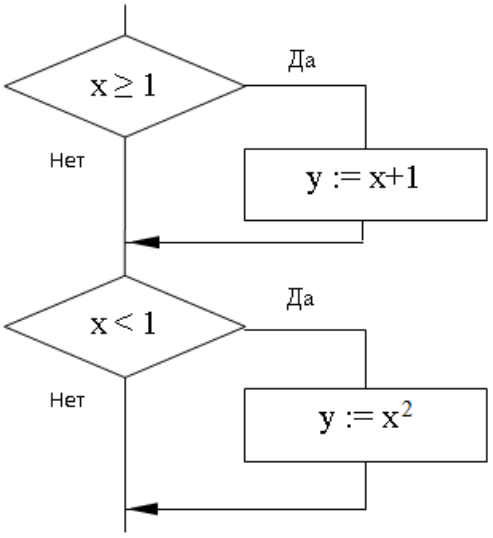


Рис. 11.

Второй вариант алгоритма, изображенный на рис. 12., выполнен с использованием структуры типа "ВЛОЖЕННОЕ ВЕТВЛЕНИЕ". Во втором операторе ЕСЛИ правая ветвь является избыточной (лишней), что также усложняет алгоритм.

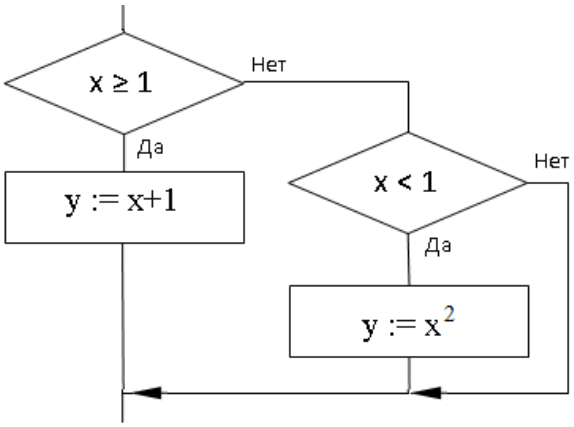


Рис. 12.

Учитывая, что в условии Примера 2 области определения являются взаимоисключающими, имеет смысл при построении алгоритма обойтись одной структурой типа "ВЕТВЛЕНИЕ" (одним оператором ЕСЛИ). Полученный алгоритм изображен на Рис. 13.

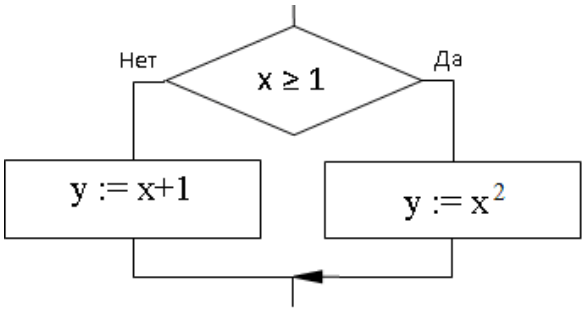


Рис. 13.

Все три изображенных алгоритма (Рис. 11-13) являются работоспособными, но имеют один и тот же недостаток: Вычисленное значение *y(x)* не сообщается пользователю. Для устранения данной ошибки необходимо ввести в алгоритм оператор ВЫВОДА. На Рис. 14. показано, как может быть решена эта задача.

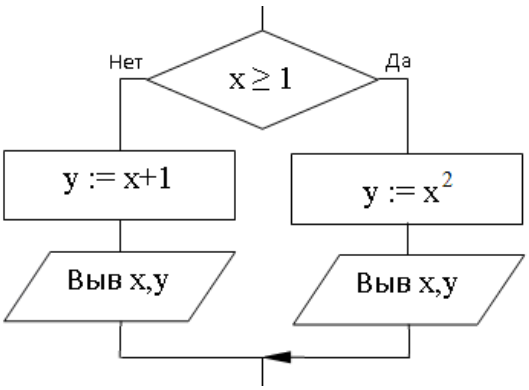
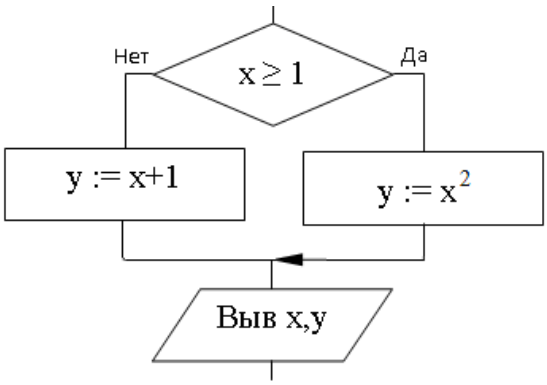


Рис. 14. а)



б)

Правая схема на Рис. 14. предпочтительнее левой, потому что в обеих ветвях оператора ЕСЛИ правой схемы (Рис. 14. а)) имеется одинаковый оператор ВЫВОДА. Имеет смысл его ВЫНЕСТИ из структуры "ВЕТВЛЕНИЕ", что и сделано в правой схеме (Рис. 14. б)).

Пример 3. Вычислить $y(x)=\begin{cases} x+1, & \text{если } x>0 \text{ и } A>1; \\ x^2, & \text{если } x<1 \text{ и } A<2. \end{cases}$

Прежде, чем строить алгоритм этой, и иной, подобной ей, задачи, рекомендуется построить область определения функции *y(x)*. В дальнейшем это позволяет грамотно организовать обход области с целью наиболее быстрого вычисления значения функции, то есть разработки эффективного алгоритма. На Рис. 12. построена область определения решаемой задачи в координатах (A, x).

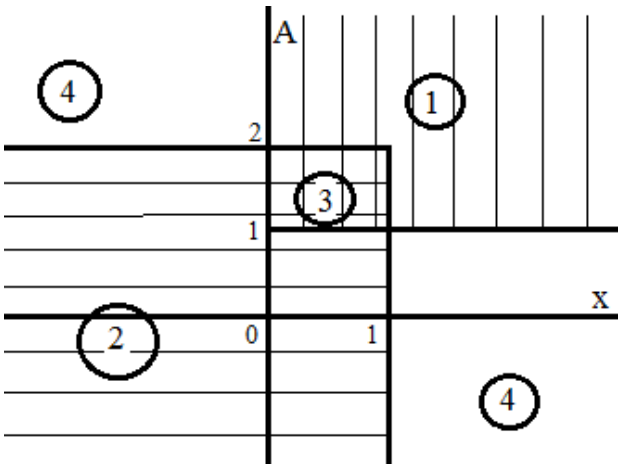


Рис. 12.

На ней обозначены: 1 и 2 - области определения функции по условиям задачи; 3 - область, где функция определена неоднозначно; 4 - в этой области значения функции не определены.

Ниже, на Рис. 13. изображен один из возможных алгоритмов решения задачи Примера 3. При построении алгоритма применена переменная Fl (флаг), которая первоначально принимает значение в соответствии с областью определения (см. Рис. 12). В дальнейшем (в нижней части алгоритма) ее значение использовано для вычисления конкретного значения функции и соответствующей печати (вывода) с помощью оператора МНОЖЕСТВЕННЫЙ ВЫБОР.

Следует обратить **внимание**, что **обе ветви** каждого оператора ЕСЛИ **сходятся в одной точке**. - Это важный признак грамотного применения типовой структуры ВЕТВЛЕНИЕ!

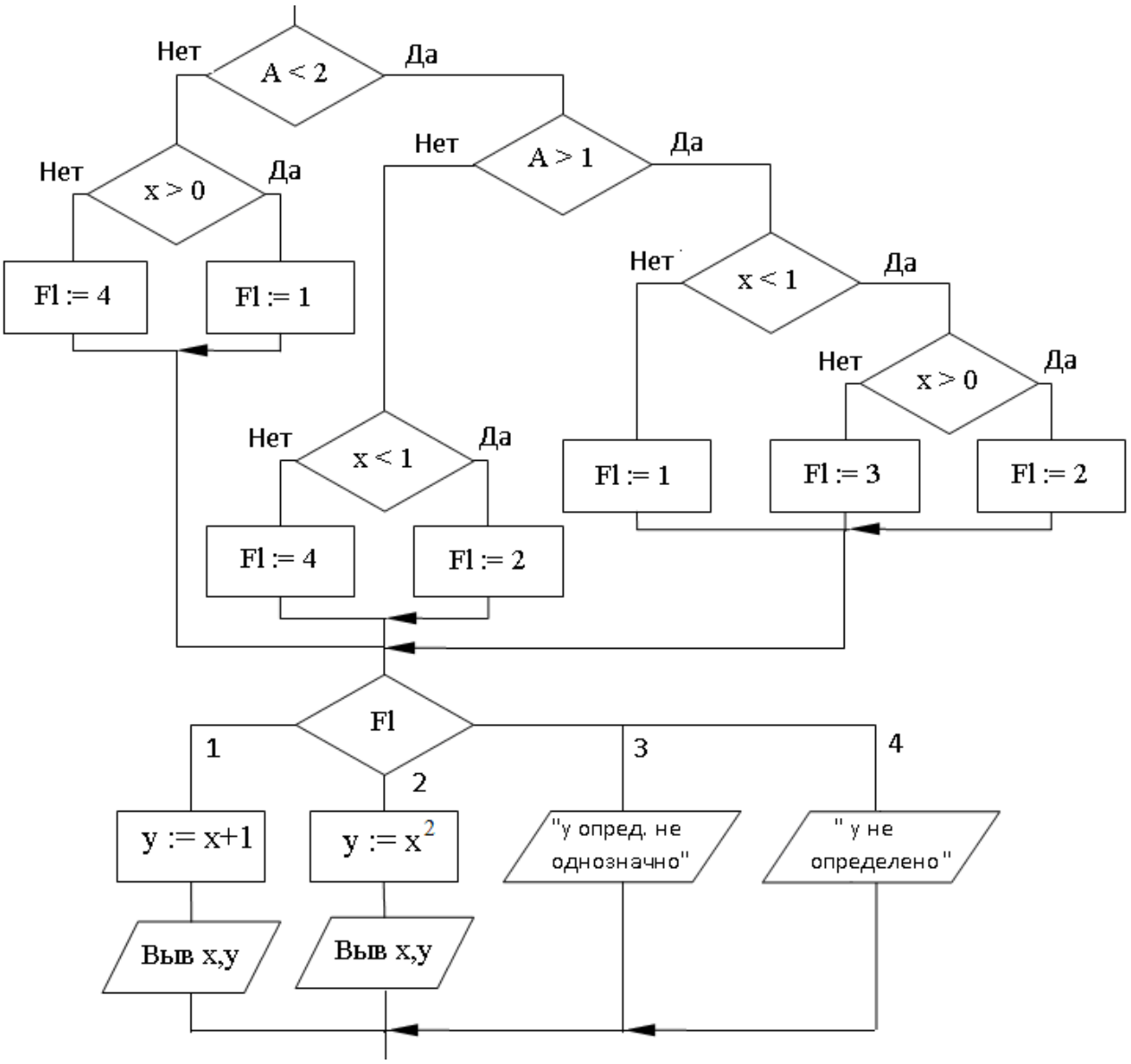


Рис. 13.

Как уже было отмечено, алгоритм, изображенный на Рис. 13, является не единственно правильным решением задачи Примера 3. Ниже, на Рис. 14., показан альтернативный алгоритм, полученный без учета области определения вычисляемой функции. Для построения корректного алгоритма также использован флаг Fl, который увеличивает свое значение на единицу после вычисления соответствующего значения функции. Таким образом Fl = 0 соответствует области определения (4), когда значение функции не определено; Fl = 2 — когда значение функции $y(x)$ определено неоднозначно; а Fl = 1 — когда функция вычислена по одному из возможных правил.

Полученное значение Fl использовано в нижней части алгоритма для организации корректной печати результата вычисления $y(x)$. Вместо применения двух операторов ЕСЛИ организовать вывод результата вычисления можно и с помощью единственного оператора МНОЖЕСТВЕННЫЙ ВЫБОР, как в предыдущем варианте.

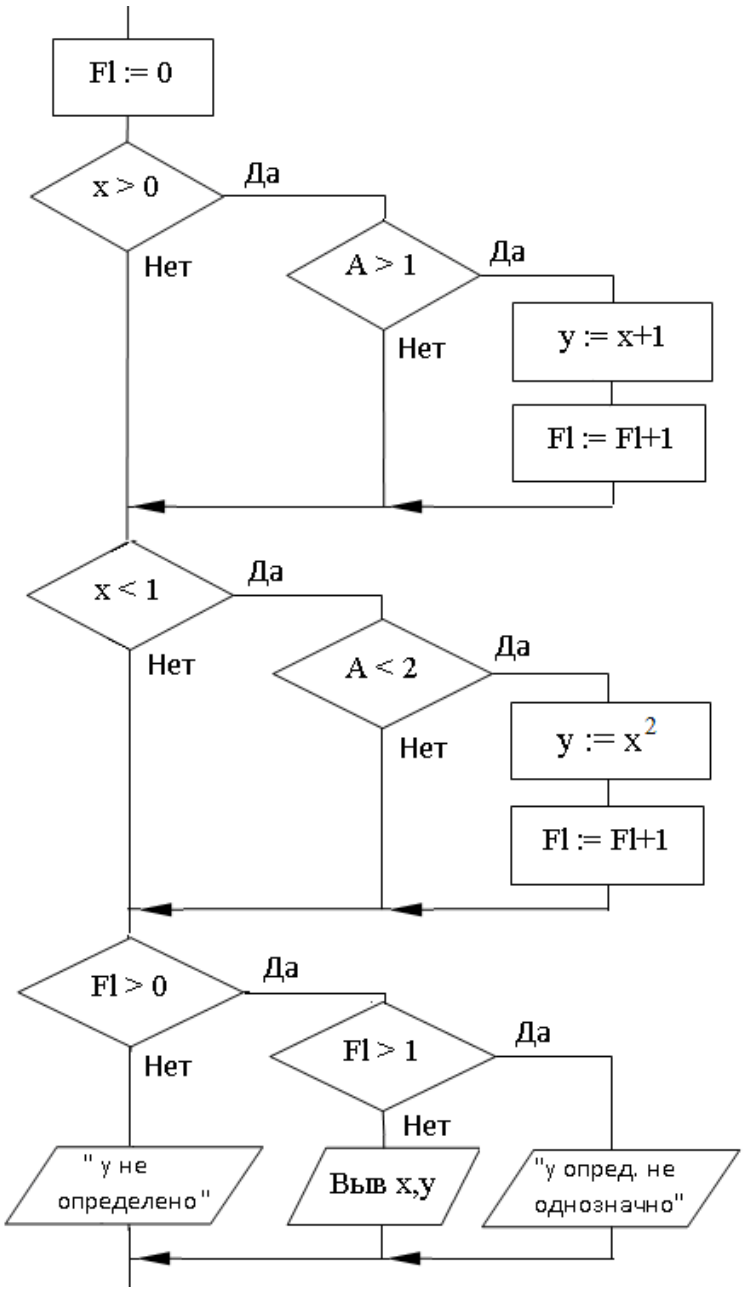


Рис. 14.

В заключение имеет смысл повторить важное замечание:

Обе ветви каждого оператора ЕСЛИ схемы ВЕТВЛЕНИЕ должны **сходиться в одной точке**!!!

2.3. Циклические алгоритмы

Большинство практических задач требует многократного повторения одних и тех же действий, то есть повторного использования одного или нескольких операторов. Многократно исполняемый участок алгоритма называется ОПЕРАТОРАМИ ЦИКЛА (иногда ТЕЛОМ ЦИКЛА, или ЦИКЛОМ). Соответственно циклический алгоритм — это алгоритм, содержащий циклы.

Различают два типа циклов: с известным и с неизвестным числом повторений. При этом в обоих случаях имеется в виду число повторений на стадии разработки алгоритма. Существует 3 типа циклических структур:

- Цикл с предусловием (цикл ПОКА);
- Цикл с постусловием (цикл ДО);
- Цикл с параметром (АРИФМЕТИЧЕСКИЙ цикл).

Тип структуры цикла определяется непосредственно решаемой задачей.

Перед рассмотрением конкретных примеров имеет смысл сформулировать **рекомендации**, необходимые и важные для разработки алгоритмов типа ЦИКЛ.

Во-первых, сначала определяются повторяемые ОПЕРАТОРЫ ЦИКЛА (тело цикла).

Во-вторых, записываются операторы, называемые НАЧАЛЬНЫМИ ОПЕРАТОРАМИ ЦИКЛА (стоят непосредственно перед циклом!!!), необходимые для правильности первого выполнения операторов цикла.

В-третьих, составляется УСЛОВИЕ ВЫХОДА ИЗ ЦИКЛА (оператор, определяющий, продолжать цикл, или необходимо его закончить).

Пример 4. Получить таблицу значений функции $y = f(x)$, $x \in [-1,\ 3]$, с шагом $dx = 0.2$.

В этой задаче требуется многократно повторять вычисление значения функции и печать аргумента и функции. Кроме того, необходимо после очередной печати изменять значения аргумента. Таким образом, операторы цикла определены!

Алгоритм может быть построен по любой типовой структуре. В качестве примера выбрана структура цикла ПОКА.

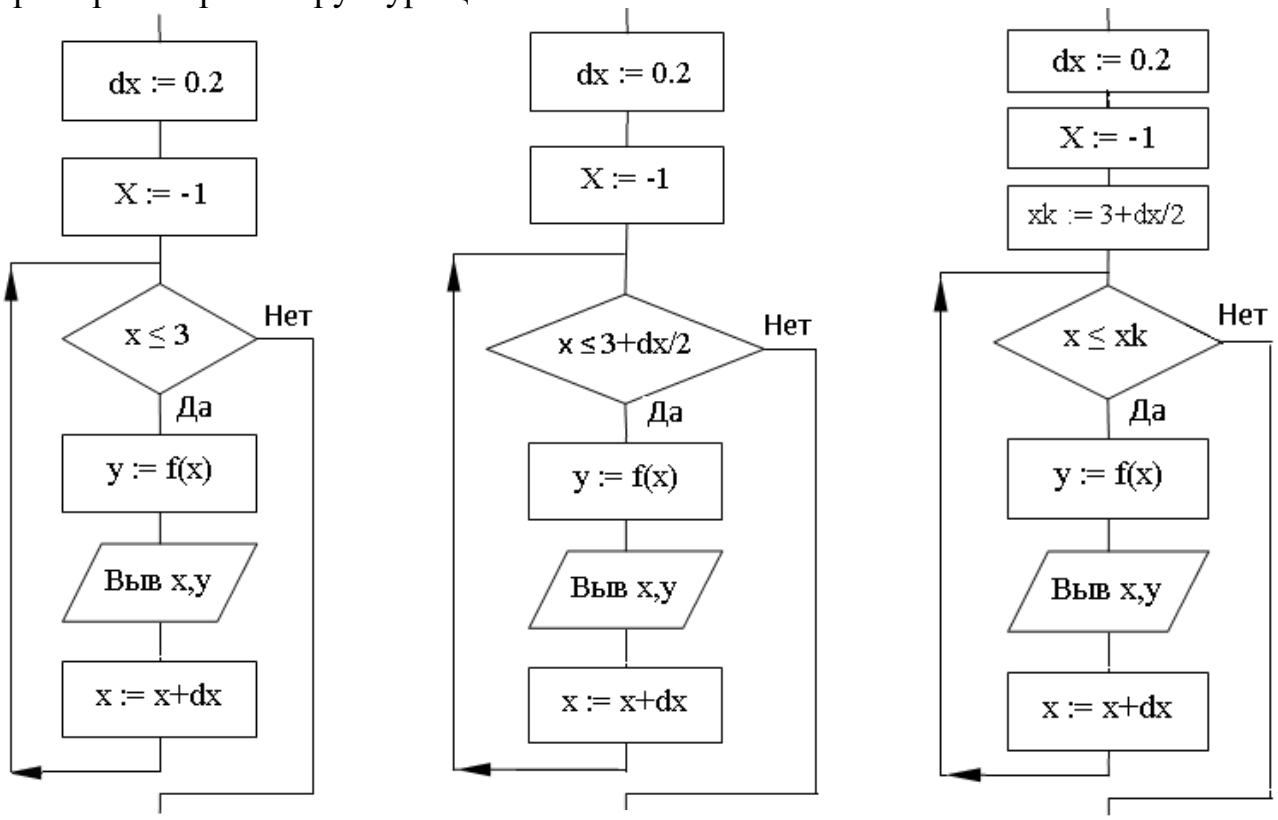


Рис. 15.

Все изображенные на Рис. 15. алгоритмы работоспособны, но каждый имеет свои особенности. Так, в левом алгоритме возможна ситуация, когда последнее значение функции, например, при $x = 3$, может быть не посчитано, следовательно, не вставлено в таблицу. Причина - вещественные числа в ЭВМ всегда представляются с погрешностью. И велика вероятность, что последнее получившееся x, например, $x = 3.0000001$ не будет использовано при вычислении функции. В среднем алгоритме эта ошибка устранена, но при каждом прохождении цикла в операторе ЕСЛИ будет вычисляться и проверяться выражение $x \le 3+dx/2$. В правом алгоритме данная неточность исправлена путем введения дополнительной переменной xk, вычисляемой непосредственно перед циклом.

Пример 5. В одномерном массиве A(10) найти максимальный элемент.

Прежде чем строить алгоритм решения данной задачи требуется определить переменную, в которой будет сохранен максимальный элемент. Первоначально в эту переменную может быть записан любой (например, первый) элемента массива A.

Операторами цикла будут действия, сравнивающие предполагаемый максимум с очередным членом массива. В том случае, если очередной член массива больше предполагаемого максимума, последний необходимо обновить.

Итак, словесное описание алгоритма сформулировано. Можно переходить к построению его блок-схемы. Как и для предыдущего примера, алгоритм может быть построен по типовой схеме цикла с предусловием (цикл ПОКА).

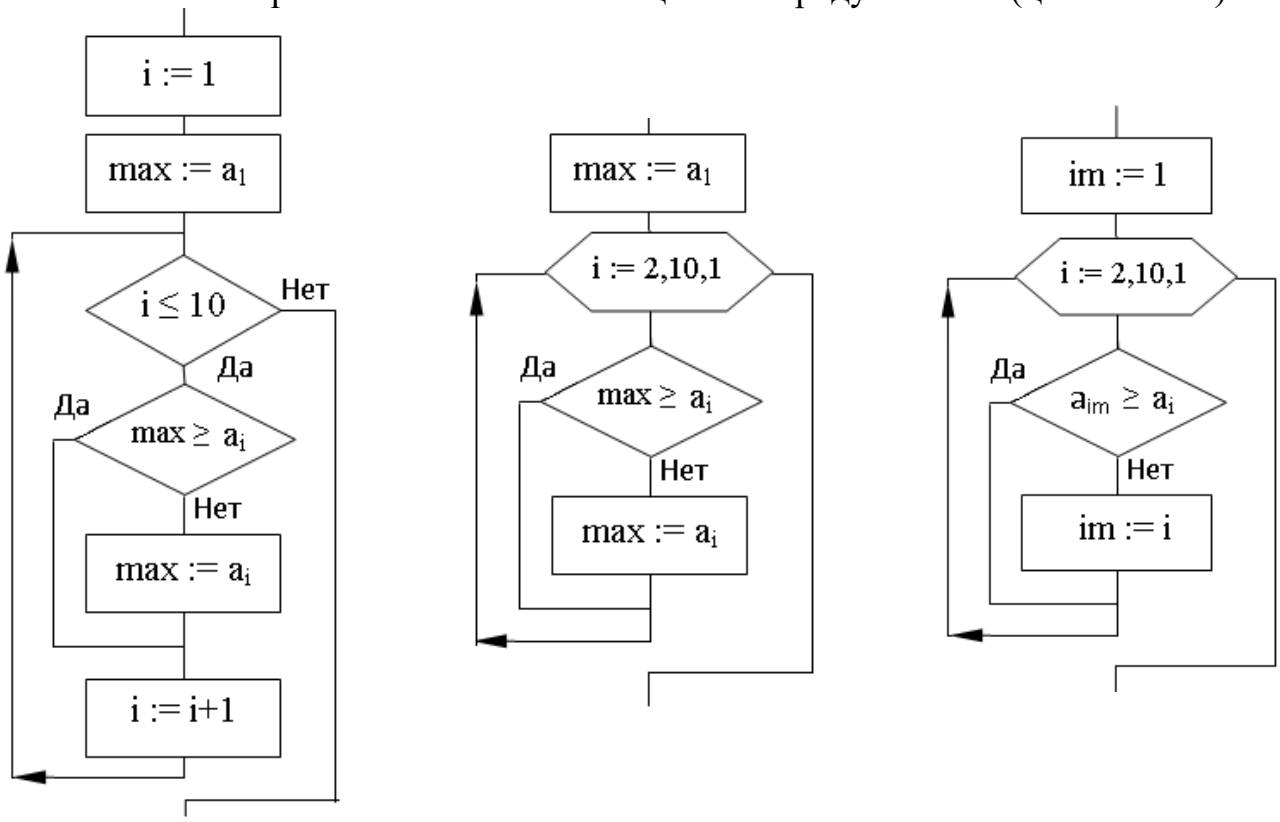


Рис. 16.

На Рис. 16 представлены три алгоритма решения задачи Пример 5. В изображенных схемах операторами цикла является блок ВЕТВЛЕНИЕ, в котором уточняется максимальный элемент в случае, если очередной элемент массива оказался больше него.

Операторами начального присвоения является инициализация переменной *max* первым элементом массива *A* и присвоение индексу очередного сравниваемого элемента единицы (*i* := 1). Следует отметить, что здесь допущена незначительная ошибка, учтенная в среднем и правом алгоритмах Рис. 16. Правильнее было бы *i* := 2.

Оператором выхода из цикла является оператор ЕСЛИ, в котором номер элемента массива A сравнивается с 10 (номером последнего элемента).

Два оставшихся (средний и правый) алгоритма Рис. 16 изображены по типовой схеме АРИФМЕТИЧЕСКОГО цикла (цикла с параметром). Причем, по правому алгоритму отыскивается не значение максимального элемента, а его место в массиве A (индекс *im*).

Пример 6. Упорядочить элементы массива A(10) по убыванию.

Существует несколько способов упорядочивания массива. Наиболее распространенным считается метод "пузырька". Многим нравится метод Вставок. Но самым понятным следует назвать метод Выбора. Идея этого метода заключается в том, что сначала среди всех элементов массива ищется наибольший, который меняется местами с первым элементом. Затем ищется максимальный среди всех элементов, за исключением первого (начиная со 2-го), и происходит его перестановка со вторым. Целиком вся процедура упорядочивания понятна из алгоритма, изображенного на Рис. 17.

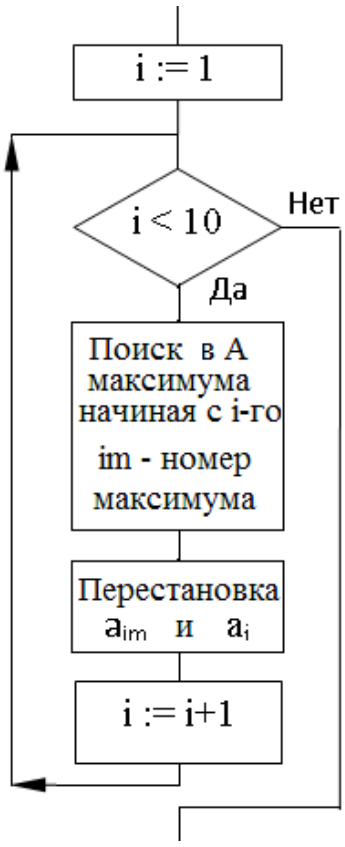


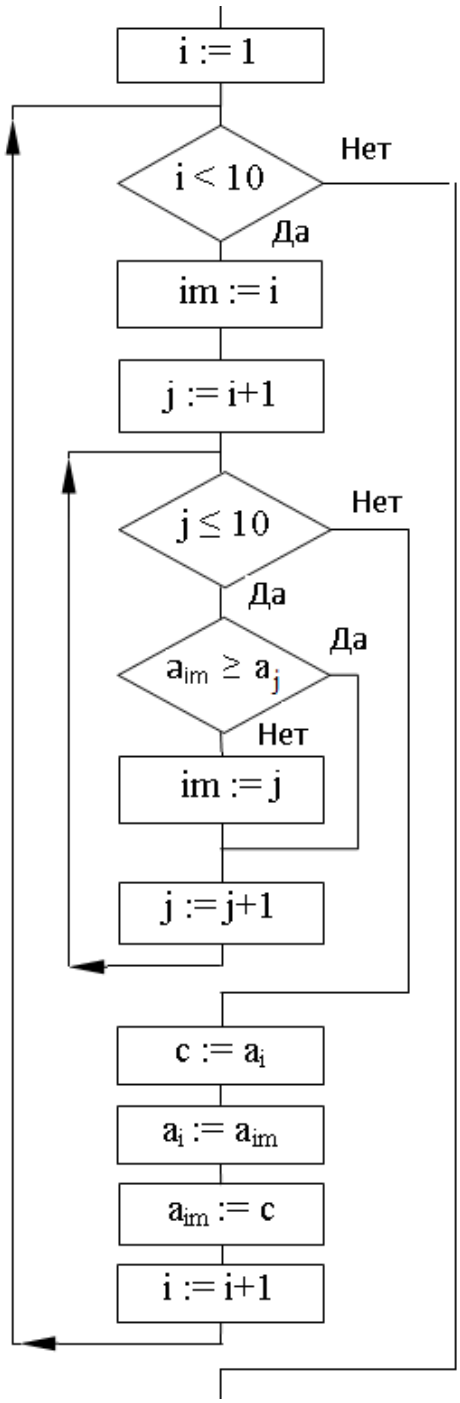
Рис. 17.

Алгоритм поиска максимального элемента рассмотрен ранее (см. Пример 5.). Его необходимо подкорректировать, чтоб максимум искался среди элементов, начиная не с 1-го, а i-го, и заменить переменную цикла i на j.

Процедура перестановки двух элементов сложности не вызывает.

Ниже, на Рис. 18., представлен весь алгоритм решения задачи Примера 6.

Алгоритм упорядочивания массива также может быть построен с использованием специализированных функций, таких как Функция поиска номера максимального элемента в массиве A начиная с i-го и Функция перестановки значений двух вещественных (или целых) чисел.



Правый алгоритм упорядочивания массива построен по типовой схеме АРИФМЕТИЧЕСКИЙ цикл. Последовательность операторов аналогична алгоритму, изображенному на левой схеме. Обе схемы включают в себя два цикла (второй, внутренний, вложен в первый) и оператор Ветвления во внутреннем цикле. Назначение циклов и оператора ЕСЛИ понятно из схемы алгоритмов.

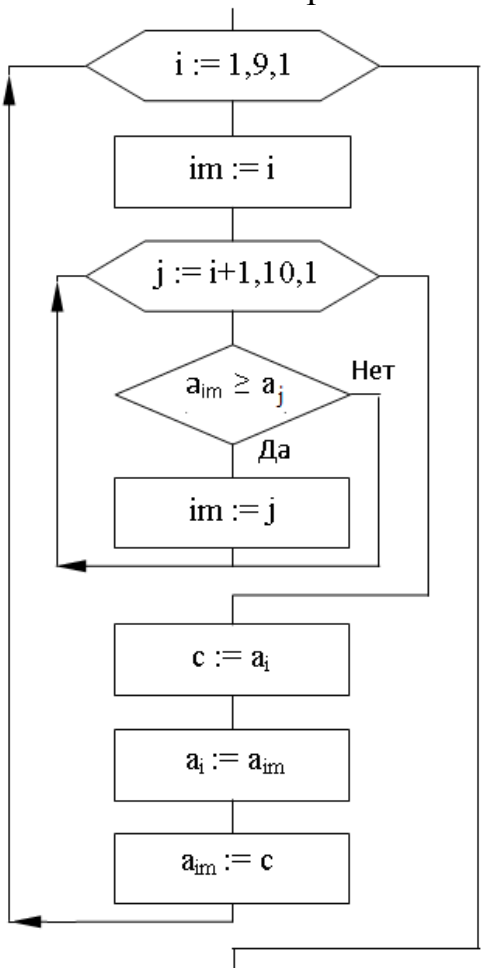


Рис. 18.

Пример 7. Вычислить сумму элементов матрицы A(10x10).

Сумма элементов матрицы находится путем добавления значения соответствующего элемента матрицы к переменной Sum. Первоначально Sum должна быть инициализирована Нулем (Sum := 0). Добавление выполняется построчно сверху вниз (внешний цикл). Внутри каждой строки - слева направо (внутренний цикл). Первый индекс элемента A_{ij} матрицы соответствует номеру строки, в которой находится данный элемент. Второй индекс равен номеру столбца.

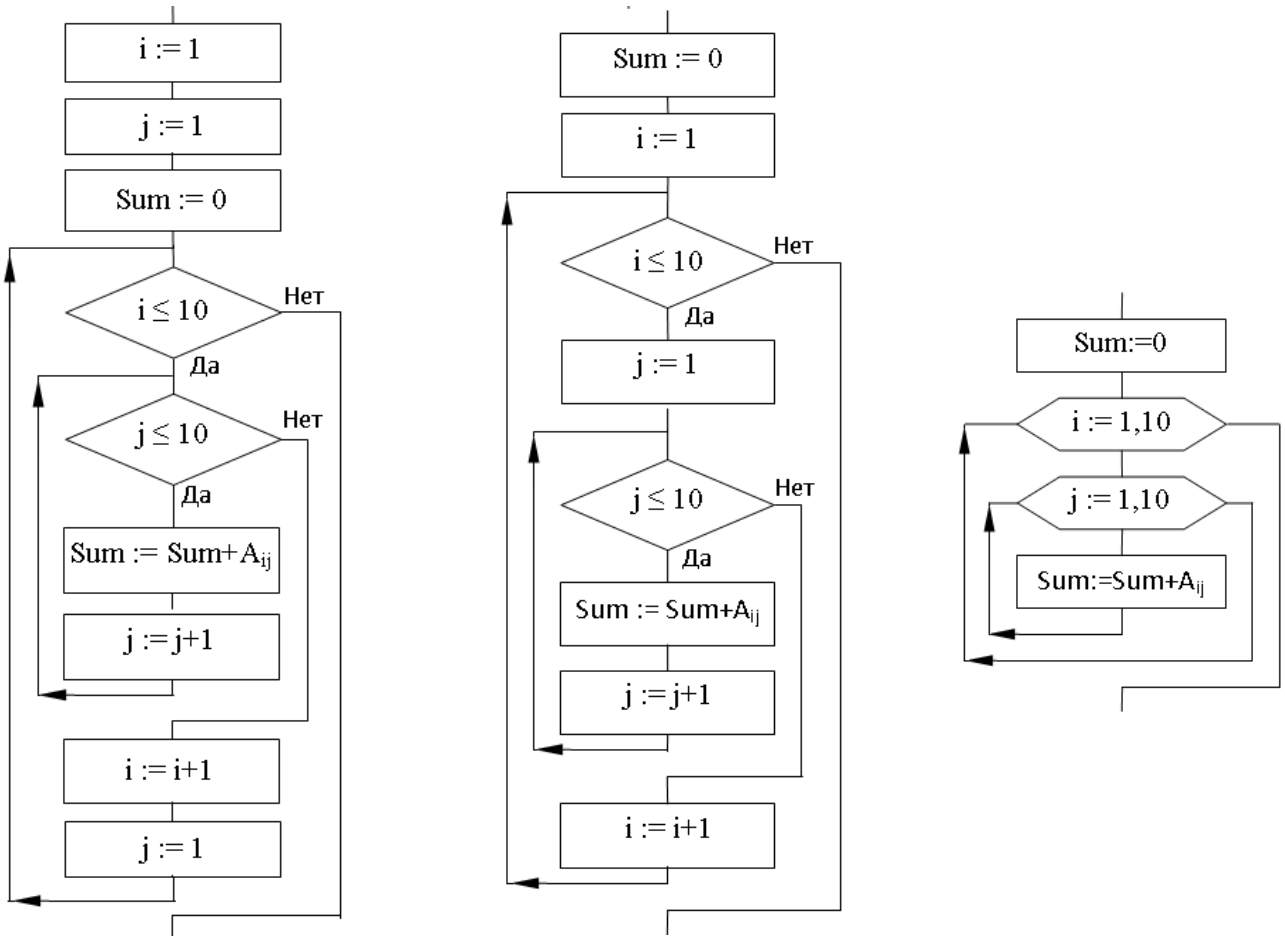


Рис. 19.

На Рис. 19. изображены три схемы алгоритма. Левая и средняя построены по типовой схеме цикла ПОКА. Правая - АРИФМЕТИЧЕСКИЙ цикл. По последовательности операторов она полностью идентична алгоритму, изображенному на средней схеме Рис. 19.

Левая схема, несмотря на то что "работает", имеет существенную ошибку: Внутренний цикл не содержит оператора начального присвоения. Этот "потерянный" оператор (j :=1) расположен после оператора i := i+1 во внешнем цикле, что позволяет студентам настаивать на том что схема - "работающая". Но ни в одном из языков программирования нет оператора цикла, с помощью которого можно было бы реализовать данную схему!!!

В среднем алгоритме эта ошибка устранена.

Пример 8. Вычислить суммы элементов строк матрицы A(10x10).

Задача практически повторяет предыдущую (Пример 7). Алгоритм строится также с помощью двух циклов (внешнего и внутреннего). Внутренний цикл служит для добавления значения элементов строки, определяемой внутренним циклом, к переменной Sum. Отличием является то, что переменную Sum необходимо обнулять перед внутренним циклом. (Перед циклом, а не после него!!!!). Кроме этого, требуется организовать вывод значения этой переменной после окончания внутреннего цикла. Если выводить значения суммы элементов каждой строки не требуется, то для получения сумм можно исполь-

зовать массив Sum, количество элементов которого совпадает с числом строк матрицы A. И индекс i элемента этого массива (Sum_i) равняется номеру строки матрицы, элементы которой суммируются. Во внешнем цикле как раз и происходит изменение индекса строки матрицы A.

2.4. Циклы с неизвестным числом повторений

Все рассмотренные ранее примеры на циклические алгоритмы реализовывались (могли быть реализованы) по типовой схеме АРИФМЕТИЧЕСКИЙ цикл. Особенность их заключалась в том, что, так называемая ПЕРЕМЕННАЯ цикла изменяла своё значение от заданного начального до конечного значения с некоторым шагом, и для каждого значения этой переменной тело цикла выполнялось один раз.

Наряду с рассмотренными существуют и циклы с неизвестным числом повторений. Именно с помощью таких циклов алгоритмизируются задачи на вычисление суммы ряда с заданной точностью.

Предварительно имеет смысл рассмотреть упрощенную задачу, когда количество членов ряда ограничено, например, числом m:

Пример 9. Вычислить $y(x) = 1 - \frac{x}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \ldots + \frac{x^m}{m!}$.

Данная задача, как и все предыдущие, также решается с применением циклов с заданным числом повторений.

Прежде всего следует определить закономерность: Каждый очередной член ряда получается из предыдущего по формуле:

$$a_i = -a_{i-1} \cdot \frac{x}{i},$$

называемой рекуррентной.

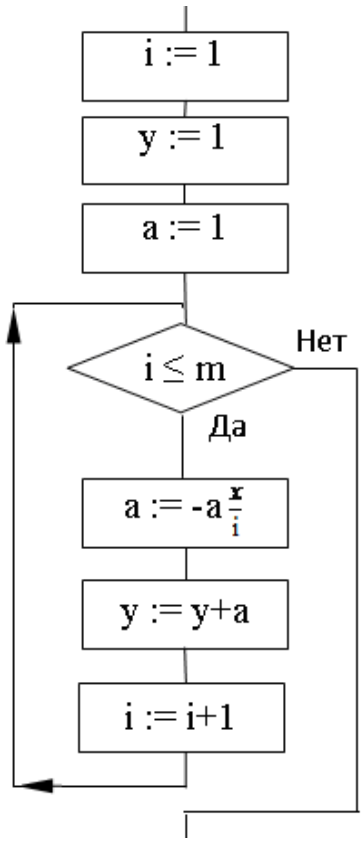
В отдельных случаях (для более сложных задач) бывает удобным получить рекуррентные формулы не для очередного члена ряда, а для его компонентов, например, числителя и знаменателя. Так, для задачи Примера 9 эти соотношения будут иметь вид:

$$a_i = \frac{ch_i}{zn_i}, \quad ch_i = -ch_{i-1} \cdot x, \quad zn_i = zn_{i-1} \cdot i,$$

где ch_i и zn_i - числитель и знаменатель i-го члена ряда соответственно.

На Рис. 20. изображены два варианта алгоритма вычисления конечной суммы ряда. В левой схеме использовано рекуррентное соотношение $a_i = -a_{i-1} \cdot \frac{x}{i}$, а в правой: $a_i = \frac{ch_i}{zn_i}$, где $ch_i = -ch_{i-1} \cdot x$, $zn_i = zn_{i-1} \cdot i$.

Правда, при таком представлении очередного члена есть опасность, что при очередном члене ряда стремящемся к нулю, числитель и знаменатель его будут стремиться к бесконечности. Значит, возможно переполнение разрядной сетки.



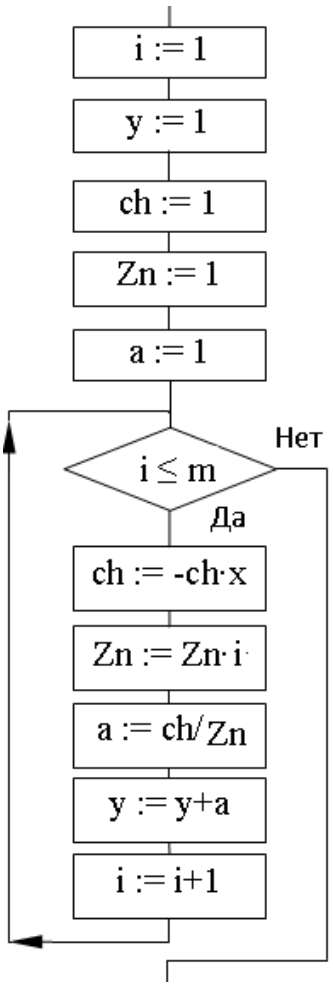


Рис. 20.

Следует обратить особое внимание на **важную закономерность**:

В теле цикла операторы должны располагаться в обязательной последовательности (удобнее пояснить этот факт на примере правой схемы): сначала вычисляется числитель очередного члена ряда, затем его знаменатель, далее собственно член и уточняется значение суммы ряда. И только последним оператором увеличивается значение i.

Данная закономерность должна соблюдаться при организации любого цикла. По крайней мере, оператор изменения управляющей переменной цикла должен быть последним в теле цикла.

Операторы начального присвоения, стоящие перед циклом, подбираются из условия, чтоб первое прохождение по циклу было правильным и соответствовало условию задачи Примера 9.

После того, как построен алгоритм вычисления суммы конечного числа членов ряда, можно переходить и к циклам с неизвестным числом повторений.

Пример 10. Вычислить $y(x) = 1 - \frac{x}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \ldots + \frac{x^m}{m!} - \ldots$ с относительной погрешностью ε.

Задача данного Примера отличается от предыдущей тем, что вычисления необходимо прекратить, когда относительная величина уточнения решения сравнится с заданной погрешностью, то есть выполнится неравенство:

$$\left|a_i / y\right| \le \varepsilon .$$

Алгоритм удобнее строить по типовой структуре цикла ДО. На Рис. 21. приведены три варианта разработанного алгоритма. Первые две схемы соответствуют методике, рассмотренной в Примере 9. Операторы цикла выполняются до тех пор, пока неравенство $\left|a_i / y\right| > \varepsilon$ выполняется.

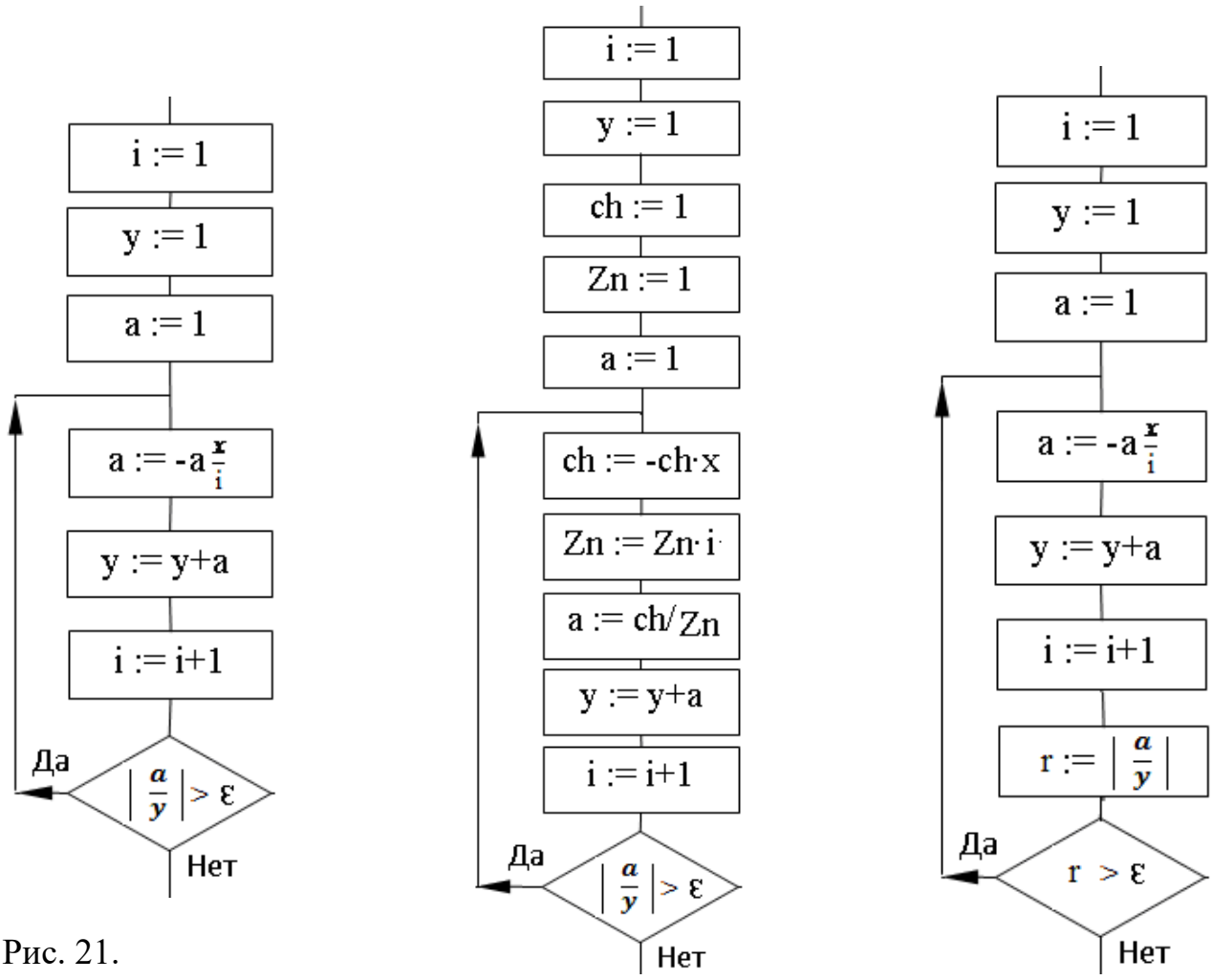


Рис. 21.

В третьем (правом) алгоритме, с целью упрощения условия выхода из цикла, введена переменная r, значение которой принимается равным величине относительной погрешности вычисления суммы ряда. Оператор, в котором это значение вычисляется, стоит одним из последних в теле цикла.

Данное упрощение может быть рекомендовано для условий проверки выхода из циклов обоих типов: ПОКА и ДО. С этой целью вводятся специальные переменные, называемыми ФЛАГАМИ, а в тело цикла добавляются операторы, в которых вычисляются значения этих флагов. Такое дополнение существенно упрощает условие проверки выхода из цикла, как это показано на правой схеме Рис. 21.

Замечание! Задача Примера 10 обладает одной неприятной особенностью: при определенном сочетании входных параметров возможно "зацикливание" алгоритма! Для исключения его в алгоритм следует добавить оператор, контролирующий максимальное количество прохождений по циклу. Это значение можно определить постоянной величиной, например 100 (или 1000), либо ввести как входной параметр: i_{max}. Справа, на Рис. 22., показан фрагмент алгоритма, в котором данное замечание учтено. Оператор Fl := 1 лучше вынести за пределы цикла (поставить непосредственно перед циклом).

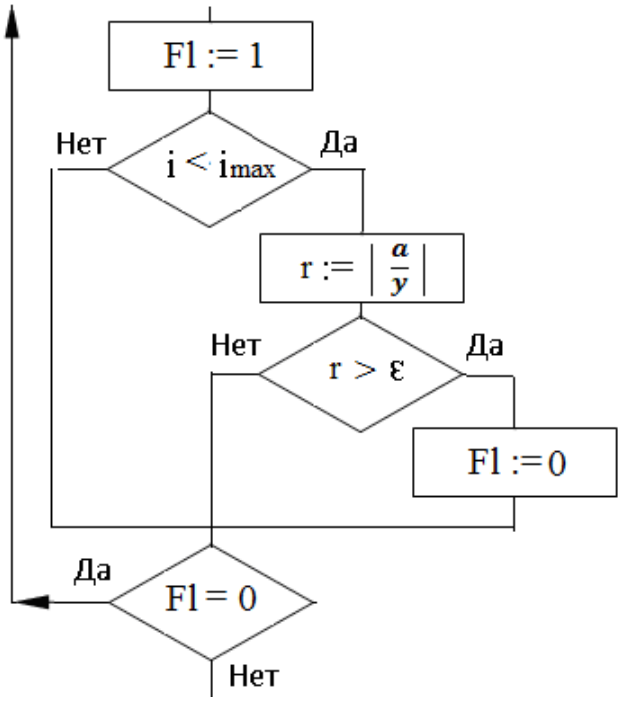


Рис. 22.

2.5. Рекурсии (рекурсивные алгоритмы)

Рекурсия - вычислительный процесс, направленный на решение определенной задачи таким образом, что само решение использует этот же процесс, решающий аналогичную подзадачу.

В программировании под рекурсией понимают такую реализацию, в которой подпрограмма использует в своем теле вызов самой себя. Такие вызовы называют рекурсивными. Когда функция А в своем теле вызывает только одну рекурсивную функцию (саму себя), то говорят о простой рекурсии. Под косвенной рекурсией понимают явление, когда рекурсивные функции вызывают друг друга (например, функция А вызывает функцию В, а функция В вызывает А).

Тема рекурсии достаточно сложна и обычно возникают трудности в процессе проектирования рекурсивных программ. Рекурсивные алгоритмы сложно отлаживать, но порой они позволяют очень гибко и красиво решить задачу. Правда, во многих случаях за внешней красотой может скрываться более длительный вычислительный процесс.

При реализации рекурсивных алгоритмов требуется уверенность, что алгоритм является конечным, то есть выполнение рекурсивной функции должно когда-нибудь завершиться. Конечность рекурсивного алгоритма обеспечивается с помощью специального (терминального, граничного) условия, реализуемого оператором ЕСЛИ. Естественно, помимо конечности алгоритм должен быть правильным.

Любой рекурсивный алгоритм можно заменить нерекурсивным (циклическим), несмотря на то, что на его реализацию может потребоваться большее

время. Во многих случаях проектирование циклического алгоритма позволяет проще и быстрее решить поставленную задачу.

На Рис. 23. приведены Циклический (слева) и Рекурсивный (справа) алгоритмы решения эталонной задачи - вычисления функционала: m!

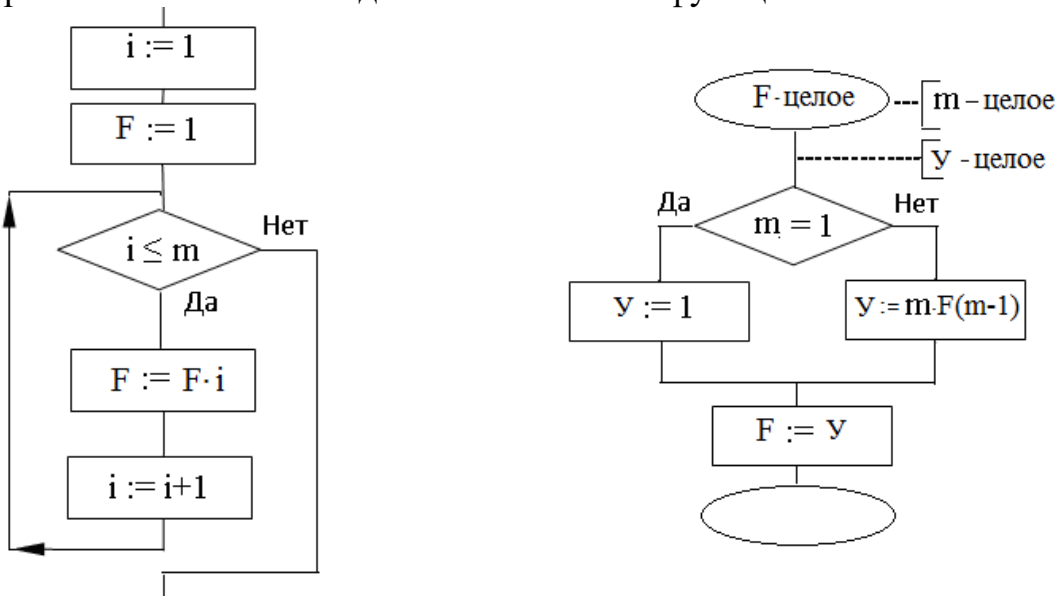


Рис. 23.

В главной программе вызов рекурсивной функции будет иметь вид: Y:=F(m).

Рекурсия может быть использована и для других циклических задач, например, для вычисления суммы ряда. Технология вычисления конечной суммы ряда (см. Примеры 9 и 10) отработана на примере циклических алгоритмов (Рис. 20. и Рис. 21. соответственно). В настоящем разделе рассматривается рекурсивная алгоритмизация этой задачи. На Рис. 24. приведены два рекурсивных алгоритма решения задачи Примера 9.

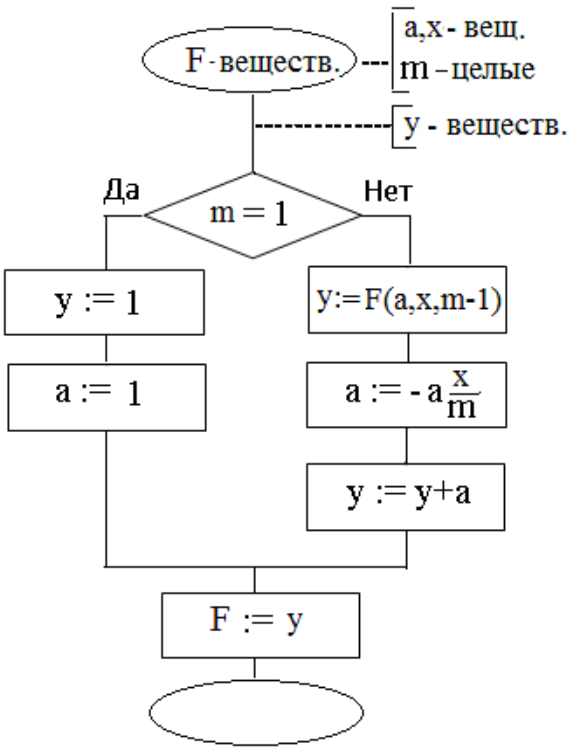
Оба алгоритма существенно отличаются друг от друга. Функции, построенные по ним, имеют разный набор параметров. Различными будут и вызовы этих функций. В главной программе вызвать левую функцию можно так: Y := F(a, x, m). Правая же функция должна вызываться при следующих значениях параметров: Y := F(1., 1., 1, x, m).

Алгоритм Рис. 24. б) при несущественной доработке может быть использован и для вычисления суммы ряда с заданной точностью (задача Примера 10). Для этого необходимо изменить всего лишь условие в операторе ЕСЛИ. Дополнительно увеличивается число параметров функции, - к уже имеющимся добавляется требуемая точность.

Следует заметить, что левый алгоритм (Рис. 24. а)) называется НИСХОДЯЩЕЙ рекурсией, а правый (Рис. 24. б)) - ВОСХОДЯЩЕЙ.

В нисходящей рекурсии основные вычисления (вычисление очередного члена ряда и добавление его к переменной, в которой накапливается сумма ряда) выполняются после рекурсивного обращения к функции F. Кроме того, один из параметров функции, а именно a является выходным параметром.

Функция F, кроме уточняемой (накапливаемой) суммы ряда, возвращает и значение предыдущего члена ряда.



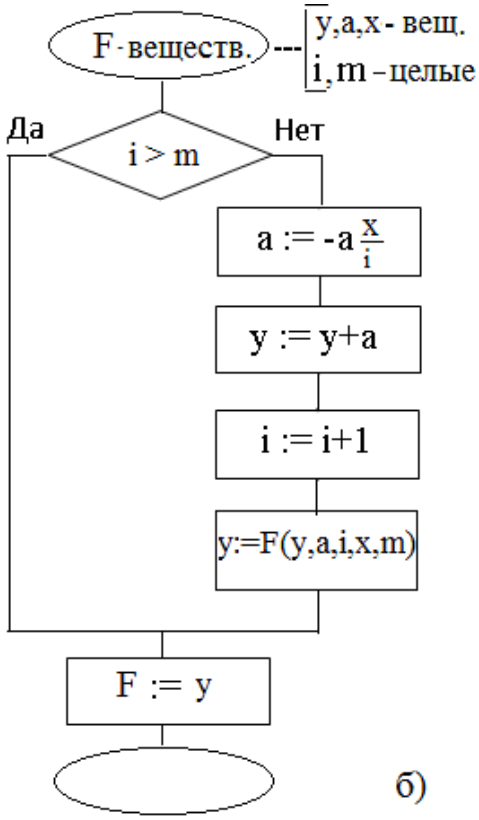


Рис. 24. а) б)

При восходящей рекурсии вычислительный процесс начинается сначала, как в циклическом алгоритме. Номер шага постоянно возрастает: i:= i+1. В качестве терминальной (граничной) ситуации выбирается момент достижения условий окончания процесса i>m. Начальные условия задаются в качестве значений аргументов при вызове рукурсивной функции. В восходящей рекурсии параметры, характеризующие состояние процесса, вычисляются на каждой стадии рекурсии в процессе выполнения прямого хода. Результат строится постепенно и окончательное значение приобретается в момент достижения терминальной ситуации. Таким образом, список аргументов функции, проектируемой по схеме восходящей рекурсии, содержит две группы параметров: одна - это исходные данные и результат, вторая - промежуточные значения переменных процесса.

Достоинством восходящей рекурсии является экономия памяти - стека, куда помещаются при рекурсивном вызове функции значения переменных и параметров, а также еще невыполненные операторы, что расположены после обращения к функции. Как следствие - восходящая рекурсия более эффективна по быстродействию. Поэтому во всех случаях, когда это возможно, рекомендуется переходить от нисходящей рекурсии к восходящей.

Важное замечание: Если есть возможность решить задачу без использования рекурсии, то лучше этой возможностью воспользоваться. Данный раздел описан только для того, чтоб понять, как создаются циклические алгоритмы, и показать, как они работают.

3. ТЕСТИРОВАНИЕ И ОТЛАДКА АЛГОРИТМОВ

Процесс создания алгоритма, как и любой этап разработки программного обеспечения, не обходится без ошибок. Фирма *Hewlett-Packard* установила процентное соотношение ошибок (рис. 25.), обнаруживаемых в ПО на разных стадиях разработки. Это соотношение - типичное для многих фирм, производящих ПО.

Рис. 25. Процентное соотношение ошибок при разработке ПО

Как видно из рисунка, почти треть всех ошибок приходится на этап построения алгоритма. Следовательно, полученный в результате алгоритмизации готовый алгоритм необходимо подвергнуть тщательной проверке. Исследования IBM показали, что чем позже обнаруживается ошибка в программе, тем дороже обходится ее исправление; эта зависимость близка к экспоненциальной.

Проверка правильности алгоритма заключается в его тестировании, верификации, валидации и отладке.

3.1. Тестирование алгоритма

Тестирование алгоритма позволяет выявить логические ошибки его построения. Подобные ошибки обнаруживаются при тестировании алгоритма различными тестовыми наборами данных.

Различают структурное и функциональное тестирование.

Структурное тестирование основывается на детальном изучении логики алгоритма и подборе тестов, позволяющих обеспечить максимально возможное количество проверяемых операторов, логических ветвлений, условий и циклов.

При **функциональном тестировании** логика алгоритма не учитывается, а обращается внимание лишь на входные и выходные спецификации (значения переменных). Функциональное тестирование алгоритма – это проверка пра-

вильности или неправильности работы алгоритма на специально заданных тестовых примерах – задачах с известными входными данными и результатами (иногда достаточны их приближения).

Перед структурным тестированием необходимо визуально проверить как структуру алгоритма в целом, так и отдельных его блоков. В большинстве случаев каждый блок алгоритма - "подалгоритм" (логически завершенный фрагмент) должен иметь один вход и один выход.

Как уже отмечено, ошибки в алгоритме обнаруживаются при проверке его работы на большом количестве различных тестовых наборов данных. Следовательно, эффективное проектирование текстовых наборов данных - задача достаточно сложная.

Тестовый набор должен быть минимальным и полным, то есть обеспечивающим проверку каждого отдельного типа наборов входных данных, особенно исключительных случаев.

При использовании структурного тестирования для построения тестовых наборов данных возможно использование следующих критериев:

- набор данных должен обеспечивать выполнение каждого оператора, по крайней мере, один раз;
- тестовые наборы данных в узлах ветвления с более чем одним условием должны обеспечивать принятие каждым условием значения истина или ложь хотя бы по одному разу;
- тестовые наборы данных в узлах ветвления с более чем одним условием должны обеспечивать перебор всех возможных сочетаний значений условий в одном узле ветвления.

Построение тестовых наборов данных опирается на так называемые контрольные примеры.

Контрольные примеры (тесты) – это специально подобранные задачи, результаты которых заранее известны или могут быть определены без существенных затрат.

Наиболее простые способы получения тестов:

- Подбор исходных данных, для которых несложно определить результат вычислений вручную или расчетом на калькуляторе.
- Использование результатов, полученных на других ЭВМ или по другим программам.
- Использование знаний о физической природе процесса, параметры которого определяются, о требуемых и возможных свойствах рассчитываемой конструкции. Хотя точное решение задачи заранее известно, суждение о порядке величин позволяет с большой вероятностью оценить достоверность результатов.

В общем случае тестирование всех возможных вариантов невозможно, хотя бы потому что их бесконечное множество. Поэтому тестируют ключевые, особые случаи, так называемые точки бифуркации.

Пример. Для задачи решения квадратного уравнения $ax^2 + bx + c = 0$ такими исключительными случаями, например, будут:

- $a = b = c = 0$;
- $a = 0$, b, c – отличны от нуля;
- $D = b^2 – 4ac < 0$ и др.

Для несложных алгоритмов грамотный подбор тестов и полное тестирование может дать полную картину их работоспособности (неработоспособности).

При тестировании сложных алгоритмов различают уровни тестирования:

- **Модульное тестирование** (Unit Testing)
  Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в отдельных частях (блоках) алгоритма, которые доступны и могут быть протестированы по отдельности.
- **Интеграционное тестирование** (Integration Testing)
  Проверяется взаимодействие между компонентами алгоритма после проведения компонентного (модульного) тестирования.

Сложные алгоритмы бывает удобным представить как последовательность (набор, комбинацию) более простых "подалгоритмов". Разбиение одного монолитного алгоритма на несколько маленьких позволяет упростить процедуру тестирования и поиска возможных ошибок, выполняя тестирование каждого отдельного блока независимо от остальных. В этом случае тестировать придется не только сами "подалгоритмы", но и точки перехода от одного к другому.

В тестировании отдельных фрагментов сложного алгоритма самое важное — это протестировать все возможные сценарии выполнения. Иными словами, нужно не просто покрыть все ветки алгоритма, но и все возможные их комбинации. Такой прием, конечно, никак не может уменьшить количество тестов при сохранении эквивалентного покрытия.

Следует иметь ввиду, что даже использование большого объема входных (тестовых) данных не гарантирует выявление всех ошибок, так как даже видимое 100% покрытие вовсе не означает, что все возможные комбинации были протестированы, особенно для достаточно сложных алгоритмов.

3.2. Трассировка

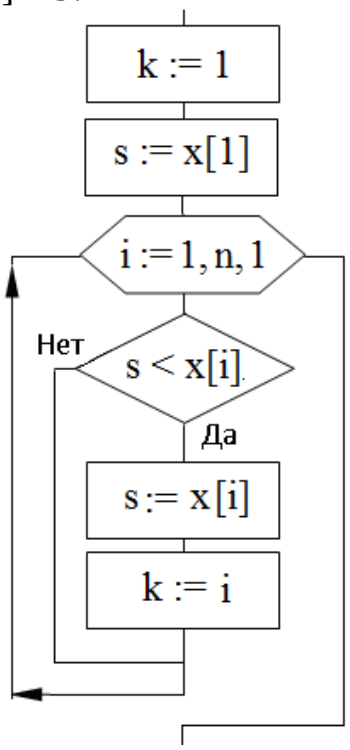
Для проверки правильности работы алгоритма, особенно при функциональном тестировании, рекомендуется пошагово отслеживать изменение всех переменных после выполнения каждого оператора. Такой процесс называется трассировкой.

Трассировка – это метод пошаговой фиксации динамических составляющих алгоритма на некотором тексте. Трассировка - проверка переменных и их значений в процессе пошагового выполнения всех операторов (блоков) алгоритма. Трассировка осуществляется с помощью трассировочных таблиц, в которых каждая строка соответствует определённому состоянию алгоритма, а столбец – определённому состоянию параметров алгоритма (входных, выходных и промежуточных переменных). Трассировка облегчает тестирование и понимание алгоритма, особенно когда написанная по этому алгоритму программа выдает ошибочный, не ожидаемый результат. Осуществляя пошаговую трассировку, разработчик вникает в логику работы алгоритма, проверяя на каждом шаге, правильны ли были его рассуждения при создании этого алгоритма.

Использование трассировочных таблиц удобно применять при тестировании всевозможных циклов, а именно проверять правильность вхождения в цикл, своевременность выхода из цикла, в том числе отсутствие зацикливания.

Пример. Определим функцию фрагмента алгоритма вида (см. рис.) на тесте:

n = 2; x[1] = 4; x[2] = 9:



Если выписать трассировочную таблицу вида:

i	S	x[i]	K	s<x[i]	i<=n
1	4	4	1	Нет	Да
2	4	9	2	Да	Да
3	—	—	—	—	Нет

то функция алгоритма становится более понятной – она состоит в нахождении индекса максимального элемента ряда.

Трассировочная таблица моделирует работу процессора во время выполнения программы (алгоритма). Данная таблица, кроме значений переменных, определенных в алгоритме (программе), может содержать столбы «Команда алгоритма», куда помещается очередная команда, и «Выполняемое действие» для отображений действий, которые выполняет арифметико-логическое устройство процессора. Таким образом, алгоритм совместно с трассировочной таблицей полностью моделируют процесс обработки информации, происходящий в компьютере, позволяя провести как структурное, так и функциональное тестирование этого алгоритма.

3.3. Верификация, валидация и отладка

Одновременно с проверкой логики работы алгоритма выполняется более строгая проверка – верификация, которая представляет собой теоретическое обоснование правильности вычислений, реализованных алгоритмом, и включает логико-математическое доказательство того, что вычислительное поведение алгоритма верно.

Верификация и Валидация есть Технологии проверки и подтверждения, используемые в процессе разработки программного продукта, позволяющие выявлять ошибки на ранних этапах проектирования. Большинство ошибок возникает на этапе формирования первичной спецификации, однако проявляются только при тестировании. Используя модели для виртуального тестирования на ранних этапах проектирования, специалисты могут существенно сократить время разработки.

Операции проверки, подтверждения и тестирования можно выполнять на всех этапах процесса проектирования ПО.

Верификация (verification, доказательство правильности) — проверка, проверяемость, способ подтверждения, проверка с помощью доказательств, каких-либо теоретических положений, алгоритмов, программ и процедур путем их сопоставления с опытными (эталонными или эмпирическими) данными, алгоритмами и программами. Принцип верификации был выдвинут Венским кружком (1924 г.).

Верификация – это процесс оценки того, насколько алгоритм по итогам его работы соответствует заданным условиям, некоторой математической модели конкретной системы, процесса.

Наряду с верификацией алгоритма выполняется его валидация.

Валидация (validation) — это определение соответствия разрабатываемого алгоритма (и соответствующего программного продукта) ожиданиям и потребностям пользователя, требованиям к системе.

Другими словами, верификация - это проверка корректности выполнения отдельного этапа (блока) алгоритма (программного продукта), а валидация - это проверка соответствия продукта (ПО) целям создания системы, как их понимает пользователь и/или заказчик.

Поэтому процедуре валидации, как правило, подвергается весь программный продукт, вся система, компонентом которой это ПО является. Это своего рода заключительный этап разработки программ, реализуемый в форме испытаний: предварительных, приемочных, сертификационных, комплексных и так далее, в определенных эксплуатационных режимах.

Однако валидация выполняется и на более ранних этапах, при проверке разработанного алгоритма.

Верификация (и валидация) проводится при функциональном тестировании алгоритма с использованием трассировочных таблиц по специально подобранным тестам.

Иногда с целью верификации алгоритмов используют прием экспериментального тестирования программ, написанных по данному алгоритму: проверяемую программу неоднократно запускают с теми входными данными, относительно которых результат известен заранее. Затем сравнивают машинный результат с ожидаемым. Если результаты совпадают, то появляется небольшая уверенность, что и последующие вычисления не приведут к ошибкам.

Для сложных алгоритмов удачное тестирование не дает гарантии того, что программа (алгоритм) не содержит ошибок.

Некоторые скрытые ошибки в сложных программах могут выявиться только в процессе их эксплуатации, на этапе сопровождения. На этом этапе программисты совместно с пользователями уточняют и улучшают разработанный алгоритм.

Процесс поиска и исправления (явных или неявных) ошибок в алгоритме называется **отладкой алгоритма**.

Задачи отладки. При всех методах отладки алгоритмов система отладки должна решать следующие задачи:

- обнаруживать наличие ошибок;
- проводить диагностику и локализацию ошибок;
- устранять ошибки и корректировать алгоритмы и программы, а также соответствующую документацию;
- информировать оператора о ходе, вычислений, переменных величинах и передачах управления в отлаживаемых алгоритмах и программах.

Самыми сложными среди перечисленных являются задачи обнаружения, диагностики и локализации ошибок.

Некоторые (скрытые, труднообнаруживаемые) ошибки в сложных программных комплексах могут выявиться только в процессе их эксплуатации, на последнем этапе поиска и исправления ошибок – этапе сопровождения. На этом этапе также уточняют и улучшают документацию, обучают персонал использованию алгоритма (программы).

4. СЛОЖНОСТЬ АЛГОРИТМА

4.1. О необходимости оценки алгоритмов

Как уже было сказано, алгоритм — точное предписание, задающее последовательность действий для решения конкретной задачи. Но большинство стоящих перед человеком задач может быть решено разными способами, то есть по разным алгоритмам. Следовательно, имеет место проблема анализа алгоритмов, оценки их качества, сравнения различных алгоритмов друг с другом. Подобная задача возникает и при анализе алгоритмов решения задач на ЭВМ.

Существует ряд важных практических причин для анализа алгоритмов. Основной из них является необходимость получения оценок объема памяти или времени работы, которое потребуется алгоритму для успешной обработки конкретных данных.

При разработке алгоритмов возникает проблема выбора между объёмом памяти и скоростью работы получаемой программы. Задачу можно решить быстро, используя большой объём памяти, или медленнее, занимая меньший объём.

Типичным примером служит алгоритм поиска кратчайшего пути. Представив карту города в виде сети, можно написать алгоритм для определения кратчайшего расстояния между двумя любыми точками этой сети. Чтобы не вычислять эти расстояния всякий раз, можно вычислить кратчайшие расстояния между всеми точками и сохранить результаты в таблице. Когда пользователю понадобится узнать кратчайшее расстояние между двумя заданными точками, берется готовое расстояние из таблицы. Результат получается мгновенно, но данный способ решения задачи требует огромного объёма памяти.

Из рассмотренного примера вытекает проблема объёмно-временной сложности: алгоритм оценивается как с точки зрении скорости выполнения, так и с учетом потреблённой памяти.

Для оценки алгоритмов вводятся следующие критерии и понятия: Трудоемкость, Стоимость, Эффективность, Сложность алгоритма, Вычислительная сложность, Порядок сложности...

4.2. Трудоемкость и стоимость алгоритма

Трудоемкость алгоритма — это количество операций, которые необходимо выполнить в процессе решения поставленной задачи с помощью разработанного алгоритма.

Практические оценки трудоемкости вычислительных алгоритмов, как правило, осуществляются раздельно для разных операций с группированием только однородных действий, таких как сложение и вычитание. Операции умножения и деления существенно различаются по длительности, поэтому оцениваются раздельно. В алгоритмах сортировки следует раздельно учитывать количества операций сравнения и перестановок. В алгоритмах поиска до-

статочно оценить только количество операций сравнения. Следовательно, для оценки трудоемкости алгоритма отдельно определяется число операций сложения и вычитания, умножения и деления, количество операций сравнения и перестановок.

Трудоемкость алгоритма зависит от свойств исходных данных:

- объем данных (длина входа);
- конкретика значений данных;
- порядок поступления данных с конкретными значениями.

Объем исходных данных влияет на трудоемкость практически всех алгоритмов, но и конкретика данных весьма существенна.

В зависимости от конкретных значений исходных данных различают:

- лучший случай,
- средний случай,
- худший случай.

Например, количество операций при вычислении значения полинома существенно зависит от количества нулевых коэффициентов, а при сортировке массива — от исходного порядка следования элементов. При благоприятном раскладе (**лучший случай**) перестановки данных могут не потребоваться вовсе. Если же все элементы массива расположены в обратном порядке, то для сортировки их потребуется максимальное количество операций (**худший случай**). Рассматривается и так называемый **средний случай**, когда конкретика данных и их значений заранее неизвестна.

С трудоемкостью алгоритма связано понятие стоимости, понимаемой как оценка используемых ресурсов. **Стоимость алгоритма** позволяет оценить время выполнения и объем требуемой памяти (если речь идет о выполнении задачи на ЭВМ).

Трудоемкость может быть оценена не только в количестве конкретных операторов, но и во временных единицах (секунды, миллисекунды, число временных тактов процессора, число выполнения циклов и т.д.). - **временная стоимость** алгоритма.

Для оценки объема требуемой памяти используется **емкостная стоимость**, вычисляемая в битах, байтах, словах....

На стоимость алгоритма влияют факторы:

- быстродействие компьютера и его емкостные ресурсы (в первую очередь — объем оперативной памяти). В самом деле, чем ниже тактовая частота процессора, чем меньше объем оперативного запоминающего устройства, тем медленнее выполняются арифметические и логические операции, тем чаще (для больших задач) приходится обращаться к медленно действующей внешней памяти, и, следовательно, большее время затрачивается на реализацию алгоритма;
- выбранный язык программирования. Задача, запрограммированная, например, на языке Ассемблер, в общем случае решится быстрее, чем

по тому же самому алгоритму, но запрограммированному на языке более высокого уровня, например на С;

- выбранный математический метод решения задачи.

4.3. Сложность и эффективность алгоритма

Основным критерием оценки качества алгоритма является **сложность** алгоритма. Наряду с этим качеством используется обратное понятие: **эффективность** алгоритма. Именно трудоемкость алгоритма определяет его сложность. Чем большее время и объем памяти необходимы для реализации алгоритма, тем алгоритм сложнее и, соответственно, ниже его эффективность. Сложность алгоритма (по аналогии с трудоемкостью) подразделяется на временную и емкостную. Временная сложность — это критерий, характеризующий временные затраты на реализацию алгоритма. Емкостная сложность — критерий, характеризующий затраты памяти на те же цели.

Временная сложность определяется количеством элементарных операций (инструкций), совершаемых алгоритмом для решения им поставленной задачи. Этим показателем определяется и время выполнения будущей программы. Разумеется, в размерных единицах времени (секундах) оно зависит еще и от скорости работы процессора (тактовой частоты). Для того чтобы показатель временной сложности алгоритма был инвариантен относительно технических характеристик компьютера, его измеряют в относительных единицах: в количестве выполняемых операций.

Показатель временной сложности становится особенно важным для задач, предусматривающих интерактивный режим работы программы, или для задач управления в режиме реального времени. Часто программисту, составляющему программу управления каким-нибудь техническим устройством, приходится искать компромисс между точностью вычислений и временем работы программы. Как правило, повышение точности ведет к увеличению времени.

Емкостная сложность характеризует объем памяти, который необходим для реализации алгоритма. В общем случае требуемый объем включает в себя память, необходимую для программы, исходных данных, а также промежуточных и конечных результатов. Как уже было отмечено, объем требуемой памяти измеряется в битах, байтах, страницах, килобайтах, мегабайтах...

Емкостная сложность программы становится критической, когда объем обрабатываемых данных оказывается на пределе оперативной памяти ЭВМ. На современных компьютерах острота этой проблемы снижается благодаря как росту объема ОЗУ, так и эффективному использованию многоуровневой системы запоминающих устройств. Программе оказывается доступной очень большая, практически неограниченная область памяти (виртуальная память). Недостаток основной памяти приводит лишь к некоторому замедлению работы из-за обменов между оперативной памятью и диском. Существуют приемы, позволяющие минимизировать потери времени при таком обмене. Использование кэш-памяти и аппаратного просмотра команд программы на требуемое

число ходов вперед позволяет заблаговременно переносить с диска в оперативную память нужные значения.

Программисты обычно сосредоточивают внимание на скорости алгоритма (времени выполнения задачи), учитывая, конечно же и требования к объёму памяти, свободному месте на диске, потому как использование быстрого алгоритма не приведёт к ожидаемым результатам, если для его работы понадобится больше памяти, чем есть у компьютера. Также следует иметь ввиду, что сложная программа может выполняться быстро, а простая — очень долго.

Наряду с рассмотренными существенными характеристиками алгоритма рассматриваются его структурная и когнитивная сложности.

Структурная сложность — характеристика количества управляющих структур в алгоритме (программе) и специфики их взаиморасположения.

При функционировании программы многообразие ее поведения и количество связей между ее входными и результирующими данными в значительной степени определяется ***набором маршрутов***, по которым исполняется программа.

То есть, ***сложность программного модуля связана*** не столько с размером (числом операторов), сколько ***с числом маршрутов*** ее исполнения и их сложностью.

Для повышения качества программы маршруты возможной обработки данных должны быть тщательно проверены (протестированы) на этапе разработки и отладки алгоритма, и тем самым определяют сложность его создания.

Все маршруты алгоритма условно можно разделить на две группы:

- вычислительные маршруты**;**
- маршруты принятия логических решений**.**

Вычислительные маршруты включают в себя маршруты арифметической обработки данных и предназначены для преобразования величин, являющихся квантованными результатами измерения непрерывных, как правило, физических характеристик.

Для проверки вычислительных маршрутов строится достаточно простая стратегия их тестирования. Во всем диапазоне входных переменных выбирается ***несколько характерных точек*** (предельные значения, значения в точках разрыва и несколько промежуточных значений), для которых проверяется работоспособность алгоритма.

В большинстве программных комплексов доля вычислительной части относительно невелика (общее число арифметических операций не выходит за предел 5–10%), поэтому вычислительные маршруты не определяют структурную сложность алгоритма.

Маршруты принятия логических решений (так называемые потоки управления) определяются результатом функционирования схем (блоков, функций) принятия решения и преобразования логических переменных. При этом каждое изменение любой логической переменной может определять разные области результирующих значений на выходе программы. Такое преобразование переменных обеспечивается алгоритмами со сложной логической

структурой, содержащей ряд проверок логических условий, циклов для поиска и отбора переменных, а также логические преобразования переменных. В результате образуется множество маршрутов обработки исходных данных, которые и определяют сложность структуры алгоритма.

Выделение маршрутов исполнения алгоритма (маршрутов принятия решения), минимально необходимых для его проверки, и оценки структурной сложности может осуществляться по различным критериям. Наилучшим является критерий, позволяющий выделить все реальные маршруты исполнения алгоритма при любых сочетаниях исходных данных. При этом формирование маршрутов зависит не только от структуры алгоритма, но и от значений переменных в различные моменты времени. Такое выделение маршрутов трудно формализовать, и оно представляется очень трудоемким для оценки показателей сложности структуры.

Сложность маршрутов принятия логических решений оценивается суммарным количеством ветвлений во всех маршрутах. Разные маршруты могут пересекаться, проходить через общие операторы, поэтому маршрутная сложность как правило превышает количество операторов логических решение (ветвления).

Для правильно структурированных алгоритмов (в которых отсутствуют циклы с несколькими выходами, нет переходов внутрь циклов или условных операторов и нет принудительных выходов из внутренней части циклов или условных операторов) сложность структуры будет определена простым суммированием всех операторов ветвления.

Структурная сложность алгоритма в первую очередь определяет трудоемкость тестирования и сопровождения программного модуля, вероятности пропущенных ошибок и затрат на разработку программы целом, а также позволяет оценить потенциальную надежность функционирования алгоритма (программы).

Когнитивная сложность — характеристика доступности алгоритма для понимания специалистами прикладных областей.

Когнитивная сложность - простота (англ. *cognitive complexity – simplicity*). Понятие когнитивной сложности было введено психологами в 1950-х годах и первоначально применялось преимущественно для понимания сложности структур и процессов организационной деятельности человека и сложности взаимодействия человека и компьютера.

В данном контексте под когнитивной сложностью алгоритма подразумевается степень понимания разработанного алгоритма, возможность его анализа прежде всего не программистами, а работниками других областей знаний, науки и техники, для которых разрабатывается конкретный алгоритм, программный комплекс. Чем понятнее написан, представлен, изображен алгоритм, тем проще и эффективнее будет проведено его тестирование совместными усилиями разработчиков и заказчиков программного обеспечения. Когнитивная сложность алгоритма, как и его структурная сложность, определяет

трудоемкость тестирования данного алгоритма и вероятность пропущенных в нем ошибок.

4.4. Асимптотическая сложность. Порядок сложности

Асимптотическая сложность алгоритмов представляет собой некоторую оценку времени и памяти, которые понадобятся программе в процессе ее исполнения, в зависимости от объема обрабатываемых данных.

Это понятие возникло как следствие ***Вычислительной сложности***, обозначающей функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных. Само понятие вычислительной сложности широко используется в Теории сложности вычислений, выходящей за пределы нашего курса.

Несмотря на то, что функция временной сложности алгоритма может быть определена точно, в большинстве случаев искать её точное значение не имеет смысла.

- Во-первых, значение временной сложности зависит от вида множества элементарных операций (сложения, умножения, битовых операций,...).
- Во-вторых, важно знать, как сложность алгоритма зависит от объёма входных данных. Например, массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Во внимание принимается порядок зависимости сложности алгоритма от объема входных данных.
- В-третьих, при увеличении размера входных данных вклад постоянных множителей и слагаемых низших порядков, фигурирующих в выражении точного времени работы, становится крайне незначительным, потому при определении искомой зависимости не учитывается.

Важна лишь **асимптотическая сложность**, то есть сложность алгоритма при стремлении размера входных (обрабатываемых) данных к бесконечности.

Порядок сложности алгоритма — это функция, доминирующая над точным выражением временной сложности.

Порядок сложности алгоритма обычно выражает его эффективность через количество обрабатываемых данных. Обозначается порядок сложности как **O(f(n))**.

Использование заглавной буквы O (так называемой Big O Notation) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма растёт в зависимости от объёма входных данных не быстрее, чем f(n), умноженная не некоторую константу.

Допустим, некоторому алгоритму нужно выполнить $4n^3 + 7n$ условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление $7n$. Тогда говорят, что временная сложность этого алгоритма равна $O(n^3)$, то есть зависит от размера входных данных кубически.

С помощью $O(f(n))$ математически описывается сложность алгоритма в условиях наихудшего сценария при большом количестве входных данных. Лучший способ сравнения эффективностей алгоритмов состоит в сопоставлении их порядков сложности. Этот метод применим как к временной, так и к пространственной (емкостной) сложности.

Различают следующие порядки сложности алгоритма.

O(1) - сложность порядка 1 (order 1)

Такое случается, когда время работы алгоритма не зависит от размера входных данных. Например, для определения значения второго элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных второй элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.

Сложность $O(1)$ означает, что алгоритм выполняется за постоянное/константное время (constant time).

O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Необходимо "пройти" по всем n элементам массива, чтобы понять, какой из них максимальный. Чем больше размер массива данных, тем большее количество элементов необходимо просмотреть для получения результата.

O(log n) — логарифмическая сложность

Простейший пример задачи с такой сложностью — бинарный поиск. Если массив отсортирован, можно отыскать в нём какое-то конкретное значение методом деления пополам. После проверки среднего элемента, если он больше искомого, отбрасывается вторая половина массива — там его точно нет. Если же меньше, то наоборот — отбрасывается первая половина. Затем процесс деления выбранной части массива повторятся до нахождения искомого элемента. Таким образом, в наихудшем случае будет проверено $\log n$ элементов.

O(n^2) — квадратичная сложность

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как $n * n$, то есть n^2.

Бывают и другие оценки сложности, но все они основаны на том же принципе.

Средний и наихудший случаи

Порядок сложности является верхней границей сложности алгоритмов. Если алгоритм имеет большой порядок сложности, это вовсе не означает, что программа будет выполняться действительно долго. На некоторых наборах данных выполнение алгоритма занимает намного меньше времени, чем можно предположить на основе их сложности.

Например, требуется найти заданный элемент в массиве A. Возможны следующие варианты развития событий.

- Если искомый элемент находится в конце списка, то программе придётся выполнить n шагов. В таком случае сложность алгоритма составит $O(n)$. В этом наихудшем случае время работы алгоритма будем максимальным.
- С другой стороны, искомый элемент может находится в списке на первой позиции. Алгоритму придётся сделать всего один шаг. Такой случай называется наилучшим и его сложность можно оценить, как $O(1)$.
- Оба эти случая маловероятны. Ожидаемая ситуация, когда элемента списка расположены в произвольном порядке, искомый элемент может находиться где-то в середине массива. В среднем потребуется сделать $n/2$ сравнений, чтобы найти требуемый элемент. Значит сложность этого алгоритма в среднем составляет $O(n/2) = O(n)$.

Порядок средней сложности (как в последнем примере) и сложности в наихудшем случае совпадали, но для многих алгоритмов наихудший случай сильно отличается от ожидаемого. Например, алгоритм быстрой сортировки в наихудшем случае имеет сложность порядка $O(n^2)$, в то время как ожидаемое (среднее) поведение описывается оценкой $O(n \cdot \log(n))$, что много быстрее.

Замечание. Порядок сложности может быть характеристикой и объёма занимаемой памяти в зависимости от размера входных данных. Некоторые алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Известны примеры, когда эффективные алгоритмы требуют таких больших объёмов машинной памяти (без возможности использования более медленных внешних средств хранения), что этот фактор сводит на нет преимущество «временной эффективности» алгоритма.

Таким образом, для оценки эффективности алгоритма важен не только порядок временной сложности, но и порядок емкостной сложности. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

4.5. Оценка сложности отдельных алгоритмов

В качестве примера определения порядка сложности рассматриваются следующие примеры. Для сокращения объема текста алгоритмы записываются на языке С.

Пример 1

```
for (int i = 0; i < n; ++i) {
     for (int j = 0; j < m; ++j) {
          a[i][j]++;
     }
}
```

Трудоемкость рассматриваемого фрагмента программы определяется:

- двумя операторами начального присвоения,
- n операциями инкремента управляющей переменной внешнего цикла,
- $n \cdot m$ операциями инкремента управляющей переменной внутреннего цикла,
- n операциями сравнения во внешнем цикле,
- $n \cdot m$ операциями сравнения во внутреннем цикле,
- $n \cdot m$ операциями нахождения элемента в массиве (вычислений относительного адреса элемента),
- $n \cdot m$ операциями инкремента.

Суммарная трудоемкость будет определена как сумма всех описанных трудоемкостей, то есть в первом приближении ее можно определить, как $4(n \cdot m) + 2n + 2$ условных операций.

Несмотря на такое значительное число операций сложность алгоритма определяется порядком: $O(n \cdot m)$.

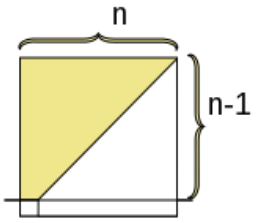
При получении порядка сложности рассматриваемого примера учитывалось, что асимптотические обозначения задают класс функций, то есть нет разницы между $O(4(n \cdot m) + 2n + 2)$ и $O(n \cdot m)$, поэтому при верхней оценке трудоемкости берется самая быстро растущая функция. В данном случае это $O(n \cdot m)$.

Пример 2

```
for (int i = 0; i < n-1; ++i) {
     for (int j = 0; j < n-i; ++j) {
          a[i][j]++;
     }
}
```

В отличие от Примера 1, во внутреннем (вложенном) цикле перебираются элементы не до `n`, а до `n-i`, при этом `i` не является константой, а зависит от `n`.

Сложность внутреннего цикла определяется порядком: $O(n-i)=O(n)$. Таким образом, сложность приведенного алгоритма можно оценить как $O(n^2)$.



Графическая интерпретация Примера 2

Из рисунка видно, что обработано будет примерно $n^2/2$ элементов матрицы. Из этого примера можно сделать вывод о том, что "игры" с начальным и конечным значением счетчика цикла обычно на трудоемкость не влияют.

Пример 3

```
for (int i = 0; i < n; ++i) {
     for (int j = 1; j < n; j *= 2) {
          a[i][j]++;
     }
}
```

В рассматриваемом примере управляющая переменная внутреннего цикла изменяется в два раза, то есть принимает значения: 1, 2, 4, 8,... 2^k. При этом $k=\log_2(n)$. Таким образом, трудоемкость внутреннего цикла будет оценена как $O(\log(n))$, следовательно, трудоемкость всего алгоритма будет равна

$$O(n\cdot\log(n)).$$

Пример 4

```
for (int i = 0; i < 100; ++i) {
     for (int j = 1; j < n; ++j) {
          ++count;
     }
}
```

Трудоемкость этой задачи определяется порядком $O(100n)=O(n)$.

В данном примере показано, что при асимптотическом анализе можно отбрасывать константы (даже очень большие).

В некоторых задачах `n` может быть ограничено, например: `n < 100000000`. И это не значит что его всегда можно игнорировать. Очевидно, что будь в рассматриваемом примере такое большое значение — его обязательно нужно считать за `n`.

Пример 5

```
for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n/i; ++j) {
          matrx[i][i]++;
     }
}
```

Во вложенном цикле рассматриваемого примера выполняется `n/i` итераций.

То есть в `i`-й строке матрицы `matrx` обрабатывается `n/i` элементов.

Трудоемкость обработки `i`-й строки оценивается порядком: $O\left(\sum_{j=0}^{n/i} 1\right) = O(n/i)$.

Суммарный порядок сложности всего алгоритма будет равен:

$$O\left(\sum_{i=0}^{n} \frac{n}{i}\right) = O\left(n \cdot \sum_{i=0}^{n} \frac{1}{i}\right) = O(n \cdot \log(n)).$$

В последней формуле использовано свойство близости в некотором приближении операций суммирования и интегрирования, а также: $\int \frac{dx}{x} = \ln x$.

Примеры 5 и 3 на самом деле сильно отличаются друг от друга. В примере 3 каждая итерация вложенного цикла выполняла одинаковое число итераций (их количество `log(n)`), а в примере 5 для каждой следующей строки матрицы обрабатывалось меньше столбцов, чем для предыдущей. Тем не менее, асимптотические оценки обоих примеров совпадают.

Пример 6

```
int Factorial(int n){
     int y=1;
     if (n > 1) y = n*Factorial(n-1);
     return y;
}
```

В данном примере функция рекурсивно вызывает саму себя. Сложность рекурсивных алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. В рассматриваемом примере функция выполняется `n` раз. Следовательно, вычислительная сложность этого алгоритма равна `O(n)`.

Пример 7

```
void DoubleRecursive(int n) {
     printf(" n = %d\n", n);
     if (n > 0) {
          DoubleRecursive(n-1);
          DoubleRecursive(n-1);
     }
}
```

Рекурсивный алгоритм, который вызывает себя несколько раз, называется многократной рекурсией. Такие функции гораздо сложнее анализировать, кроме того, они существенным образом усложняют алгоритм. Поскольку функция вызывается дважды, можно предположить, что её сложность будет равна `O(2n)=O(n)`. Но на самом деле ситуация гораздо сложнее. При внима-

тельном исследовании этого алгоритма становится очевидным, что его сложность равна $O\left(2^{n+1}-1\right)=O\left(2^{n}\right)$.

Замечание. Объёмная сложность рекурсивных алгоритмов

Для всех рекурсивных алгоритмов очень важно понятие объёмной сложности. При каждом вызове функция запрашивает небольшой объём памяти для сохранения в стеке значений отдельных переменных (параметров), но этот объём может значительно увеличиваться в процессе рекурсивных вызовов. По этой причине всегда необходимо проводить хотя бы поверхностный анализ объёмной сложности рекурсивных процедур.

Наиболее сложными частями алгоритма обычно является выполнение циклов и обращение к функции. Если одна функция вызывает другую, то необходимо более тщательно оценить сложность последней. Когда в ней выполняется определённое число инструкций (например, вывод на печать), то на оценку сложности это практически не влияет. Если же в вызываемой функции выполняется $O(n)$ элементарных действий (шагов), то эта функция может значительно усложнить алгоритм. Когда функция вызывается внутри цикла, то влияние может быть намного больше.

В качестве примера рассмотрим две функции: *ONE* со сложностью $O(n^3)$ и *TWO* со сложностью $O(n^2)$. Так, если во внутренних циклах функции *ONE* происходит вызов функции *TWO*, то сложности процедур перемножаются. Сложность всего алгоритма составляет $O\left(n^3\right)\cdot O\left(n^2\right)=O\left(n^5\right)$. В том случае, когда основная программа вызывает процедуры по очереди, то их сложности складываются: $O\left(n^3\right)+O\left(n^2\right)=O\left(n^3\right)$.

4.6. Шпаргалка по асимптотической сложности алгоритмов

В предыдущем пункте показано, как определить порядок сложности для различных алгоритмов, но... сколько времени люди пишут программы, разрабатывают алгоритмы, столько же времени и оценивают сложность полученных продуктов. Таким образом, можно "не изобретая велосипед", воспользоваться готовыми оценками сложности наиболее распространенных алгоритмов.

Ниже как раз и приведена эта полезная информация. Рассмотрены следующие задачи:

- поиск нужного элемента в разных структурах данных;
- сортировка элементов в массиве;
- оценка эффективности разных способов организации данных;
- задачи с динамически выделяемой памятью (списки, кучи) и графами.

В таблицах отдельные поля выкрашены в разные цвета, соответствующие:

Хорошо	Приемлемо	Плохо

Поиск

Алгоритм	Структура данных	Временная сложность		Сложность по памяти
		В среднем	В худшем	В худшем
Поиск в глубину (DFS)	Граф с |V| вершинами и |E| ребрами	-	O(|E| + |V|)	O(|V|)
Поиск в ширину (BFS)	Граф с |V| вершинами и |E| ребрами	-	O(|E| + |V|)	O(|V|)
Бинарный поиск	Отсортированный массив из n элементов	O(log(n))	O(log(n))	O(1)
Линейный поиск	Массив	O(n)	O(n)	O(1)
Кратчайшее расстояние по алгоритму Дейкстры используя двоичную кучу как очередь с приоритетом	Граф с |V| вершинами и |E| ребрами	O((|V| + |E|) log |V|)	O((|V| + |E|) log |V|)	O(|V|)
Кратчайшее расстояние по алгоритму Дейкстры используя массив как очередь с приоритетом	Граф с |V| вершинами и |E| ребрами	O(|V|^2)	O(|V|^2)	O(|V|)
Кратчайшее расстояние используя алгоритм Беллмана–Форда	Граф с |V| вершинами и |E| ребрами	O(|V||E|)	O(|V||E|)	O(|V|)

Сортировка

Алгоритм	Структура данных	Временная сложность			Вспомогательные данные
		Лучшее	В среднем	В худшем	В худшем
Быстрая сортировка	Массив	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Сортировка слиянием	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Пирамидальная сортировка	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Пузырьковая сортировка	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка вставками	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка выбором	Массив	O(n^2)	O(n^2)	O(n^2)	O(1)
Блочная сортировка	Массив	O(n+k)	O(n+k)	O(n^2)	O(nk)
Поразрядная сортировка	Массив	O(nk)	O(nk)	O(nk)	O(n+k)

Структуры данных

Структура данных	Временная сложность								Сложность по памяти
	В среднем				В худшем				В худшем
	Индексация	Поиск	Вставка	Удаление	Индексация	Поиск	Вставка	Удаление	
Обычный массив	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Динамический массив	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Односвязный список	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Двусвязный список	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Список с пропусками	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Хеш таблица	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Бинарное дерево поиска	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Декартово дерево	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
Б-дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Красно-черное дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Расширяющееся дерево	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
АВЛ-дерево	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Кучи

Куча	Временная сложность						
	Преобразование к куче	Поиск максимума	Извлечение максимума	Увеличить ключ	Вставить	Удалить	Слияние
Связный список (отсортированный)	-	O(1)	O(1)	O(n)	O(n)	O(1)	O(m+n)
Связный список (не отсортированный)	-	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)
Бинарная куча	O(n)	O(1)	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(m+n)
Биномиальная куча	-	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Фибоначчева куча	-	O(1)	O(log(n))	O(1)*	O(1)	O(log(n))*	O(1)

Представление графов

Пусть дан граф с |V| вершинами и |E| ребрами, тогда

Способ представления	Память	Добавление вершины	Добавление ребра	Удаление вершины	Удаление ребра	Проверка смежности вершин
Список смежности	O(|E| + |V|)	O(1)	O(1)	O(|E| + |V|)	O(|E|)	O(|V|)
Список инцидентности	O(|E| + |V|)	O(1)	O(1)	O(|E|)	O(|E|)	O(|E|)
Матрица смежности	O(|V|^2)	O(|V|^2)	O(1)	O(|V|^2)	O(1)	O(1)
Матрица инцидентности	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|V||E|)	O(|E|)

Представленная информация может быть использована разработчиками алгоритмов решения различных практических задач с целью создания высокоэффективных программ и программных комплексов.

5. РЕКОМЕНДАЦИИ ПО СОСТАВЛЕНИЮ АЛГОРИТМА

В данном разделе собраны вместе все рекомендации и замечания, что были сформулированы в различных частях настоящего пособия.

- Любой алгоритм определяется следующими атрибутами: именем, входными и выходными данными, началом, командами (действиями, операторами), концом алгоритма.
  Имя (наименование) алгоритма раскрывает его смысл и определяется решаемой задачей.
  Входные и выходные данные необходимо описать до создания алгоритма, на этапе осмысления задачи. Это как в математике: Вопросы "**Что дано**?", "**Что надо получить**?" определяют отправную точку и цель любого процесса. Не исключением является и процесс создания алгоритма.
  Входные и выходные данные, а также используемые переменные лучше всего **именовать в соответствии с их назначением** (обеспечить мнемоническую связь с обозначениями величин, используемых в алгоритме). Не рекомендуется в одном программном модуле одну и ту же переменную применять для обозначения разных переменных алгоритма.
  Все переменные с сопроводительными комментариями имеет смысл записать на отдельном листе бумаги.

- **Переменные** и **параметры**, используемые в алгоритме для реализации решаемой задачи, **должны быть объявлены** (описаны) в начале блок- схемы. Рекомендуется **отказаться от** использования **глобальных переменных**. Все данные, которые использует функция, должны поступать в нее как **входные параметры**. **Возвращаемые значения** также должны быть помещены в список параметров, либо в случае одного результата возвращены через имя функции.
  Все переменные перед первым их использованием должны быть **инициализированы** (им должны быть присвоены соответствующие значения).

- При разработке сложных, объемных алгоритмов правильнее **воспользоваться методом декомпозиции** (методом пошаговой детализации), когда решение одной большой задачи заменяется последовательным решением серии меньших задач, пусть и взаимосвязанных, но более простых, которые, в свою очередь, также могут быть расчленены на части. Каждая часть впоследствии может быть оформлена отдельной функцией. Пошаговая детализация выполняется до тех пор, пока **блок-схема** любого **алгоритма** не уместится **на одной странице**.

- При разработке алгоритмов следует использовать **принцип структурного программирования.** То есть алгоритм любой сложности может быть представлен комбинацией нескольких **типовых** (базовых) **структур**. Основные

структуры алгоритмов - это ограниченный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий. Характерной особенностью этих структур является наличие у них **одного входа** и **одного выхода**.

Возврат назад осуществляется только в циклах!!!!

Обычно при составлении схемы **блоки размещаются последовательно**, друг под другом, в порядке их выполнения**.** Это дает простую и наглядную структуру алгоритма, по которой легко далее составить программу.

Применение типовых структур позволяет сделать алгоритм понятным и читаемым, что уменьшает количество ошибок на следующем этапе разработки программы - кодировании.

- Типовые структуры ветвление, множественное ветвление, обход или множественный выбор должны быть построены таким образом, что все **ветви каждой структуры сходятся в одной точке**. Это важный признак грамотного применения указанных типовых структур.

  Если в каждой из ветвей имеется **одинаковый оператор**, то его рекомендуется (по возможности) вынести из этой структуры, разместив (по смыслу) до или после нее.

- При разработке циклических алгоритмов все действия по их проектированию выполняются в следующей последовательности:
  - Во-первых, сначала определяются повторяемые **операторы цикла** (тело цикла).
  - Во-вторых, записываются операторы, называемые **начальными операторами цикла** (стоят непосредственно перед циклом!!!), необходимые для правильности первого выполнения операторов цикла.
  - В-третьих, составляется **условие выхода из цикла** (оператор, определяющий, продолжать цикл, или необходимо его закончить).

  Если в алгоритме содержится два или более циклов, то визуально должно выполняться следующее: **циклы будут вложенными** друг в друга (внутренний и внешний), или **последовательно расположены**. Если циклы накладываются друг на друга, то однозначно имеет место ошибка!

  Для случая вложенных циклов у **каждого из них имеется полный комплект операторов**: собственные операторы выхода из цикла, собственные тела циклов, и, самое главное - собственные операторы начального присвоения.

  Типовую структуру **арифметический цикл следует применять** только тогда, когда известна **управляющая переменная цикла**, ее начальное и конечное значения.

  Оператор, **изменяющий значение управляющей переменной** цикла, **располагается последним** в теле цикла.

  В качестве начального и конечного значений управляющей переменной арифметического цикла **использовать переменные**, а не выражения; зна-

чения этих переменных необходимо вычислить непосредственно перед циклом.

Повторяющиеся **с одинаковыми данными вычисления** правильнее **вынести их цикла**, выполнив их до входа в цикл.

В циклах с неизвестным числом повторений при определенном сочетании входных параметров возможно "зацикливание" алгоритма! Для исключения его в алгоритм следует добавить оператор, контролирующий максимальное количество прохождений по циклу.

При использовании циклов **пока** и **до** **условия выхода** из циклов должны быть наиболее **простыми**. В случае сложных условий с целью их упрощения вводятся специальные переменные, называемыми **флагами**, а в тело цикла добавляются операторы, в которых вычисляются значения этих флагов. Тогда выход из цикла организуется как проверка значения этих флагов.

В любом цикле **выход** из него должен быть **только один**!!!

- Практически все разрабатываемые алгоритмы необходимо **проверить** с помощью соответствующих **трассировочных таблиц** при различных входных наборах данных. Особенное внимание следует уделять **операторам ветвления**. При верификации **циклических структур** обязательной проверке подлежит **первое вхождение в цикл** (правильность операторов начального присвоения) и **выход из цикла.**

- При реализации **рекурсивных алгоритмов** требуется, чтоб алгоритм являлся конечным. **Конечность рекурсивного алгоритма** обеспечивается с помощью специального (терминального, граничного) условия, реализуемого **оператором ЕСЛИ**.

  При выборе между восходящей и нисходящей рекурсией следует знать, что восходящая рекурсия более эффективна по быстродействию и в меньшей степени использует стек. Поэтому во всех случаях, когда это возможно, рекомендуется вместо нисходящей рекурсии строить восходящую.

  Если есть возможность решить задачу без использования рекурсии, то лучше этой возможностью воспользоваться.

- Алгоритм желательно сопровождать **комментариями**, позволяющими легко проследить за логической взаимосвязью отдельных его частей.

ЗАКЛЮЧЕНИЕ

Программирование - важнейший процесс создания последовательности команд, организующий работу любого компьютер. Алгоритмизация - важнейший этап программирования. Изучению принципам построения алгоритмов посвящено настоящее пособие.

В учебном пособии даны понятия алгоритма и алгоритмизации, рассмотрены типовые структуры алгоритмов, на конкретных примерах проанализированы особенности каждой из структур применительно к типам решаемых задач.

Первые разделы пособия посвящены изучению алгоритмов и способов их построения. Подробно рассмотрены свойства каждой типовой схемы алгоритма, различные подходы и основные приемы его разработки.

Особое внимание уделено рекомендациям по разработке различных алгоритмов.

Отдельно описаны способы и методики проверки правильности создаваемого алгоритма, его тестирования и отладки.

Рассмотрен анализ эффективности (сложности) алгоритмов с многочисленными примерами.

На основании изложенного можно констатировать:

"ГРАМОТНО НАПИСАННЫЙ АЛГОРИТМ -

- ЗАЛОГ КАЧЕСТВЕННОЙ ПРОГРАММЫ!"

СПИСОК ЛИТЕРАТУРЫ

1. Единая система программной документации/ Сборник стандартов, ГОСТ 19.001-77 ... ГОСТ 19.701-90.- М.: Издательство стандартов, 1994.- 156 с.
2. Н. Вирт "Алгоритмы и структуры данных. Новая версия для Оберона.- М.: ДМК Пресс, 2010. - 272 с.
3. Кнут Д. Э. Искусство программирования. Том 1. Основные алгоритмы = The Art of Computer Programming. Volume 1. Fundamental Algorithms / под ред. С. Г. Тригуб (гл. 1), Ю. Г. Гордиенко (гл. 2) и И. В. Красикова (разд. 2.5 и 2.6).- 3.- Москва: Вильямс, 2002.- Т.1.- 720 с.
4. Филиповский В.М. Основы программирования и алгоритмизации. Ч. 1. Технология создания программ: Учебное пособие.- СПб.: СПбПУ, 2018.- 38 с., ил.
5. Ундаров С.А., Филиповский В.М. Основы технологии программирования: Учеб. пособие.- СПб.: СПбГТУ, 1998.- 31 с., ил.

П. ПРИЛОЖЕНИЯ

П.1. Задания на ВЕТВЛЕНИЕ

Приведенные в условиях задач величины с индексами следует рассматривать как элементы массивов (одномерных или двумерных). Циклы при построении алгоритмов в данных заданиях не использовать.

1. Вычислить значение функции Z при одном значении X: $Z = Y^2 + X^2$, где

$$Y = \begin{cases} |X| + 2, & \text{если } X < 3; \\ X^2 - a_1, & \text{если } X = 3; \\ X - C, & \text{если } 3 < X < 10; \\ a_2^2, & \text{если } 10 \le X. \end{cases}$$

2. Вычислить значение функции Z по одной из формул:

$$Z = \begin{cases} X - b_1, & \text{если } X < 6; \\ X^2 + b_2, & \text{если } X = 15 \text{ или } 20; \\ b_3 + X & \text{в остальных случаях.} \end{cases}$$

3. Определить значение наибольшего элемента главной диагонали матрицы A(1:3, 1:3).

4. Вывести значения x_1, x_2, x_3 в порядке возрастания.

5. Вычислить значение Y по одной из заданных формул:

$$Y = \begin{cases} \ln\left|X + a^2\right|, & \text{если } X < 7; \\ \sqrt{|b_{11} + b_{21}|} + b_{22}, & \text{если } X = 10; \\ \dfrac{3.5b_{12}}{0.5|b_{11} \cdot b_{22}| + 2}, & \text{в остальных случаях.} \end{cases}$$

6. Определить значение наименьшего элемента седьмого столбца матрицы B(1:3, 1:7).

7. Вычислить значение Y по одной из заданных формул:

$$Y = \begin{cases} \sqrt[3]{\dfrac{\ln x + 5.5|a_1|}{1.5x}}, & \text{если } 15 < x < 21; \\ a_1 x^2 + a_2 x + a_3 & \text{в остальных случаях.} \end{cases}$$

8. Определить значение наибольшего элемента второй строки матрицы C(1:3, 1:3).

9. Вычислить значение Y по одной из заданных формул:

$$Y = \begin{cases} 1.5 - \sqrt{|c_1|} + e^{c_2}, & \text{если } x < 10; \\ x\left(\ln|c_1| 0.5c_3\right)^2, & \text{если } 15 < x \le 25; \\ \text{вывести : "значение } Y \text{ в остальных} \\ \text{случаях не определено."} \end{cases}$$

10. Вычислить значения функций w и v:

$$w = \begin{cases} \dfrac{c_1 + c_3 + \sqrt{|c_1|}}{4x - 3b}, & \textit{если знаменатель не равен нулю или } x < 5; \\ \ln|c_1 x + 2.5b|, & \textit{в остальных случаях} \end{cases}$$

$$v = w^2 + c_1 x$$

11. Определить значение наименьшего элемента главной диагонали матрицы B(1:3, 1:3).

12. Массив B(1:3) – целый. Какому элементу массива равна величина D? D = a + a^2 + a^3 (a – целая величина).

13. Вывести значения x_1, x_2, x_3, x_4 в порядке убывания.

14. Вычислить значение Y по одной из заданных формул:

$$Y = \begin{cases} \ln|X + a^2|, & \textit{если } X < 7; \\ \sqrt{|b_{11} + b_{22}|} + b_{33}, & \textit{если } X = 10; \\ \dfrac{5b_{13}}{|b_{11} \cdot b_{12}| + 4}, & \textit{в остальных случаях.} \end{cases}$$

15. Вычислить значение функции Z по одной из формул:

$$Z = \begin{cases} b_1, & \textit{если } X \le 0; \\ X^2 + b_2, & \textit{если } 0 < X \le 1; \\ b_3 + X^4 & \textit{в остальных случаях.} \end{cases}$$

16. Вычислить значение функции Z при одном значении X: Z = Y^2 + X^2, где

$$Y = \begin{cases} |X| + 2a_3, & \textit{если } X < 6; \\ X^2 - a_1, & \textit{если } X = 6; \\ X - C, & \textit{если } 6 < X < 20; \\ a_2^2, & \textit{если } 20 \le X \end{cases}$$

17. Вычислить значение Y по одной из заданных формул:

$$Y = \begin{cases} \ln|X + b_{21}{}^2|, & \textit{если } X < 5; \\ \dfrac{b_{12}}{|b_{11} \cdot b_{22}| + 2}, & \textit{в остальных случаях.} \end{cases}$$

18. Определить значение наименьшего элемента четвертого столбца матрицы B(1:3, 1:4).

19. Вычислить значение Y по одной из заданных формул:

$$Y = \begin{cases} \ln x + 9a_1, & \textit{если } x > 0 \\ -\dfrac{xa_2}{x^2 a_3 - 7}, & \textit{в остальных случаях} \end{cases}$$

20. Вычислить значение Y по одной из заданных формул:

$$Y=\begin{cases} x^2+3x+9a_1, & \textit{если } x\le 3; \\ -\dfrac{a_2\sin x}{x^2a_3-9}, & \textit{в остальных случаях}; \end{cases}$$

21. Вычислить значение Y по одной из заданных формул:

$$Y=\begin{cases} \ln x+9a_1, & \textit{если } x>0 \\ -\dfrac{xa_2}{x^2a_3-7}, & \textit{в остальных случаях} \end{cases}$$

22. Вычислить значение Y по одной из заданных формул:

$$Y=\begin{cases} 9a_1-x, & \textit{если } x>1.1 \\ -\dfrac{\sin 3xa_2}{x^4a_3+1}, & \textit{если } x<1.1 \end{cases}$$

23. Определить значение наименьшего элемента третьей строки матрицы B(1:3, 1:4).

24. Вычислить значения функций w и v:

$$w=\begin{cases} \dfrac{a_1+a_3+|a_2|}{4x-3b}, & \textit{если знаменатель не равен нулю или } x<15; \\ \ln|a_1x+2|, & \textit{в остальных случаях} \end{cases}$$

$$v=w+(a_1+a_2+a_3)\cdot x$$

25. Вычислить значение Y по одной из заданных формул:

$$Y=\begin{cases} 5-\sqrt{|c_1|}, \textit{если } x<10; \\ x\left(\ln|c_2|0.5c_3\right)^2, \textit{если } 25<x\le 35; \\ \textit{вывести}: \text{"}\textit{значение Y в остальных случаях не определено.}\text{"} \end{cases}$$

26. Вычислить значение Y по одной из заданных формул:

$$Y=\begin{cases} 1.2x^2-3x-9b_1, & \textit{если } x>3 \\ \dfrac{12.1b_2}{2x^2+b_3+1}, & \textit{если } x\le 3 \end{cases}$$

27. Вычислить значение Y по одной из заданных формул:

$$Y=\begin{cases} x^2+4x+5a_1, & \textit{если } x\le 2 \\ \dfrac{a_2}{x^2+4x+5a_3}, & \textit{если } x>2 \end{cases}$$

28. Вычислить значение Y по одной из заданных формул:

$$Y=\begin{cases} -3x+9b_1, & \textit{если } x>3 \\ \dfrac{x^3+b_2}{x^2+b_3+8}, & \textit{если } x\le 3 \end{cases}$$

П.2. Простые задания на циклические алгоритмы

Исходные данные должны включать и положительные числа, и отрицательные, и нули.

1. В массиве из 20 целых чисел найти наименьший элемент и поменять местами с первым элементом.
2. В массиве из 10 целых чисел найти наименьший элемент и поменять его местами с последним элементом.
3. В массиве из 15 вещественных чисел найти наибольший элемент и поменять его местами с последним элементом,
4. В массиве из 25 вещественных чисел найти наименьший элемент и поменять его местами с первым элементом.
5. Упорядочить по неубыванию массив, содержащий 20 целых чисел.
6. Упорядочить по невозрастанию массив, содержащий 25 вещественных чисел.
7. Упорядочить по неубыванию массив, содержащий 15 вещественных чисел.
8. Упорядочить по невозрастанию массив, содержащий 25 вещественных чисел.
9. Вычислить сумму и количество элементов массива X(100) для 0<=X<=1
10. Вычислить среднее арифметическое значение элемента положительного массива A(80).
11. Переписать элементы массива X(70) в массив Y и подсчитать их количество, (-1 <=X<=1).
12. Определить максимальный элемент массива B(50) и его порядковый номер.
13. Вычислить минимальный элемент массива C(40) и его порядковый номер.
14. Найти максимальный и минимальный элементы массива D(80) и поменять их местами,
15. Вычислить среднее геометрическое элементов массива Y(20).
16. Расположить в массиве R сначала, положительные, а затем отрицательные элементы массива Z (30).
17. В массиве N(50) определить сумму и количество элементов массива, кратным трём.
18. Вычислить сумму и количество элементов массива X(N).
19. Найти среднее геометрическое элементов массива A, где a>0, N<=40.
20. Переписать в массив Y подряд положительные элементы массива X(N), где X>0, N<=40.
21. Определить максимальный элемент массива B(k) и его порядковый номер, где X<0, K<=40.
22. Определить минимальный элемент массива C(k) и его порядковый номер, где -1 <=X<=1, K<=20.
23. В массиве F(45) подчитать количество положительных и отрицательных элементов.
24. В массиве Y(30) найти максимальный элемент и заменить его на 0.

25. В массиве A(20) найти последний отрицательный элемент и вывести его номер.
26. В массиве C(15) найти последний чётный элемент и поставить его на место первого элемента.
27. В массиве B(25) на место всех положительных элементов записать нули, а все отрицательные элементы сложить. На печать вывести новый массив и сумму отрицательных элементов.
28. В массиве A(10) найти минимальный и максимальный элементы, соответственно поменять с первым и последним элементами массива.
29. Из массива X(20) выбрать все чётные элементы и переписать их в массив A.
30. Из массива Y(l5) выбрать все нечётные элементы и переписать их в тот же массив.

П.3. Варианты заданий с матрицами

Рекомендации по решению задач обработки массивов (матриц)

- При решении задач, оперирующих с матрицами большого или переменного размера, рекомендуется отладить алгоритм на небольших матрицах конкретного размера с последующим обобщением его на случай матриц заданного размера.
- Специфика рассматриваемых задач: в них в качестве подзадач выступают обычно операции по обработке всей матрицы в целом, части матрицы, столбца, строки и т.д. Набор этих операций невелик. Из этого набора и следует выбирать наиболее крупные операции на каждом шаге решения задачи.
- Для решения матричной задачи часто главное – определить последовательность вложенных подзадач по убывающей сложности. Иногда полезно бывает начинать с выбора самой мелкой операции и искать операции возрастающей сложности.

1. Матрицу A(1:n,1:m) умножить на k и найти максимальный элемент 2-го столбца.
2. Найти сумму матриц A(1:n,1:m) и B(1:n,1:m). Умножить каждый элемент 1-й строки матрицы A на соответствующий элемент последней строки матрицы B.
3. Найти сумму элементов каждой строки матрицы A(1:n,1:m) и максимальный элемент 2-й строки.
4. Определить наибольший элемент в каждом столбце матрицы A(1:m, 1:k). Вычислить сумму элементов 2-го столбца.
5. Записать элементы матрицы A(1:n,1:m) в виде массива B(1:n*m). Найти минимальный элемент 2-ой строки.
6. В матрице A(1:n,1:m) сместить каждую строку, начиная со второй на одну вверх. Первую строку поставить на место последней. Найти разность максимального элемента 6-ой строки и минимального элемента 2-ого столбца.

7. Переставить строки матрицы A(1:n,1:m): первую с последней, вторую с предпоследней и т.д. Умножить третью строку на сумму элементов 5-го столбца.
8. Элементы матрицы A(1:n,1:m), большие 5 и меньшие 20 записать в массив B(1:n*m). Вычислить разность соответствующих элементов первой и последней строки.
9. Образовать матрицу A(1:n,1:n), все элементы главной диагонали которой равны единице. Угловые элементы матрицы принять равными m, все остальные элементы – равными 10.
10. Элементы массива B(1:к) расположить в порядке возрастания их значений. Вычислить произведение максимального элемента массива на сумму всех элементов массива.
11. Определить имеются ли равные элементы в массиве B(1:к), и если имеются то заданный массив умножить на A.
12. Определить номера строк матрицы A(1:n,1:m), совпадающих с массивом B(1:к). Последний столбец матрицы умножить на P.
13. В матрице A(1:n,1:m) определить наибольший элемент и умножить его на сумму элементов главной диагонали.
14. Определить наименьший элемент каждой четной строки матрицы A(1:n,1:m) (n - четное) и сумму этих элементов.
15. Переставить значения каждой пары элементов матрицы A(1:n,1::n), симметричных относительно главной диагонали. Найти максимальный элемент первого столбца.
16. Матрицу A(1:n,1:m) умножить на k и найти максимальный элемент 2-го строки.
17. Найти сумму матриц A(1:n,1:m) и B(1:n,1:m). Умножить каждый элемент 1-го столбца матрицы A на соответствующий элемент последнего столбца матрицы B.
18. Найти произведение элементов каждой строки матрицы A(1:n,1:m) и максимальный элемент 2-го столбца.
19. Определить наименьший элемент в каждом столбце матрица A(1:m, 1:k). Вычислить сумму элементов 2-ой строки.
20. Записать элементы матрицы A(1:n,1:m) в виде массива B(1:n*m). Найти максимальный элемент 2-ой строки.
21. В матрице A(1:n,1:m) сместить каждую строку, начиная со второй на одну вверх. Первую строку поставить на место последней. Найти разность максимального элемента 1-ой строки и минимального элемента последнего столбца.
22. Переставить строки матрицы A(1:n,1:m): первую с последней, вторую с предпоследней и т.д. Умножить первую строку на сумму элементов последнего столбца.

23. Элементы матрицы A(1:n,1:m), большие 5 и меньшие 20 записать в массив B(1:n*m). Вычислить разность соответствующих элементов первого и последнего столбцов.

24. Образовать матрицу A(1:n,1:n), все элементы главной диагонали которой равны нулю. Угловые элементы матрицы принять равными 5, все остальные элементы – равными 9.

25. Элементы массива B(1:к) расположить в порядке убывания их значений. Вычислить произведение минимального элемента массива на сумму всех элементов массива.

26. Определить имеются ли равные элементы в массиве B(1:к), и если имеются то заданный массив умножить на 25.

27. Определить номера строк матрицы A(1:n,1:m), совпадающих с массивом B(1:к). Последнюю строку матрицы умножить на P.

28. В матрице A(1:n,1:m) определить наименьший элемент и умножить его на сумму элементов главной диагонали.

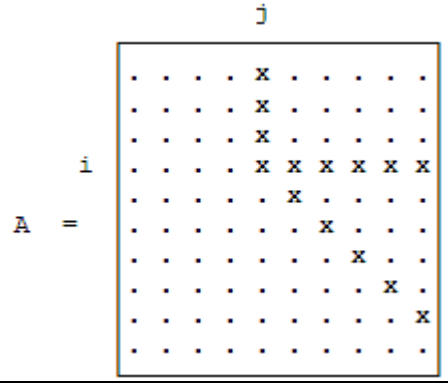
29. Определить наибольший элемент каждой четной строки матрицы A(1:n,1:m) (n - четное) и сумму этих элементов.

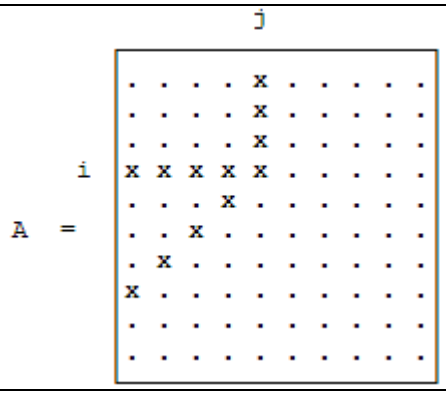
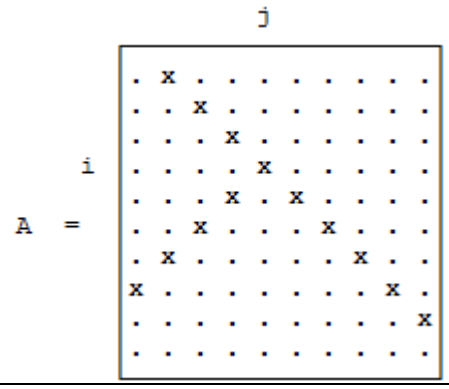
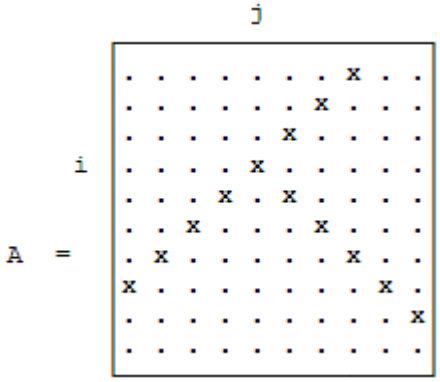
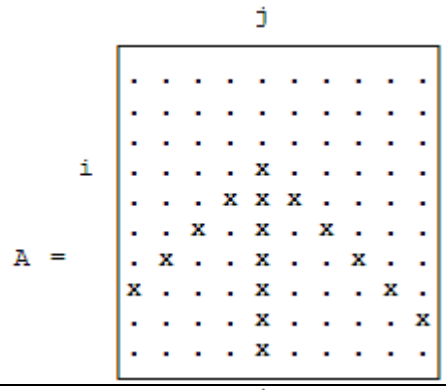
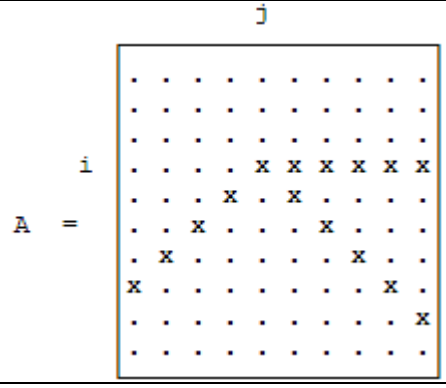
30. Переставить значения каждой пары элементов матрицы A(1:n,1::n), симметричных относительно главной диагонали. Найти максимальный элемент первой строки.

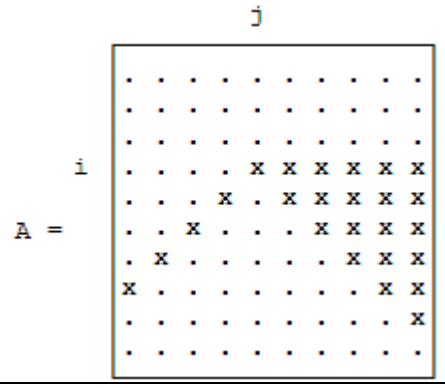
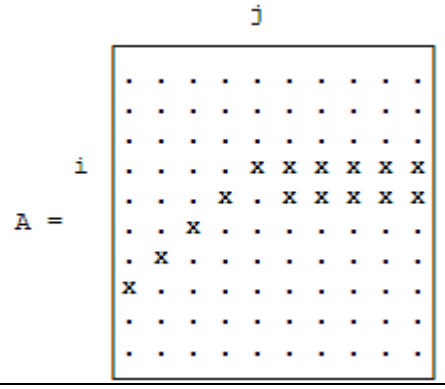
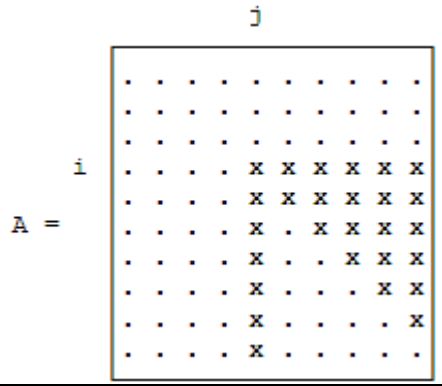
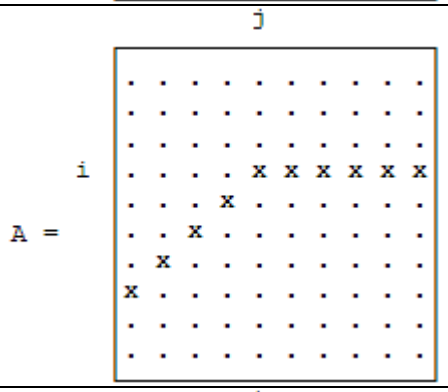
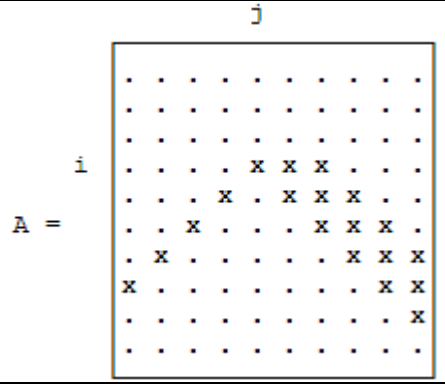
П.4. Задачи на преобразование матриц

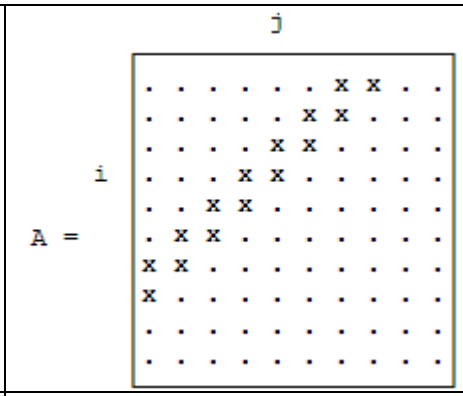
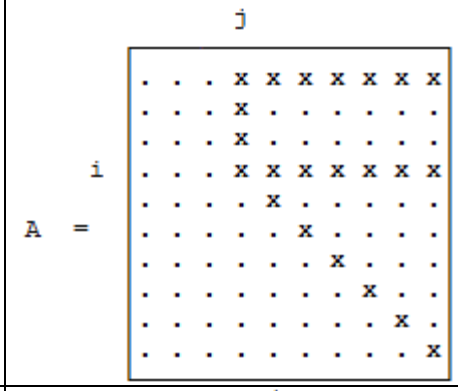
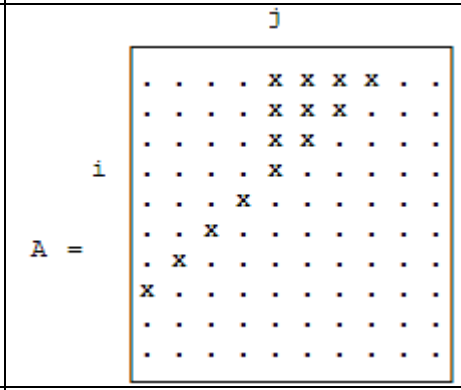
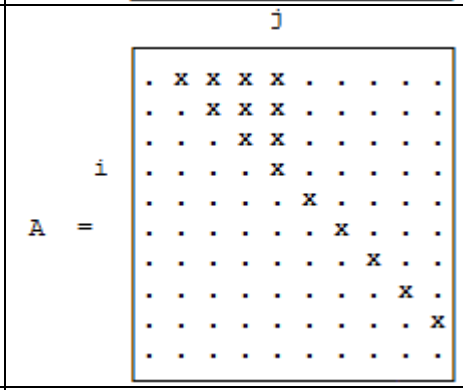
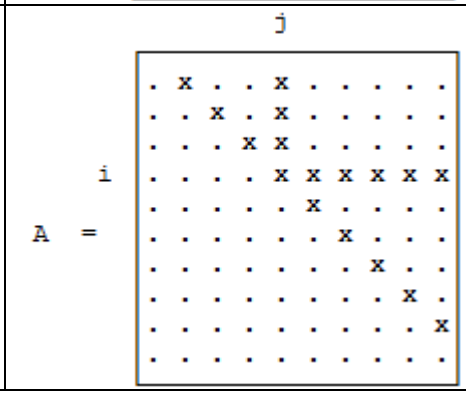
Требуется преобразовать квадратную матрицу A размерности m по правилу, написанному в задании. Под преобразованием понимается замена всех элементов исходной матрицы. В задании определено правило преобразования элемента матрицы, расположенного в i-й строке и j-м столбце. Преобразовать требуется все элементы матрицы! Вспомогательными массивами и рекурсиями не пользоваться. Разработанный алгоритм закодировать как функцию.

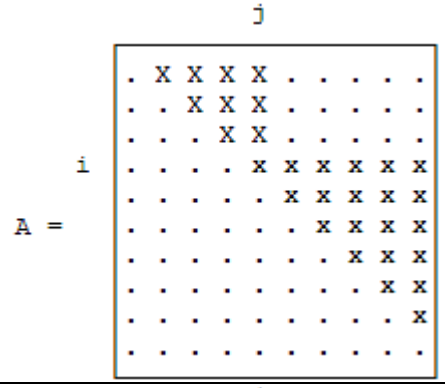
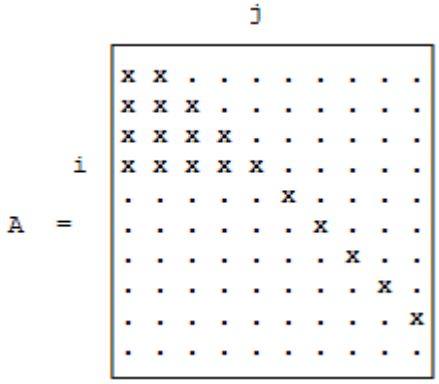
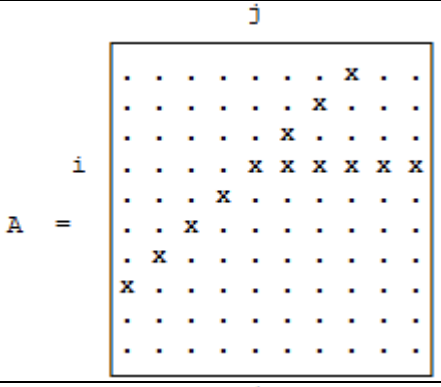
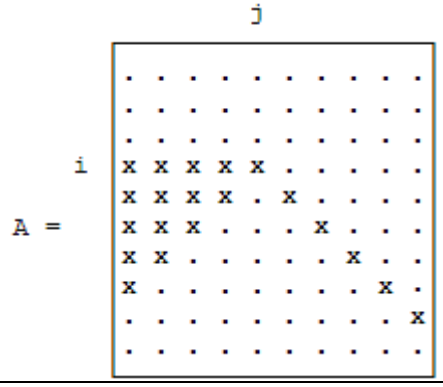
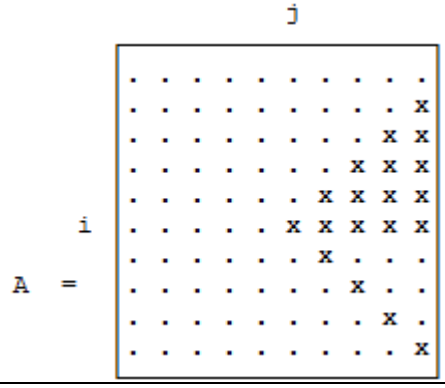
Главная сложность преобразования заключается в определении такого порядка (очередности) обновления, при котором ранее измененные элементы матрицы не учитывались бы при обновлении следующих элементов.

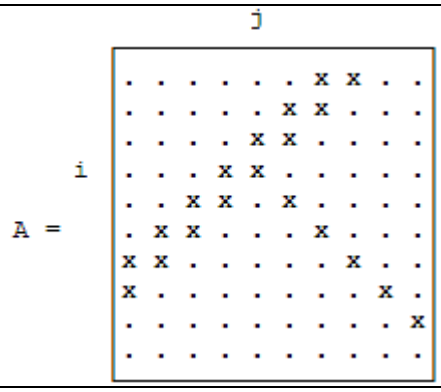
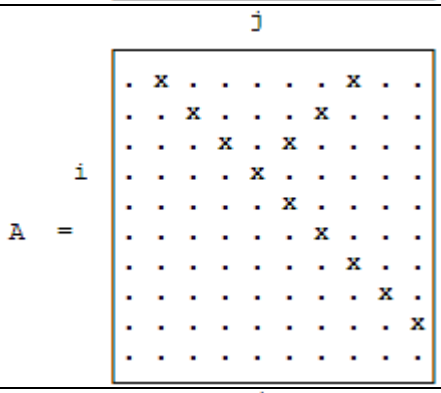
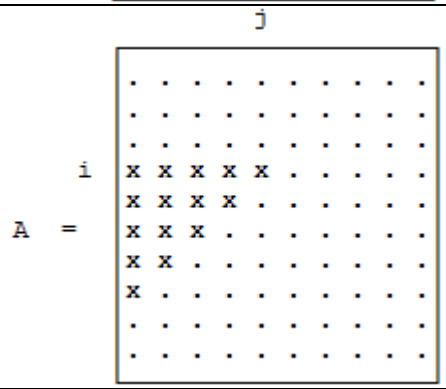
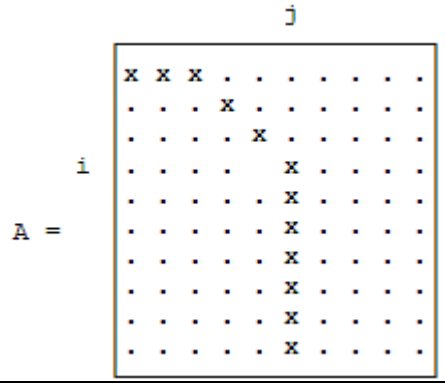
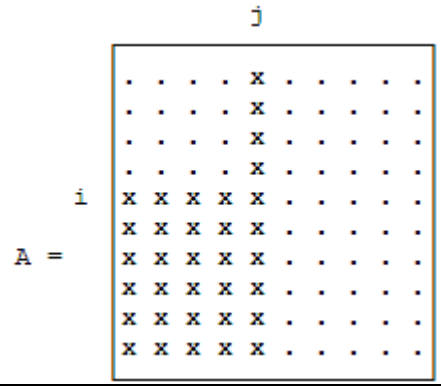
1	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 11, B(i,j) = i - j.	

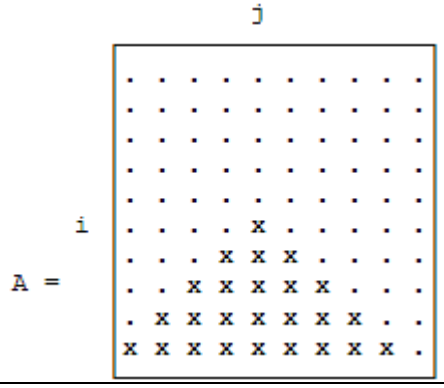
2	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 10, B(i,j) = Abs(2i-3j).	
3	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = Abs(i-2j).	
4	Элемент A(i,j) обновленной матрицы есть разность минимального и максимального из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 11, B(i,j) = i-2*j+i*j.	
5	Элемент A(i,j) обновленной матрицы есть сумма минимального и максимального из элементов исходной матрицы, помеченных символами "X"(смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = Abs(2*i-j-i*j).	
6	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 12, B(i,j) = 2*i-3*j.	

7	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 10, B(i,j) = 2*i-2*j.	
8	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 10, B(i,j) = i+j.	
9	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 12, B(i,j) = 3*i-j.	
10	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = i-j.	
11	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = i-6*j+i*i.	

12	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = 5i+4j.	
13	Элемент A(i,j) обновленной матрицы есть МАКСИМАЛЬНЫЙ ИЗ элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 11, B(i,j) = i-j.	
14	Элемент A(i,j) обновленной матрицы есть РАЗНОСТЬ минимального и максимального из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 11, B(i,j) = i-2*j.	
15	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = 2*i-j.	
16	Элемент A(i,j) обновленной матрицы есть МИНИМАЛЬНЫЙ из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = 2*i+j.	

17	Элемент A(i,j) обновленной матрицы есть ПОЛУСУММАСУММА минимального и максимального элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 12, B(i,j) = i+2j.	
18	Элемент A(i,j) обновленной матрицы есть МИНИМАЛЬНЫЙ из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = i+4j.	
19	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 12, B(i,j) = 2i-3j.	
20	Элемент A(i,j) обновленной матрицы есть МИНИМАЛЬНЫЙ из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = i-j+i*i.	
21	Элемент A(i,j) обновленной матрицы есть МИНИМАЛЬНЫЙ из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 8, B(i,j) = 2i-j+i*i.	

22	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 10, B(i,j) = 2i-j.	
23	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 10, B(i,j) = i+4j.	
24	Элемент A(i,j) обновленной матрицы есть МАКСИМАЛЬНЫЙ из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 10, B(i,j) = i+j.	
25	Элемент A(i,j) обновленной матрицы есть МАКСИМАЛЬНЫЙ из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 8, B(i,j) = 2*i-3*j.	
26	Элемент A(i,j) обновленной матрицы есть СУММА элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 7, B(i,j) = Abs(3i-j).	

27	Элемент A(i,j) обновленной матрицы есть ПОЛУСУММА минимального и максимального из элементов исходной матрицы, помеченных символами "X" (смотри рисунок). Полученную функцию использовать для преобразования квадратной матрицы B размерности 9, B(i,j) = Abs(i-2*j).	

П.4. Вычисление суммы ряда

Прежде, чем приступить к разработке алгоритма вычисления суммы ряда, следует определить рекуррентную формулу, по которой очередной член ряда может быть получен из предыдущего.

Если удобно, то рекуррентные формулы могут быть получены отдельно для числителя и знаменателя очередного члена ряда. В этом случае с целью предотвращения переполнения разрядной сетки необходимо предусмотреть в алгоритме периодическое деление числителя и знаменателя данной дроби на некоторое одинаковое число.

При вычислении суммы ряда с заданной точностью при определенном сочетании входных параметров возможно "зацикливание" алгоритма. Для исключения данной проблемы в алгоритм следует добавить оператор, контролирующий максимальное количество прохождений по циклу.

1	$y=\sum_{n=0}^{\infty}\frac{(x+3)^n}{n!}$, $-\infty<x<\infty$.	14	$y=\sum_{n=1}^{\infty}\frac{(-1)^{n-1}2^{2n-1}x^{2n}}{(2n)!}$, $-1\le x\le 1$.
2	$y=\sum_{n=0}^{\infty}nx^n$, $-1<x<1$.	15	$y=\sum_{n=1}^{\infty}\frac{(-1)^{n-1}x^n}{n}$, $-1<x<1$.
3	$y=\sum_{n=1}^{\infty}\frac{x^n}{n}$, $-1<x<1$.	16	$y=\sum_{n=1}^{\infty}\frac{a(a-1)\ldots(a-n+1)x^n}{n!}$, $-\infty<x<\infty$.
4	$y=\sum_{n=0}^{\infty}\frac{x^n}{n+1}$, $-1<x<1$.	17	$y=\sum_{n=0}^{\infty}(-1)^n x^{2n}$, $-1<x<1$.
5	$y=\sum_{n=0}^{\infty}\frac{(-1)^n(x-2)^n}{n^2}$, $1\le x\le 3$.	18	$y=\sum_{n=0}^{\infty}\frac{(x-1)^{2n+1}}{(2n+1)(x+1)^{2n+1}}$, $0<x\le 1$.
6	$y=\sum_{n=1}^{\infty}\frac{(-1)^{n-1}x^n}{n}$, $-1<x\le 1$.	10	$y=\sum_{n=1}^{\infty}\frac{(x-1)^n}{nx^n}$, $0.5\le x$.
7	$y=\sum_{n=0}^{\infty}\frac{5^n x^n}{(n+1)^2}$, $-0.2\le x\le 0.2$.	20	$y=\sum_{n=1}^{\infty}\frac{(-1)^{n-1}x^n}{n\cdot 2^n}$, $-2<x\le 2$.

8	$y=\sum_{n=0}^{\infty}\frac{x^n}{2^n}$, $-2<x<2$.	21	$y=\sum_{n=1}^{\infty}\frac{(-1)^{n+1}(x-3)^{n-1}}{3^n}$, $3<x\le 4$.
9	$y=\sum_{n=0}^{\infty}\frac{x^n}{n!}$, $-\infty<x<\infty$.	22	$y=\sum_{n=1}^{\infty}\frac{(-1)^n x^n(\ln x)^n}{n!}$, $0<x\le 1$.
10	$y=\sum_{n=1}^{\infty}\frac{x^{2n}}{4^n}$, $-2<x<2$.	23	$y=\sum_{n=0}^{\infty}\frac{(-1)^{n+1}}{(2n+1)x^{2n+1}}$, $1<\lvert x\rvert$.
11	$y=\sum_{n=0}^{\infty}x^n$, $-1<x<1$.	24	$y=\sum_{n=2}^{\infty}\frac{(-1)^{n-1}2^{n-1}x^n}{(n-1)!}$, $-1\le x\le 1$.
12	$y=\sum_{n=1}^{\infty}\frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!}$, $-\infty<x<\infty$.	25	$y=\sum_{n=1}^{\infty}\frac{(-1)^{n+1}x^{2n-1}}{2^{2n-1}(2n-1)!}$, $0\le x\le 2$.
13	$y=\sum_{n=0}^{\infty}\frac{(-1)^n x^{2n}}{(2n)!}$, $-\infty<x<\infty$.	26	$y=\sum_{n=1}^{\infty}\frac{(-1)^n 2^{2n}x^{2n}}{(2n)!}$, $-1\le x\le 1$.