Дисциплина: Теория Алгоритмов 4 курс 7 семестр
Лекционный курс

Вопросы к зачету

1. Основная задача теории алгоритмов. Методы исследования алгоритмов.
2. Понятие алгоритма. Принцип потенциальной осуществимости. Основные свойства алгоритмов. Понятие исполнителя алгоритмов.
3. Классификация алгоритмов. Блок-схемы описания алгоритмов. Формы записи алгоритмов.
4. Сложность алгоритмов. Варианты оценки сложности. Асимптотическая сложность алгоритма.
5. Реально выполнимые алгоритмы. Совпадение классов полиномиальных и реально выполнимых алгоритмов.
6. Полиномиальные и не полиномиальные алгоритмы. Примеры полиномиальных алгоритмов.
7. Примеры задач НП. Задача коммивояжера. Замкнутость класса задач НП.
8. Алгоритмизация и программирование.
9. Методы и средства программирования. Классификация языков программирования.
10. Методы построения эффективных алгоритмов.
11. Формальные языки и их построение.
12. Классификация формальных языков по Хомскому.
13. Машина Тьюринга. Работа Машины Тьюринга.
14. Машина Тьюринга. Программа Машины Тьюринга.
15. Машина Тьюринга. Программирование задач. Примеры.
16. Машина Поста. Особенности машины Поста.
17. Алгорифмы Маркова. Принцип нормализации. Программирование задач. Примеры.
18. Нотации Бекуса-Наура. Построение нотаций. Примеры.
19. Понятие вычислимой и рекурсивной функции. Базовые рекурсивные функции. Общерекурсивные функции.
20. Тезисы Черча и Клини. Частично-рекурсивные функции. Операция минимизации.
21. Основная задача теории алгоритмов. Понятие неразрешимой задачи. Экстраалгоритм.
22. Грамматика формальных языков.

Лекция №1

Тема №1: Введение в теорию алгоритмов

Возникновение математической теории алгоритмов. Парадоксы теории множеств. Основная проблема теории алгоритмов. Массовые проблемы. Экстраалгоритм и неразрешимые проблемы. Самоприменимость. Теорема Геделя. Разрешимость аксиоматических теорий.

1. Введение

Дисциплина «Теория алгоритмов» относится к вариативной части Блока 1. Дисциплины (модули) учебного плана. Она изучается после дисциплин «Дискретная математика», «Математическая логика», «Программирование». Для ее освоения студенты также используют знания, умения, навыки, сформированные в ходе изучения основных математических курсов: «Математический анализ» «Алгебра», «Геометрия».

Освоение данной дисциплины является основой для последующего изучения учебных дисциплин: «Информационные системы», «Практикум по решению задач на ЭВМ», «Основы искусственного интеллекта», «Компьютерное моделирование», прохождения педагогической практики, а также курсов по выбору студентов, содержание которых связано с готовностью студента углубить свои знания в области современной теоретической информатики.

Основная цель курса – формирование систематических знаний в области теории алгоритмов.

Целью освоения дисциплины «Теория алгоритмов» является:

– формирование систематических знаний о современных методах информатики, её месте и роли в системе наук;

– расширение и углубление понятий теоретической информатики, теории кодирования, алгоритмизации и программирования;

– развитие абстрактного мышления, пространственных представлений, вычислительной, алгоритмической культур и общей математической и информационной культуры.

Задачи дисциплины:

1. Формирование у студентов представления об общих проблемах и задачах теории алгоритмов: алгоритмах и представлении; методах описания и оценки алгоритмов, проблемы существования алгоритмов; проблемам анализа алгоритмов.
2. Выработка навыков решения задач, связанных с построением алгоритмов для машины Тьюринга, Поста, алгорифмов Маркова, с описанием алгоритмов с помощью грамматик языков и продукционных правил (нотаций Бекуса).
3. Подготовка студентов к восприятию других дисциплин, связанных с алгоритмами.
4. Стимулирование самостоятельной работы по освоению содержания дисциплины и формированию необходимых компетенций.

В результате изучения курса "Теория алгоритмов" необходимо знать:

- Основные теоретические основания теории алгоритмов (алгоритм, исполнитель алгоритма, виды алгоритмов, описание алгоритмов, описание формальных языков и грамматик).
- Формальные методы представления алгоритмов (программы Машины Тьюринга, алгорифмы Маркова, рекурсивные функции)
- Основы теории алгоритмов; понятия сложности алгоритма, асимптотической сложности алгоритма, реально выполнимых алгоритмов; основы теории полиномиальных алгоритмов.
- Основные методы разработки эффективных алгоритмов.
- Теорию построения и классификации формальных языков.

В результате изучения курса «Теория алгоритмов» необходимо уметь:

- Использовать знания по теории алгоритмов в профессиональной деятельности;
- Оценивать сложность алгоритмов решения задач, строить эффективные алгоритмы;
- Использовать методы теории алгоритмов для построения программ машины Тьюринга, алгорифмов Маркова, нотаций Бекуса.

Для изучения дисциплины «Теория алгоритмов» предполагается в течение 5 семестра 6 курса обучения проведение следующего количества занятий:

Семестр	Лекции	Практические занятия
5	13	13

Выделены для изучения следующие темы:

- Введение в теорию алгоритмов
- Основы теории формальных методов представления алгоритмов
- Основы теории формальных языков и грамматик

Курс «Теория алгоритмов» содержит лекционные и практические занятия.

На лекционные занятия выносятся общетеоретические темы.

На практических занятиях (которые, как правило, построены по типу семинарских занятий) разбираются методы решения задач, связанных с различными темами данного курса.

2. Используемая литература

При изучении данного курса необходимо привлечение дополнительного материала и проработка лекционного материала с помощью следующей учебной литературы:

1. Математическая логика и теория алгоритмов / сост. А.Н. Макоха, А.В. Шапошников, В.В. Бережной ; Министерство образования РФ и др. – Ставрополь : СКФУ, 2017. – 418 с. – Режим доступа: – URL: http://biblioclub.ru/index.php?page=book&id=467015 – Текст : электронный.

2. Кожухов, С.Ф. Сборник задач по дискретной математике : учебное пособие / С.Ф. Кожухов, П.И. Совертков. — 2-е изд., стер. — Санкт-Петербург : Лань, 2018. — 324 с. — ISBN 978-5-8114-2588-4. — Текст : электронный // Электронно-библиотечная система «Лань» : [сайт]. — URL: https://e.lanbook.com/book/102606.

3. Егоров, Д.Л. Теория вычислительных процессов и структур / Д.Л. Егоров ; Министерство образования и науки РФ, Казанский национальный исследовательский технологический университет. – Казань : КНИТУ, 2018. – 92 с. : схем., табл., ил. – Режим доступа: – URL: http://biblioclub.ru/index.php?page=book&id=500683 – Текст : электронный.

4. Алымова, Е.В. Конечные автоматы и формальные языки / Е.В. Алымова, В.М. Деундяк, А.М. Пеленицын ; Министерство науки и высшего образования РФ, ФГАОУ ВО «Южный федеральный университет». – Ростов-на-Дону; Таганрог : Издательство ЮФУ, 2018. – 292 с.: ил. – Режим доступа: – URL: http://biblioclub.ru/index.php?page=book&id=499456 – Текст : электронный.

5. Перемитина, Т.О. Математическая логика и теория алгоритмов / Т.О. Перемитина ; Министерство образования и науки Российской Федерации, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). – Томск : ТУСУР, 2016. – 132 с. : ил. – Режим доступа: – URL: http://biblioclub.ru/index.php?page=book&id=480886 – Текст : электронный.

6. Теория алгоритмов / сост. А.А. Брыкалова ; Министерство образования и науки РФ, Федеральное государственное автономное образовательное учреждение высшего образования «Северо-Кавказский федеральный университет». – Ставрополь : СКФУ, 2016. – 129 с. : ил. – Режим доступа: по подписке. – URL: http://biblioclub.ru/index.php?page=book&id=467402. – Библиогр. в кн. – Текст : электронный.

3 Рейтинговая система оценки текущей успеваемости студентов

Распределение рейтинговых баллов по видам оцениваемых работ представлено в следующей таблице.

№	Наименование разделов	Виды оцениваемых работ	Максимальное кол-во баллов
1	**Основы алгоритмизации и теории алгоритмов**	Домашняя практическая работа Письменная проверочная работа Активная работа на занятиях	8 12 2
2	**Методы представления алгоритмов**	Домашняя практическая работа Письменная проверочная работа Активная работа на занятиях	8 12 2
3	**Основы теории формальных языков и грамматик**	Домашняя практическая работа Письменная проверочная работа Активная работа на занятиях	5 10 1
4	Текущая аттестация по всем разделам	Компьютерное тестирование	40
ВСЕГО			100

1.Основная задача теории алгоритмов. Методы исследования алгоритмов.

Во всех сферах своей деятельности, и в частности, в сфере обработки информации, человек сталкивается с различными способами или методиками решения задач. Они определяют порядок выполнения действий для получения желаемого результата — мы можем трактовать это

как первоначальное или интуитивное определение алгоритма. Теория алгоритмов — раздел математики, изучающий общие свойства алгоритмов. Эта теория тесно связана с математической логикой, поскольку на понятие алгоритма опирается одно из центральных понятий математической логики — понятие *исчисления*. По существу вся математика связана с теми или иными алгоритмами. Первоначально алгоритмические проблемы и исследовались в древнем мире в рамках логики (еще не математической) как проблема поиска общего решения проблемы или доказательства утверждений. В средние века арабские ученые дали первое определение понятия «алгорифм» близкое к нашему понятию алгоритм. Отсюда и пошло английское слово «алгоритм», которое было заимствовано и в русском языке. С появление в 19 веке понятия программирования, математики стали интересоваться алгоритмами более предметно и в конце 19 века было положено начало исследованиям по общей теории алгоритмов. В это время математика переживала некий кризис, связанный с выявлением парадоксальности оснований математики.

К концу XIX века математика уже очень сильно развилась, были построены матанализ и алгебра, имелись важные результаты из теории чисел. Здание математики было высоким и красивым, но его основание было непрочным. Теория множеств и логика находились в зачаточном состоянии, а ведь на них должна опираться вся математика. Фактически проблемы находились в общем построении математики, так изменение одного постулата/аксиомы геометрии привели к созданию разных неевкидовых геометрий, причем все они были строго обоснованы и их одновременное существование вызывало смешанные чуства у математиков. Все это привело к необходимости четкой формулировки и строгому математическому описанию теоретических основ построения аксиоматической теории. Фактически аксиоматика всегда лежала в основе всех научных теорий, но теперь нужно было сделать строгое математическое описание этого подхода. В результате на рубеже XIX и XX веков Давид Гильберт поставил задачу перед математиками всего мира: построить всеобщую аксиоматику (метаматематику) на примере арифметики. К выбору аксиом надо было подойти очень ответственно. Из них должны выводиться привычные нам свойства чисел. Аксиомы не должны противоречить друг другу (это свойство аксиоматики называется *непротиворечивостью*). Аксиом должно быть достаточно, чтобы о любом утверждении можно было сказать, истинно оно или ложно (это свойство аксиоматики называется *полнотой*). Далее в начале XX века Давид Гильберт провозгласил цель аксиоматизировать всю математику, и для завершения этой задачи оставалось доказать непротиворечивость и логическую полноту арифметики натуральных чисел. 7 сентября 1930 года в Кёнигсберге проходил научный конгресс по основаниям математики, и на этом конгрессе 24-летний Курт Гёдель впервые обнародовал две фундаментальные теоремы о неполноте, показавшие, что программа Гильберта не может быть реализована, метаматематика не возможна. Это заставило математиков создать математическую теорию алгоритмов, которая должна была определять границы разрешимости/доказуемости различных аксиоматических теорий.

Теóрия алгорúтмов — наука, находящаяся на стыке математики и информатики, изучающая общие свойства и закономерности алгоритмов и разнообразные формальные модели их представления. К задачам теории алгоритмов относятся формальное доказательство алгоритмической неразрешимости задач, асимптотический анализ сложности алгоритмов, классификация алгоритмов в соответствии с классами сложности, разработка критериев сравнительной оценки качества алгоритмов и т. п. Вместе с математической логикой теория алгоритмов образует теоретическую основу вычислительных наук.
Теория алгоритмов развивается, главным образом, по трём направлениям:

1. Классическое исследование алгоритмов:
    - Формальная формулировка задач;
    - Понятие проблемы разрешения;
    - Классификация уровней сложности («P», «NP» и другие).
2. Анализ алгоритмов:
    - Асимптотический анализ сложности задач и скорости работы алгоритов.

        - Оценка ресурсоёмкости и времени выполнения (в частности, для рекурсивных алгоритмов);
        - Оценка роста потребности в ресурсах (например, времени выполнения) с увеличением объёма данных.
    - Практический анализ трудоемкости алгоритмов:
        - Получение «явных» функции трудоёмкости;
        - Интервальный анализ функций;
        - Поиск критериев качества;
        - Методика рационального выбора.

3. Построение эффективных алгоритмов (разработка методов повышения эффективности алгоритмов).

Основная задача Теории Алгоритмов — вопрос о существовании алгоритма решения той или иной задачи. Основой для такой постановки задачи стали работы Давида Гильберта о метаматематике — теории акиоматических систем, которая должна была дать основу для автоматизации доказательства любой теоремы. Однако она столкнулась с проблемой разрешимости акиоматической системы, т. е. существовании алгоритмов вывода или доказательства теорем в этой системе. Оказалось вопрос существования алгоритмов решения задач требует особой проработки.

Развитие теории алгоритмов начинается с доказательства Куртом Гёделем теорем о неполноте формальных систем, включающих арифметику, первая из которых была доказана в 1931 году. Возникшее в связи с этими теоремами предположение о невозможности алгоритмического разрешения многих математических проблем (в частности, проблемы выводимости в исчислении предикатов) вызвало необходимость стандартизации понятия алгоритма. Первые стандартизованные варианты этого понятия были разработаны в 1930-е годы в работах Алана Тьюринга, Эмиля Поста и Алонзо Чёрча. Предложенные ими машина Тьюринга, машина Поста и лямбда-исчисление оказались эквивалентными друг другу. Основываясь на работах Гёделя, Стивен Клини ввёл понятие рекурсивной функции, также оказавшееся эквивалентным вышеперечисленным. Одним из наиболее удачных стандартизованных вариантов алгоритма является введённое Андреем Марковым понятие нормального алгоритма. Оно было разработано десятью годами позже работ Тьюринга, Поста, Чёрча и Клини в связи с доказательством алгоритмической неразрешимости ряда алгебраических проблем. В последующие годы значительный вклад в теорию алгоритмов внесли Дональд Кнут, Альфред Ахо и Джеффри Ульман.

Одним из наиболее замечательных достижений математической логики и теории алгоритмов явилась разработка понятия рекурсивной функции и формулировка тезиса Чёрча, утверждающего, что понятие рекурсивной функции является уточнением интуитивного понятия алгоритма. Само понятие «алгоритм» сформировалось лишь в первой половине XX века. Началом систематической разработки теории алгоритмов можно считать 1936 году, когда А. Чёрч опубликовал первое уточнение понятия вычислимой функции и привёл первый пример функции, не являющейся вычислимой. Приблизительно в это же время появились первые уточнения понятия алгоритма (в терминах идеализированных вычислительных машин). В дальнейшем теория алгоритмов получила развитие в трудах многих математиков (А. М. Тьюринг, Э. Л. Пост, С. К. Клини, А. А. Марков, А. Н. Колмогоров и др.).

Проблема построения алгоритма, обладающего теми или иными свойствами, называется *алгоритмической проблемой*. Важный пример алгоритмической проблемы — проблема вычисления данной функции (требуется построить алгоритм, вычисляющий эту функцию). Функция называется *вычислимой*, если существует вычисляющий ее алгоритм. Основными математическими моделями понятия алгоритма являются машины Тьюринга, частично рекурсивные функции и др.

Анализ трудоёмкости алгоритмов (теория сложности вычислений)

Цель анализа — нахождение оптимального алгоритма. Критерий — трудоёмкость (количество необходимых алгоритму элементарных операций). Функция трудоёмкости — соотношение трудоёмкости с входными данными. Один из упрощённых видов анализа затратности алгоритмов — асимптотический, с большим объёмом входных данных. Используемая оценка функции трудоёмкости — **«сложность» алгоритма**, позволяющая определить связь трудоёмкости с объёмом данных. В асимптотическом анализе алгоритмов используются обозначения, принятые в математическом асимптотическом анализе.

21.Основная задача теории алгоритмов. Понятие неразрешимой задачи. Экстраалгоритм.

Неразрешимые задачи и экстраалгоритм

Одна из проблем решаемых теорией алгоритмов — это сложность задач и сложность требуемых алгоритмов. При этом был определен класс NP-алгоритмов, который имел максимальную сложность. Однако можно определить класс еще более сложных алгоритмов, которые невозможны с точки зрения основных свойств алгоритмов. Если NP-алгоритмы можно рассматривать как потенциально осуществимые алгоритмы, то эти алгоритмы не осуществимы даже потенциально или требуют бесконечного числа операций. В теории алгоритмов они получили название экстраалгоритмов. Как правило экстраалгоритмы возникают при попытке решения задач очень большой общности. Чем более общая задача решается, тем сложнее нужен алгоритм. Это так называемые массовые проблемы. Например попытка построить алгоритм, который мог бы проверять возможность использования произвольного алгоритма для произвольного данного является экстраалгоритмом. Это и проблема самоприменимости — существует ли алгоритм проверяющий применение любого алгоритма к самому себе. Таким образом, проблема отсутствия алгоритма проверки применимости любого алгоритма фактически приводит к экстраалгоритму, то есть к неразрешимости задачи. На этом были основаны парадоксы теории множеств, когда парадоксальность возникает при включении множества как элемента самого себя (пример — парадокс брадобрея «Может ли брадобрей брить всех, кто не бреется сам?»).

> Принято задачи, которые решаются экстраалгоритмом называть неразрешимыми. В теории алгоритмов особой проблемой остается доказательство неразрешимости тех или иных задач. По крайней мере, такое доказательство позволило бы прекратить поиски метода ее решения.

Как правило возможность строгого доказательства алгоритмической неразрешимости вознакает при исследовании различных массовых проблем (проблем нахождения единого метода решения некоторого класса задач, условия которых могут варьироваться в известных пределах). Чем более массовая задача, тем больше вероятность того, что алгоритм ее общего решения — экстраалгоритм.

Простейшие примеры алгоритмически-неразрешимой массовой проблемы — проблема применимости алгоритма к самому себе, проблема остановки, которая заключается в следующем: требуется найти общий метод, который позволял бы для произвольной машины Тьюринга (заданной посредством своей программы) и произвольного начального состояния ленты этой машины определить — завершится ли работа машины за конечное число шагов, или же будет продолжаться неограниченно долго? В течение первого десятилетия истории теории алгоритмов, неразрешимые массовые проблемы были обнаружены лишь в ней самой (в том числе, описанная выше «проблема применимости») и в математической логике («проблема выводимости» в классическом исчислении предикатов). Впоследствии, была установлена алгоритмическая неразрешимость и многих других «чисто математических» массовых проблем.

Теорема Геделя. Разрешимость аксиоматических теорий

Теорема Гёделя о неполноте и **вторая теорема Гёделя** — две теоремы математической логики о принципиальных ограничениях формальной арифметики и, как следствие, всякой формальной

системы, в которой можно определить основные арифметические понятия: натуральные числа, 0, 1, сложение и умножение.

Первая теорема утверждает, что если формальная арифметика непротиворечива, то в ней существует невыводимая и неопровержимая формула.

Вторая теорема утверждает, что если формальная арифметика непротиворечива, то в ней невыводима некоторая формула, содержательно утверждающая непротиворечивость этой арифметики.

Обе эти теоремы были доказаны Куртом Гёделем в 1930 году (опубликованы в 1931)

Если формальная система S *непротиворечива, то в ней есть невыводимые формулы или наоборот, если система S непротиворечива, то она неполна.*
В формальной арифметике S можно построить такую формулу, которая в стандартной интерпретации является истинной в том и только в том случае, когда теория S непротиворечива. Для этой формулы справедливо утверждение второй теоремы Гёделя:

> *Если формальная арифметика* S *непротиворечива, то в ней невыводима формула, содержательно утверждающая непротиворечивость* S.

Иными словами, непротиворечивость формальной арифметики не может быть доказана средствами этой теории. Однако, могут существовать доказательства непротиворечивости формальной арифметики, использующие средства, невыразимые в ней.

Доказательство теоремы Гёделя обычно проводят для конкретной формальной системы (не обязательно одной и той же), соответственно утверждение теоремы оказывается доказанным только для одной этой системы. Исследование достаточных условий, которым должна удовлетворять формальная система для того, чтобы можно было провести доказательство её неполноты, приводит к обобщениям теоремы на широкий класс формальных систем.
Фактически сегодня можно говорить о формулировании гипотезы Геделя о неполноте любой непротиворечивой формальной аксиоматической теории (АТ). Это гипотеза, так как общего доказательства нет, но нет и опровержения, для многих АТ эта теорема доказана.

Применение теории алгоритмов для исследования различных разделов математики:

- Невозможность единого алгоритма проверки общезначимости (т.Черча 1936).
- Доказательство существования алгоритмов разрешимости, непротиворечивости, полноты различных аксиоматических теорий (АТ - Аксиомы, правила вывода, теоремы. Доказательство, доказуемость. Разрешимость. Полнота и непротиворечивость АТ).
- Теорема Геделя о противоречивости полных теорий.
- Аксиоматические Теории метаматематики Гильберта и Генцена.
- Непротиворечивость исчисления высказываний и предикатов.

Лекция № 2. **Интуитивное понятие алгоритма и его свойства. Способы представления алгоритмов. Классификации алгоритмов. Основные методы разработки алгоритмов и алгоритмических структур. Рекурсия в алгоритмизации. Языки программирования.**

2.Понятие алгоритма. Принцип потенциальной осуществимости. Основные свойства алгоритмов. Понятие исполнителя алгоритмов.

3. Понятие алгоритма и исполнителя алгоритма. Основные свойства алгоритмов.

Понятие алгоритма

Алгоритм относится к основным понятиям математики, а потому не имеет точного определения. Часто это понятие формулируют так: *«точное предписание о порядке выполнения действий из заданного фиксированного множества для решения всех задач заданного класса»*.

Рассмотрим подробнее ключевые слова в этой формулировке:

− ***«точное предписание»*** означает, что предписание однозначно и одинаково понимается всеми исполнителями алгоритма и при одних и тех же исходных данных любой исполнитель получает один и тот же результат;

− ***«из заданного фиксированного множества»*** означает, что множество действий, используемых в предписании, оговорено заранее и не может меняться в ходе исполнения алгоритма;

− ***«решения всех задач заданного класса»*** означает, что это предписание предназначено для решения класса задач, а не для одной отдельной задачи.

Алгоритм всегда определяет однозначно, какое действие должно быть выполнено следующим, равно как и то, какое действие должно быть выполнено следующим.

Применительно к ЭВМ алгоритм определяет вычислительный процесс, начинающийся с обработки некоторой совокупности возможных исходных данных и направленный на получение определённых этими исходными данными результатов.

Для задания алгоритма необходимо описать следующие его элементы:

- набор объектов, составляющих совокупность возможных исходных данных, промежуточных и конечных результатов;
- правило начала;
- правило непосредственной переработки информации (описание последовательности действий);
- правило окончания;
- правило извлечения результатов.

Понятие исполнителя алгоритма

Алгоритм всегда рассчитан на конкретного исполнителя. **Исполнитель** в информатике – человек или автоматическое устройство, которому поручается исполнить алгоритм или программу. Исполнителем может быть человек, группа людей, робот, станок, компьютер, язык программирования и т.д. Важнейшим свойством, характеризующим любого из этих исполнителей, является то, что исполнитель умеет выполнять некоторые команды. Так, исполнитель-человек умеет выполнять такие команды, как «встать», «сесть», «включить компьютер» и т.д., а исполнитель-язык программирования Бейсик – команды PRINT, END, LIST и другие аналогичные. Вся совокупность команд, которые данный исполнитель умеет выполнять, называется **системой команд исполнителя** (СКИ). Область, в пределах которой действует исполнитель, называется **средой исполнителя**. Одно из принципиальных обстоятельств состоит в том, что исполнитель не вникает в смысл того, что делает, но получает необходимый результат. В таком случае говорят, что исполнитель действует **формально**, то есть отвлекается от поставленной задачи и только строго выполняет некоторые требования, инструкции.

Это важная особенность алгоритмов. Наличие алгоритма формализует процесс решение задачи, искажает рассуждения исполнителя. Использование алгоритма даёт возможность решать задачу формально, механически исполняя команды алгоритма в указанной последовательности. Целесообразность предусматриваемых алгоритмом действий обеспечивается точным анализом со стороны того, кто составляет этот алгоритм.

Введение в рассмотрение понятия «исполнитель» позволяет определить **алгоритм** как понятное и точное предписание исполнителю совершить последовательность действий, направленных на достижение поставленной цели. Наиболее распространенными и привычными являются алгоритмы работы с величинами – числовыми, символьными, логическими и т.д.

Свойства алгоритма

Свойства алгоритма принято делить на основные (необходимые по определению, без выполнения которых алгоритм не является алгоритмом) и сравнительные (с помощью которых можно сравнивать алгоритмы между собой).

Основные свойства:

- Дискретность. Алгоритм должен делить процесс решения задачи на последовательное выполнение простых (или ранее определенных) шагов (этапов).

- Детерминированность (определенность). Каждый шаг алгоритма должен быть четким, однозначным и не оставлять места для произвольного выбора.
- Результативность (конечность). За конечное число шагов алгоритм либо должен приводить к решению задачи, либо после конечного числа шагов останавливаться из-за невозможности получить решение с выдачей соответствующего сообщения.

Сравнительные свойства:

- Массовость – умение решать определенный класс задач.

Рекурсивность – использование процедур, функций, циклов и других повторяющихся конструкций

4. Принцип потенциальной осуществимости

Алгоритм должен давать результат за конечное число шагов. Таким образом, бесконечный алгоритм – это некая теоретическая абстракция, реальный алгоритм должен всегда иметь условие прерывания бесконечного повторения в таком алгоритме. Однако даже конечный алгоритм может быть настолько «сложным» (т.е. выполняться долго), что о его осуществимости можно говорить только потенциально. Кроме того, термин потенциальной осуществимостей может использоваться при условии наличия алгоритма, который невозможно или очень сложно исполнить в реальности. Таким образом, многие алгоритмы могут рассматриваться только как воображаемый процесс. Их осуществимость определяется как потенциальная в отличие от реальных алгоритмов, которые могут быть исполнены практически

8.Алгоритмизация и программирование.

Алгоритмизация и программирование

Алгоритмизация — процесс построения алгоритма для решения задачи.

Программирование — процесс построения программы для реализации алгоритма решения задачи.

Методы разработки и анализа алгоритмов

Нисходящим проектированием алгоритмов, *проектированием* алгоритмов "сверху вниз" или методом последовательной (пошаговой) *нисходящей разработки* алгоритмов называется такой метод составления алгоритмов, когда исходная задача (алгоритм) разбивается на ряд вспомогательных подзадач (подалгоритмов), формулируемых и решаемых в терминах более простых и элементарных операций (процедур). Последние, в свою очередь, вновь разбиваются на более простые и элементарные, и так до тех пор, пока не дойдём до команд исполнителя. В терминах этих команд можно представить и выполнить полученные на последнем шаге разбиений подалгоритмы (команд системы команд исполнителя).

Восходящий метод , наоборот, опираясь на некоторый, заранее определяемый корректный набор подалгоритмов, строит функционально завершенные подзадачи более общего назначения, от них переходит к более общим, и так далее, до тех пор, пока не дойдем до уровня, на котором можно записать решение поставленной задачи. Этот метод известен как метод *проектирования* "снизу вверх".

Структурные принципы алгоритмизации (***структурные методы*** алгоритмизации) – это принципы формирования алгоритмов из базовых структурных алгоритмических единиц (следование, ветвление, повторение), используя их последовательное соединение или вложение друг в друга с соблюдением определённых правил, гарантирующих читабельность и исполняемость алгоритма сверху вниз и последовательно.

Структурированный алгоритм – это алгоритм, представленный как следования и вложения базовых алгоритмических структур. У структурированного алгоритма статическое состояние (до актуализации алгоритма) и динамическое состояние (после актуализации) имеют одинаковую логическую структуру, которая прослеживается сверху вниз ("как читается, так и исполняется"). При структурированной *разработке* алгоритмов правильность алгоритма можно проследить на каждом этапе его построения и выполнения.

Теорема (о структурировании) Бема-Якопини. Любой алгоритм может быть эквивалентно представлен структурированным алгоритмом, состоящим из базовых алгоритмических структур (линейная, ветвление, цикл).

Одним из широко используемых методов *проектирования* и *разработки* алгоритмов (программ) является *модульный метод* (модульная технология).

Модуль – это некоторый алгоритм или некоторый его блок, имеющий конкретное наименование, по которому его можно выделить и актуализировать. Иногда модуль называется **вспомогательным алгоритмом**, хотя все алгоритмы носят вспомогательный характер. Это название имеет смысл, когда рассматривается динамическое состояние алгоритма; в этом случае можно назвать вспомогательным любой алгоритм, используемый данным в качестве блока (составной части) тела этого динамического алгоритма. Используют и другое название модуля – **подалгоритм**. В программировании используются синонимы – процедура, подпрограмма.

Свойства модулей:

- *функциональная целостность и завершенность* (каждый модуль реализует одну функцию, но реализует полностью);
- *автономность и независимость от других модулей* (независимость работы модуля-преемника от работы модуля-предшественника; при этом их связь осуществляется только на уровне передачи/приема параметров и управления);
- *эволюционируемость* (развиваемость);
- *открытость* для пользователей и разработчиков (для модернизации и использования);

Свойства (преимущества) модульного *проектирования* алгоритмов:

- возможность *разработки* алгоритма группой исполнителей;
- *возможность создания и ведения библиотеки* наиболее часто используемых алгоритмов (подалгоритмов);
- облегчение *тестирования* алгоритмов и обоснования их правильности ;
- упрощение *проектирования* и модификации алгоритмов ;
- уменьшение сложности *разработки* (*проектирования*) алгоритмов (или комплексов алгоритмов);
- ***наблюдаемость вычислительного процесса*** **при реализации алгоритмов.**

Понятие о типах данных

До разработки алгоритма (программы) необходимо выбрать оптимальную для реализации задачи структуру *данных*. Неудачный выбор *данных* и их описания может не только усложнить решаемую задачу и сделать ее плохо понимаемой, но и привести к неверным результатам. На структуру *данных* влияет и выбранный метод решения.

Тип *данных* характеризует область определения *значений данных*. Задаются типы *данных* простым перечислением *значений* типа, например как в *простых типах данных* , либо объединением (структурированием) ранее определенных каких-то типов – *структурированные типы данных*. Для обозначения текущих *значений данных* используются константы – числовые, текстовые, логические. Часто (в зависимости от задачи) рассматривают *данные*, которые имеют не только "линейную" (как приведенные выше), но и иерархическую структуру. В алгоритмических языках есть **стандартные типы**, например, целые, вещественные, символьные, тестовые и логические типы. Они в этих языках не уточняются (не определяются, описываются явно) и имеют соответствующие описания с помощью служебных слов. Каждый тип *данных* допускает использование определенных операций со *значениями* типа ("с типом").

Для описания величин, значение которых может изменяться вводятся переменные. К структурным типам данных можно отнести – массивы, множества, строки, списки, деревья, таблицы, файлы. Наиболее часто используемая структура данных – *массив*. Одномерный *массив* (вектор, *ряд*, линейная таблица) – это совокупность *значений* некоторого простого типа (целого, вещественного, символьного, текстового или логического типа), перенумерованных в каком-то порядке и имеющих общее имя. Для выделения конкретного элемента *массива* необходимо указать его порядковый номер в этом *ряду*. *Значение* порядкового номера элемента *массива* называется индексом элемента. Двумерный *массив* (*матрица*, прямоугольная таблица) –

совокупность одномерных векторов, рассматриваемых либо "горизонтально" (векторов-строк), либо "вертикально" (векторов-столбцов) и имеющих одинаковую размерность, одинаковый тип и общее имя. *Матрицы*, как и векторы, должны быть в алгоритме описаны служебным словом (например, таб или array), но в отличие от вектора, *матрица* имеет описание двух индексов, разделяемых запятыми: первый определяет начальное и конечное *значение* номеров строк, а второй – столбцов. Для актуализации элемента двумерного *массива* нужны два его индекса – номер строки и номер столбца, на пересечении которых стоит этот элемент.

9.Методы и средства программирования. Классификация языков программирования.

Методы и средства программирования. Классификация языков программирования.

Основы программирования

Программирование — процесс создания структуры программы, который происходит по следующим этапам:

1. Постановка задачи (выделяются цель, средства ее достижения и условия решения задачи).
2. Создание алгоритма решения задачи.
3. Запись алгоритма в виде программы на языке программирования.
4. Отладка программы.
5. Тестирование программы.

Разработка программы состоит из двух различных действий – создания алгоритма и представления его в виде программы. При этом, создание алгоритма – это, как правило, наиболее сложный этап, т.к. создать алгоритм – значит, найти метод решения задачи. Поэтому, чтобы понять, как создать алгоритм, необходимо понять процесс решения задачи.

Тестирование алгоритма – это проверка правильности или неправильности работы алгоритма на специально заданных *тестах* или тестовых примерах – задачах с известными входными данными и результатами (иногда достаточны их приближения). Тестовый набор должен быть минимальным и полным, то есть обеспечивающим проверку каждого отдельного типа наборов входных данных, особенно исключительных случаев.

Трассировка – это метод пошаговой фиксации динамического состояния алгоритма на некотором *тесте*. *Трассировка* облегчает отладку и понимание алгоритма.

Процесс поиска и исправления (явных или неявных) ошибок в алгоритме называется **отладкой алгоритма**. Как правило отладка происходит совместно с процессом **трансляции** программы (преобразования из кода языка программирования в команды процессора ЭВМ). Трансляция выполняется либо компилятором, либо интерпретатором языка программирования.

Перед программированием задачи может быть составлена графическая **блок-схема алгоритма**, которая составляется по специальным правилам из специальных графических блоков.

По одной из классификаций методы программирования делятся на следующие типы:

- Операциональные
- Процедурные, называемые также директивные (directive) или императивными (imperative),
- Декларативные (declarative) языки,
- Объектно-ориентированные (object-oriented).

Операциональное программирование обычно рассматривается на примере машинно-ориентированного языка Ассемблер (хотя оно присутствовало и в первых языках высокого уровня). **Суть этого метода программирования — представлять программу как последовательный набор операций исполнителя (как правило — процессор ЭВМ).**

Замечание: При создании компьютерных средств программирования исторически первым явилось использование программ написанных в двоичных кодах (программирование в кодах). В дальнейшем были созданы первые ассемблеры, в которых были специальные текстовые конструкции (команды), которые требовалось переводить (транслировать) в двоичные коды. Так

появились трансляторы языков программирования (интерпретаторы и компиляторы). Последующее развитие операционального программирование встретило большие трудности, которые были разрешены появлением декларативного и структурного подхода. Теоретическое развитие структурного подхода (Вирт, Якопини, Тьюринг, Бем) позволило создать процедурные зыки (классическим примером является язык Паскаль, созданный Н.Виртом как образец языка структурного/процедурного программирования).

К процедурным языкам относятся такие классические языки программирования, как Algol, Fortran, Basic, Pascal, C. **Суть этого метода программирования — представлять программу в виде структуры процедур исполнителя (как правило — транслятор языка, который преобразует эти процедуры в последовательность операций ЭВМ).**

Наиболее существенными классами декларативных языков являются **функциональные** (functional), и реляционные - **логические** (logic) языки. К категории функциональных языков относятся, например, Lisp и Haskell. Самым известным языком логического программирования является Prolog (Пролог). **Суть этого метода программирования — представлять программу как описание (декларацию) задачи с указанием требуемой цели (целевое указание). Для достижения этой цели строится автоматически алгоритм с помощью специального ядра системы, освобождая тем самым программиста от рутинных операций.**

Среди объектно-ориентированных языков программирования (языков ООП) отметим C++, Java, Object Pascal, Python. **Суть этого метода программирования — представлять программу как набор взаимодействующих объектов, обладающих собственными встроенными алгоритмами — методами. Для взаимодействия объектов проектируется некая среда, функционирующая автономно и управляемая с помощью событийного управления (обработки событий в среде).**

Особенность операционального программирования – ориентация на построение инструкции для исполнителя – процессора ЭВМ в терминах его операций. Главный недостаток – сложность программ, злоупотребление оператором перехода GOTO.

Особенность процедурного программирования – переход к более строгому, логически структурному программированию. Основные принципы процедурного программирования – использование стандартных технологий проектирования, структурного и модульного принципов, исключение GOTO, максимальное использование стандартных алгоритмов. Главный недостаток – постепенное загромождение модулей процедурами и функциями, недостаточная читабельность и понимаемость больших программ.

Особенность декларативного программирования – программист только описывает (создает декларацию) задачу, а методы и средства ее решения подбирает специальная программа – ядро языка программирования. Главный недостаток – недостаточное развитие среды и средств программирования для данных языков.

Особенность ООП – программист описывает структуру классов для объектов (создает иерахию классов по технологии наследования) и использует специальную среду проекта (как правило предоставляемую системой визуального программирования), затем описывает методы классов и обработчики событий для среды и классов/объектов. Главный недостаток – необходимость использования специальной технологии объектной декомпозиции, которая требует специальных навыков и знаний от программиста.

Лекция №3. **Запись алгоритмов с помощью языка блок-схем. Основные алгоритмические структуры. Примеры записи алгоритма с помощью языка блок-схем.**

Итерационные и циклические алгоритмы. Подпрограммы. Методы повышения эффективности алгоритмов.

3.Классификация алгоритмов. Блок-схемы описания алгоритмов. Формы записи алгоритмов.

Классификация алгоритмов.

Так как алгоритмов очень много существует множество вариантов их классификации. На практике наиболее употребительна классификация по типу решаемых задач.

1. Вычислительный алгоритм (обработка некоторой совокупности возможных исходных данных и получение результата).
2. Логический алгоритм (проверка условия).
3. Моделирующий алгоритм (алгоритм создания и заданного функционирования математической модели). Данный тип алгоритмов используется для описания поведения модели объекта.
4. Адаптивный алгоритм (обладает способностью настраиваться на решаемую задачу).
5. Вероятностный алгоритм (использует случайные данные, результат его так же в каком-то смысле случайный).
6. Алгоритм формирования и функционирования объекта, объектно-ориентированное программирование. Описывает объект какого-то класса. От моделирующего отличается тем, что объект реальный (в моделирующем алгоритме) и объект класса различны по своей природе. Реальный объект – это объект, существующий в природе, для которого создаётся математическая модель и затем данная модель и её функционирование реализуются с помощью алгоритма. Объект класса – это просто структура данных.

Запись алгоритмов. Способы представления алгоритмов

Для записи алгоритмов используются следующие варианты:

- Текстовое представление в виде пронумерованной последовательности инструкций
- Графический метод (различные виды блок-схем)
- Использование языков программирования
- Использование языка псевдокода
- Использование программ специальных алгоритмических машин (Тьюринга, Поста) или алгорифмов Маркова

Язык Блок-схем

Язык Блок-схем содержит специальные геометрические конструкции – овал для начала, конца, возврата алгоритма, прямоугольник для обычных вычислений, ромб для условий, шестиугольник для циклов, круг для соединителя.

Способы представления алгоритмов

Любой возможный алгоритм может быть представлен в виде совокупности трёх основных конструкций: линейной, разветвлённой, циклической.

Линейные алгоритмы

Линейным называется алгоритм, в котором все этапы решения задачи выполняются последовательно. Каждая операция является самостоятельной, независимой от каких- либо условий. На схеме блоки, отображающие эти операции, располагаются в линейной последовательности.

Данный алгоритм имеет следующий формат:

Начало <последовательность выполняемых команд>; Конец.

Разветвленные алгоритмы

Ветвящимся называется такой алгоритм, в котором выбирается один из нескольких возможных путей (вариантов) вычислительного процесса. Каждый подобный путь называется **ветвью алгоритма**. Признаком ветвящегося алгоритма является наличие операций проверки условия. Обычно различают два вида условий – простые и составные [2].

Простым условием (отношением) называется выражение, составленное из двух арифметических выражений или двух текстовых величин, связанных одним из знаков: >, <, ≤, ≥, =, ≠.

Из отношений можно образовать **сложные (составные)** выражения, с помощью логических операций И, ИЛИ, НЕ.

Например, x<7 и x>2; y>0 или (y<0 и x<0).

В схеме алгоритма операцию проверки выполняет **логический блок**. Он изображается ромбом, внутри которого указывается проверяемое условие (отношение), и имеет два выхода: Да и Нет.

Если условие (отношение) истинно (выполняется), то выходим из блока по выходу «Да»; если ложно (не выполняется) – по выходу «Нет».

Ветвящийся алгоритм, включающий в себя две ветви, называется простым, более двух ветвей – сложным. Сложный ветвящийся алгоритм можно представить с помощью простых ветвящихся алгоритмов.

Ветвление реализуется в виде команды:

если <ЛВ> то <Серия 1> иначе <Серия 2>.

Здесь **<ЛВ>** – это логическое выражение;

<Серия 1> – описание последовательности действий, которые должны выполниться, когда **<ЛВ>** принимает значение «истина»;

<Серия 2> – описание последовательности действий, которые должны выполниться, когда **<ЛВ>** принимает значение «ложь».

Любая из этих серий может быть пустой. В этом случае ветвление называется неполным. В противном случае ветвление называется полным.

Циклические алгоритмы

Циклом называется последовательность действий, выполняемых многократно, каждый раз при новых значениях параметров.

Алгоритмы, включающие в себя циклы, называются **циклическими**.

Различают циклические алгоритмы с параметром, с предусловием и с постусловием.

1. **Цикл с параметром** (со счётчиком) применяется в тех случаях, когда в программе какие-то действия (операторы) повторяются и при этом некоторая величина меняется с постоянным шагом.

Формат команды на алгоритмическом языке имеет следующий вид:

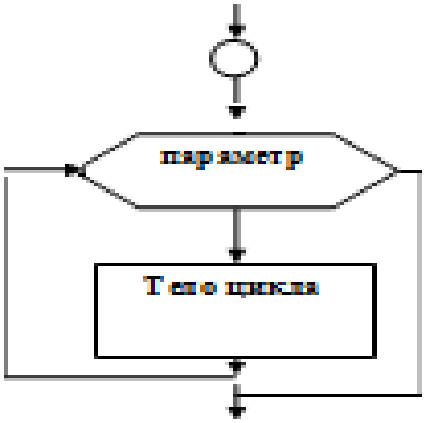
НЦ

Для <параметр цикла> от <начальное значение> до <конечное значение>

Выполнить <оператор>

КЦ

Схема, иллюстрирующая работу оператора цикла с параметром, выглядит следующим образом:

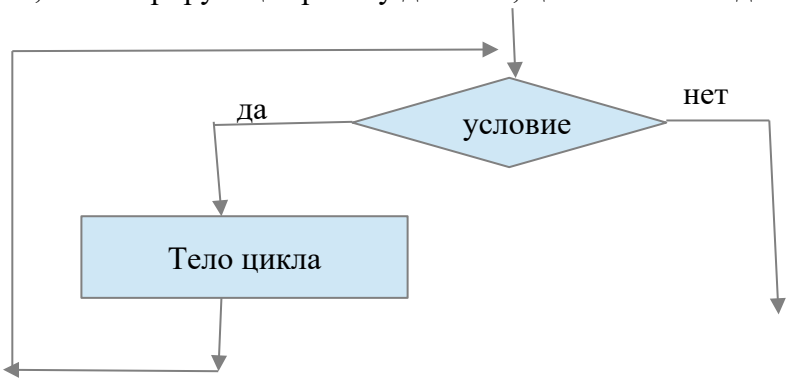


2. Оператор **цикла с предусловием** используется в тех случаях, когда число повторений действий в программе неизвестно. Его общий вид:

НЦ
Пока <условие>
<тело оператора цикла>
КЦ

Где <условие> – условие, при котором выполняется <тело оператора цикла>.

Схема, иллюстрирующая работу данного, цикла имеет вид:



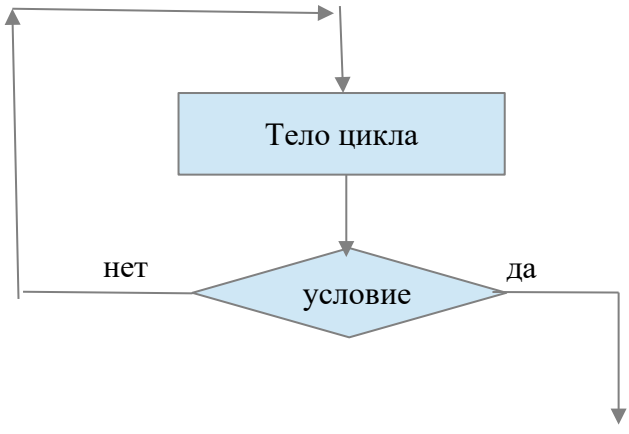
3. Оператор **цикла с постусловием** применяется тогда, когда число повторений действий заранее не известно. Данный оператор отличается от предыдущего тем, что тело цикла повторяется не менее одного раза.

Общий вид этого оператора:

НЦ
<тело оператора цикла>
КЦ при <условие>,

где <условие> – условие, при котором оператор прекращает свою работу.

Схема, иллюстрирующая работу оператора с постусловием:



Цикл называется **детерминированным**, если число повторений тела цикла заранее известно или оговорено. Примером может служить цикл с использованием оператора FOR.

Цикл называется **итерационным,** если число повторений тела цикла заранее не известно, а зависит от значений параметров (некоторых переменных), участвующих в вычислениях. Например, цикл с оператором WHILE DO или REPEAT UNTIL.

Особую задачу имеет выделение и описание подпрограмм (процедур и функций), которые выделяются в отдельную блок-схему и должны завершаться особым блоком «возврат» вместо блока «конец». В основной блок-схеме подпрограмма вызывается с помощью специального блока вызова подпрограммы — прямоугольник с двойными линиями слева и справа.

10.Методы построения эффективных алгоритмов.

Методы построения эффективных алгоритмов
Рекурсия и итерация

Алгоритм, в состав которого входит итерационный цикл, называется **итерационным** алгоритмом.

Такие алгоритмы используются при реализации итерационных численных методов.

В итерационных алгоритмах необходимо обеспечить обязательное достижение условия выхода из цикла – **сходимость итерационного процесса**. В противном случае произойдет зацикливание алгоритма, т.е. не выполнится свойство результативности.

При составлении новых алгоритмов могут использоваться алгоритмы, составленные ранее. Алгоритмы, целиком используемые в составе других алгоритмов, называют вспомогательными алгоритмами. Алгоритм может содержать обращение к себе самому как вспомогательному, и в этом случае его называют **рекурсивным**. Рекурсия является средством программирования, при котором процедура или функция прямо или косвенно вызывает сама себя. Если команда обращения алгоритма к самому себе находится в самом алгоритме, то такую рекурсию называют **прямой**. Возможны случаи, когда рекурсивный вызов данного алгоритма происходит из вспомогательного алгоритма, к которому в данном алгоритме имеется обращение. Такая рекурсия называется **косвенной**. Косвенный вызов может быть организован и более сложным образом, то есть в рекурсию могут быть вовлечены несколько подпрограмм.

Рекурсия позволяет, например, очень просто запрограммировать вычисление рекуррентных соотношений. Рассмотрим следующий способ вычисления факториала: $f_0 = 1$, $f_n = f_{n-1} \cdot n$.

При анализе рекурсивной программы обычно возникают два вопроса:

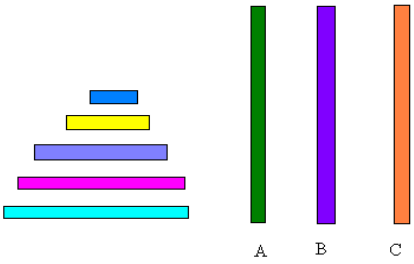
- почему программа заканчивает работу?
- почему она работает правильно?

Чтобы доказать пункт а), обычно проверяют, что с каждым рекурсивным вызовом значение какого-то параметра изменяется (чаще всего уменьшается), и это не может продолжаться бесконечно. Для ответа на вопрос b) достаточно проверить, что содержащая рекурсивный вызов программа работает правильно, предположив, что вызываемая ею одноимённая подпрограмма работает правильно. В самом деле, в этом случае в цепочке рекурсивно вызываемых программ все программы работают правильно (в этом можно убедиться, двигаясь от конца цепочки к началу).

Методы построения эффективных алгоритмов: метод бинарных деревьев, рекурсивные алгоритмы, динамическое программирование

- **Рекурсивные алгоритмы.** Метод заключается в представлении алгоритма вычисления функции в виде рекурсивной формулы. Примером является задача о Ханойских башнях.

  Есть 3 стержня и кольца разных диаметров, которые надеваются на стержни. Причем разрешается помещать кольцо только на кольцо большего диаметра:



Задача: нужно переместить кольца со стержня A на C, используя стержень B как вспомогательный. Для перемещения 2-х колец алгоритм простой — A->B, A->C,B->C. Для большего числа колец алгоритм начинает усложняться и составить процедурый алгоритм для N-колец очень сложно. При рекурсии задача решается просто:

а) Перенесем N-1 кольцо с A на B.

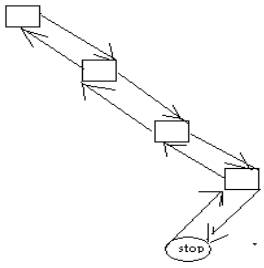
б) Перенесем 1 кольцо с A на C.

в) Перенесем N-1 кольцо с B на C.

т.е. Задача с N кольцом решается через задачу с N-1 кольцом. Тогда N->N-1->N-2->N-3->...2 => задача решена.

Рекурсия – вызов процедуры или функции самой себя. Существует много задач легко решаемых рекурсией, более того, доказано, что любую алгоритмическую задачу можно решить рекурсивно. Это методика рекурсивного программирования.

Действие рекурсии можно представлять как вызов процедуры 1 , затем такой же процедуры 2 и т.д. этот вызов должен быть конечным, следовательно, должно быть условие остановки рекурсии. Перед каждым вызовом новой итерации рекурсии это условие должно проверяться. После остановки возвращаемся назад, таким образом, идет движение по кругу: вниз потом вверх (спуск и подъем рекурсии).



Рекурсия необходима для выполнения каких-либо действий, поэтому выделяют следующие варианты использования рекурсии:

- Рекурсия с выполнением действий на спуске (до вызова рекурсии).
- Рекурсия с выполнением действий на подъеме (после вызова рекурсии).
- Рекурсия с выполнением действий на спуске и подъеме.

Если сравнить рекурсию и цикл, то главное отличие в том, что цикл – это повторение действий, рекурсия – новый вызов (новый вариант). Это два принципиально разных подхода к выполнению действий. Теоретически доказано, что оба подхода эквивалентны, но на самом деле простые задачи проще понимаются циклом, а сложные – рекурсией. Таким образом, рекурсивный подход к логическому решению более удобен и логичен.

Следует учитывать, что рекурсивное программирование и циклическое программирование настраивает программистов на разное понимание алгоритмизации. Поэтому навыки программирования в процедурном языке могут быть даже вредны для рекурсивного программирования.

- **Метод бинарных деревьев.** Суть метода в представлении решения задачи в виде поиска по бинарному дереву. Примером является задача бинарного поиска. Данный метод является так же усложненной формой метода деления задачи на подзадачи. Например, в алгоритме «вет-

вей и границ» используемого для поиска целочисленного решения задачи оптимизации (поиск максимального или минимального значения многомерной функции), исходную задачу делят на 2 задачи с помощью 2-х дополнительных условий. Эти условия подбирается так, чтобы целочисленные варианты решения удовлетворяли одному из условий, а наилучшее не целочисленное решение не удовлетворяло ни одному из условий. Таким образом, при построении новой ветки бинарного дерева удаляется хотя бы одно не целочисленное решение. Алгоритм работает до тех пор, пока не будет получено целочисленное решение.

- **Метод балансировки бинарных деревьев.** При построении бинарных деревьев желательно сделать его полностью сбалансированным. Сбалансированное бинарное дерево ускоряет алгоритм поиска по дереву. Время выполнения операций пропорционально высоте дерева. Если двоичное дерево плотно заполнено, то его высота пропорциональна логарифму (по основанию 2) от числа вершин и скорость то время $O(\log_2(n))$. Если дерево представляется как простая цепочка, то время поиска линейно от числа вершин O(n). Таким образом, балансировка или «уплотнение» двоичного дерева существенно ускоряет алгоритм. В настоящее время разработаны различные варианты алгоритмов балансировки двоичных деревьев поиска (метод Адельсон-Вельского и Ландиса, Хопкрофта, Байера для красно-черных деревьев и т.д.).
- **Метод динамического программирования**. Данный метод обычно применяется в задачах поиска оптимального решения, которое зависит от времени. Время, как правило, можно представить дискретным набором значений – этапов или шагов. Таким образом, динамическая задача представляется как многоэтапная. Характерным примером метода решения многоэтапной задачи является алгоритм динамического программирования Беллмана. В нем задача представляется в обратном порядке, от последнего этапа до первого (метод обратного планирования). На каждом шаге строится набор вариантов для этапа An, для совокупности этапов (An-1;An), для совокупности этапов (An-2, An-1;An) и т.д.. При достижении алгоритмом 1-го этапа определяется оптимальное решение. Скорость используемого алгоритма будет существенно выше алгоритма полного перебора.
- **Метод последовательного перехода.** Суть метода – оптимальное решение находится в одной точке из конечного множества точек многомерного пространства. Можно организовать полный перебор вариантов. Однако быстрее будет работать алгоритм, который будет проверять подмножество смежных (наиболее близких) точек. Наилучшая из смежных точек, на следующем шаге алгоритма становится новой опорной точкой, которая формирует подмножество смежных точек. Скорость используемого алгоритма будет существенно выше алгоритма полного перебора. Примером может служить алгоритм Симплекс-метода решения оптимизационной задачи линейного программирования.
- **Метод жадного алгоритма.** В данном случае производится проверка на существование жадного алгоритма. Если такой алгоритм решения задачи существует, то его используют. Если жадного алгоритма решения задачи не существует, то часто задачу преобразуют так, чтобы она решалась жадным алгоритмом. Для проверки существования жадного алгоритма используют теорию матроидов. Если задача удовлетворяет условию матроида, то жадный алгоритм ее решения существует.

Лекция №4. **Сложность алгоритма. Асимптотическая оценка сложности алгоритмов. Классы сложности алгоритмов. Класс полиномиальных алгоритмов. Примеры. Класс NP алгоритмов. Примеры. Замкнутость класса NP алгоритмов.**

4.Сложность алгоритмов. Варианты оценки сложности. Асимптотическая сложность алгоритма.

5.Реально выполнимые алгоритмы. Совпадение классов полиномиальных и реально выполнимых алгоритмов.

Сложность алгоритма, оценка сложности алгоритма. Понятие о полиномиальных и реально выполнимых алгоритмах.

Известно, что правильность — далеко не единственное качество, которым должна обладать хорошая программа. Одним из важнейших является эффективность. Эффективность – сравнительная характеристика алгоритма и определяется неким критерием эффективности. Это может быть время выполнения алгоритма, число операций процессора, размер памяти используемой программы и т.д.. Этот критерий определяется как зависимость или функция от различных входных данных (параметра N). В тоже время можно говорить о эффективности алгоритма и сложности задачи. Можно сравнивать разные алгоритмы для одной задачи, а можно для разных задач. Поэтому принято говорить о сложности или скорости алгоритма и о сложности задачи. Так как эти величины есть функции, то можно их сравнивать по асимптотическим оценкам при стремлении параметра N->∞. Нахождение точной зависимости критерия эффективности для конкретной программы — задача достаточно сложная. По этой причине обычно ограничиваются **асимптотическими оценками** этой функции, то есть описанием ее примерного поведения при больших значениях параметра N. При этом для асимптотических оценок используют традиционное отношение O (читается "O большое") между двумя функциями f(n)=O(n). Асимптотические оценки функции при N->∞ можно представить выражением O(f(N)), которое означает const* f(N). Пусть заданы 2 оценки O(f(N)) и O(g(N)). Будем считать, что:

Функция f(N) растет медленнее g(N), если $\lim_{N\to\infty} [O(f(N)) / O(g(N))] = 0$.

Функция f(N) растет быстрее g(N), если $\lim_{N\to\infty} [O(g(N)) / O(f(N))] = 0$.

Функции f(N) и g(N) имеют одинаковую скорость роста, если $\lim_{N\to\infty} [O(f(N)) / O(g(N))] = const \neq 0$.

Аналогично по асимптотике роста критерия эффективности можно говорить о скорости роста алгоритма. Причем, чем больше скорость роста алгоритма, тем он фактически медленнее и наоборот. Соответственно по функции асимптотики можно различать линейные O(N), квадратичные $O(N^2)$, кубические $O(N^3)$ экспоненциальные $O(e^N)$ и др. алгоритмы.

Сложность задачи можно оценить по скорости наиболее эффективного для нее алгоритма. При этом, может оказаться, что наиболее эффективный алгоритм нам пока не известен. Поэтому реальная сложность задачи может оказаться ниже чем представляемая нами, но не может оказаться больше чем текущая.

В качестве примера рассмотрим алгоритм нахождения факториала числа. Легко видеть, что количество операций, которые должны быть выполнены для нахождения факториала N! числа N в первом приближении прямо пропорционально этому числу, ибо количество повторений цикла (итераций) при вычислении по формуле N!=(N-1)!*N программе равно N. В подобной ситуации принято говорить, что алгоритм имеет ***линейную сложность*** (сложность O(N)). Можно ли вычислить факториал быстрее? Оказывается, да. Можно написать такую программу, которая будет давать правильный результат для тех же значений N с логарифмической скоростью. Про алгоритмы, в которых количество операций примерно пропорционально log(N) (в информатике обычно используют для основания двоичный логарифм) говорят, что они имеет ***логарифмическую сложность*** (O(log(N)).

Полиномиальным алгоритмом называют алгоритм, скорость которого может быть представлена полиномом какой-либо конечной степени.

Реально-выполнимым алгоритмом называют алгоритм, у которого время решения задачи возможно для реального использования на практике. Так как время решения зависит не только от алгоритма, но и от задачи (то есть от N), то некоторые задачи могут быть решены при малых N, но не решаются для больших значений N.

Когда начинающие программисты тестируют свои программы, то значения параметров, от которых они зависят, обычно невелики. Поэтому даже если при написании программы был применен неэффективный алгоритм, это может остаться незамеченным. Однако, если подобную программу попытаться применить в реальных условиях, то ее практическая непригодность проявится незамедлительно.

С увеличением быстродействия компьютеров возрастают и значения параметров, для которых работа того или иного алгоритма завершается за приемлемое время. Таким образом, увеличивается среднее значение величины N, и, следовательно, возрастает величина отношения времен выполнения быстрого и медленного алгоритмов. Исходя из практики использования различных алгоритмов для решения задач было принято использовать понятие реально выполнимых алгоритмов только для задач полиномиальной сложности и более простых. Таким образом, класс полиномиальных (P) алгоритмов совпадает с классом реально выполнимых алгоритмов.

6. Полиномиальные и не полиномиальные алгоритмы. Примеры полиномиальных алгоритмов.
7. Примеры задач НП. Задача коммивояжера. Замкнутость класса задач НП.

Полиномиальные и не полиномиальные алгоритмы. Класс NP – алгоритмов

Задачи, которые решаются различными полиномиальными алгоритмами, очень разнообразны, но для информатики наиболее популярны задачи поиска и сортировки. Простой поиск в массиве реализуется алгоритмом линейной скорости от числа элементов массива O(N). Возможен вариант быстрого бинарного поиска, если массив был заранее отсортирован. Тогда скорость алгоритма выше полиномиальной ($O(\log_2(N))$) — логарифмическая. Как правило можно выделить следующую иерархию задач по сложности решения:

- самые простые — логрифмические (бинарный поиск)
- линейные (простой поиск)
- линейно-лографимические $O(N*\log_2(N))$ (быстрая сортировка)
- квадратичные (простая сортировка массива)
- полиномиальная более высоких порядков (например сортировка многомерных массивов)
- NP – задачи (задача коммивояжера, поиск простых сомножителей)
- Экспраалгоритмы

Простые алгоритмы сортировок

К ***простым внутренним сортировкам*** относят методы, сложность которых пропорциональна квадрату размерности входных данных. Иными словами, при сортировке массива, состоящего из N компонент, такие алгоритмы будут выполнять $C*N^2$ действий, где C - некоторая константа. Количество действий, необходимых для упорядочения некоторой последовательности данных, конечно же, зависит не только от длины этой последовательности, но и от ее структуры. Например, если на вход подается уже упорядоченная последовательность (о чем программа, понятно, не знает), то количество действий будет значительно меньше, чем в случае перемешанных входных данных. Как правило, сложность алгоритмов подсчитывают раздельно по количеству сравнений и по количеству перемещений данных в памяти (пересылок), поскольку выполнение этих операций занимает различное время. Однако точные значения удается найти редко, поэтому для оценки алгоритмов ограничиваются лишь понятием "пропорционально", которое не учитывает конкретные значения констант, входящих в итоговую формулу. Общую же эффективность алгоритма обычно оценивают "в среднем": как среднее арифметическое от сложности алгоритма "в лучшем случае" и "в худшем случае", то есть

(Eff_best + Eff_worst)/2. Таким образом, этот алгоритм имеет сложность $O(N^2)$ ("порядка эн квадрат").

Сортировка информации имеет смысл, если информация структурирована. Тогда сортировка предполагает изменение структуры так, чтобы она располагалась по направлению возрастания\убывания некоторого числового критерия сортировки. Соответственно можно говорить о сортировке линейных структур (массивы, таблицы, линейные списки), сортировки деревьев, сортировки различных сетей и т.д.. В простейшем случае сортировки линейной структуры алгоритмы принято делить на простые и быстрые алгоритмы. Простые (медленные) алгоритмы имеют квадратичную по числу элементов скорость и называются квадратичными. К ним относят методы «пузырька» или обменной сортировки, выбора, вставки, дополнительного массива и т.д..

Метод выбора использует алгоритм поиска max\min. Например, для сортировки по возрастанию используем поиск минимального элемента. Найдем min элемент и поменяем его местами с первым, затем повторим это действие с частью массива без 1-го элемента и т.д..

Когда останется 1 элемент, прекратим вычисления – массив отсортирован.

Помимо класса полиномиальных алгоритмов (P – класса), могут существовать задачи не имеющие алгоритма решения полиномиальной скорости. Такие задачи принято называть NP задачами, а алгоритмы их решения NP- алгоритмами.

Доказана теорема о полноте класса NP задач:

Если существует полиномиальное решение хотя бы одной NP-задачи, то на его основе можно построить алгоритмы решения для любой NP – задачи. Таким образом, NP-задачи являются не полиномиальными только в совокупности. Это свидетельствует о полноте класса NP-задач.

Скорость роста NP-алгоритмов представляется либо как $O(e^N)$, $O(K^N)$ или O(N!) (в соответствии с формулой Стирлинга, их скорости близки и выше любой фиксированной степени).

Примерами NP-задач являются задачи:

- Разложение числа на простые сомножители. Если считать за N – число сомножителей, то скорость роста алгоритма O(N!).
- Задача коммивояжера. В данной задаче заданы N городов и расстояния между ними, необходимо построить замкнутый путь, который удовлетворял условиям – минимальности суммарного расстояния, проходил бы через все города и только по одному разу. Фактически это задача поиска минимального гамильтонова пути в графе с заданными длинами ребер. Скорость роста такого алгоритма выше полиномиальной.

Поскольку к классу реально выполнимых алгоритмов относятся только полиномиальные и более быстрые алгоритмы, то NP-алгоритмы выполнимы только потенциально. Это означает, что при увеличении числа элементов в задаче время выполнения такого алгоритма быстро выходит за пределы возможностей его реализации. Поэтому для NP- задач очень важно правильно оценить требуемое количество операций для ее реального решения.

Пример: Оценка времени решения задачи выбранным ЭВМ для разных алгоритмов

Сравнительная таблица времен выполнения алгоритмов

Сложность алгоритма	n=10	$n=10^3$	$n=10^6$
$Log_2(N)$	0.2 сек.	0.6 сек.	1.2 сек.
N	0.6 сек.	1 мин.	16.6 час.
N^2	6 сек.	16.6 час.	1902 года
2^N	1 мин.	10^{295} лет	10^{300000} лет

Лекция №5. **Понятие о методах представления алгоритмов и их роль в теории алгоритмов. Виртуальные алгоритмические машины. Определение машины Тьюринга (МТ). Описание МТ. Работа МТ. Правило останова.**

Лекция №6. **Программа МТ. Тезис Тьюринга. Примеры программирования МТ. Машина Поста. Особенности машины Поста. Сравнение виртуальных алгоритмических машин.**

Тема: Виртуальные вычислительные машины

Содержание: Машина Тьюринга. Работа Машины Тьюринга (МТ). Проблема остановки МТ. Программа Машины Тьюринга. Программирование задач для МТ. Примеры. Машина Поста. Особенности машины Поста.

13.Машина Тьюринга. Работа Машины Тьюринга.

Машина Тьюринга

Машина Тьюринга (МТ) — виртуальная (воображаемая) алгоритмическая машина преобразующая текстовые константы. Таким образом МТ — пример текстовой (словарной) функции, которая на входе имеет исходный текст, а на выходе текст результата. Т.е. она преобразует одно слово (или предложение) в другое слово (предложение) в одном и том же внешнем языке. Для описания текстов вводится внешний алфавит А, который имеет конечное число знаков (теоретически иногда рассматривают и бесконечные алфавиты) для описания исходных и результирующих текстов. Кроме того, определяют еще конечный (или иногда бесконечный) набор внутренних состояний МТ — внутренний алфавит Р.

Идея МТ была разработана британским ученым Аланом Тьюрингом с целью построить алгоритмическую машину так, чтобы с ее помощью можно было бы выделять классы алгоритмов. Так как результат практически любого алгоритма можно представить как преобразование текста, то МТ может описывать большие классы алгоритмов. Этот подход оказался удачным, МТ стала стандартным методом анализа возможности построения алгоритмов.
Тезис Тьюринга. Любой реально выполнимый алгоритм имеет реализующую его МТ.

В отличии от теоремы тезис не имеет общего доказательства, но справедлив во многих случаях (доказан для частных случаев) и не имеет опровергающих примеров. В данном случае тезис защищен от опровержения неопределенностью понятия **реально выполнимый алгоритм,** которое в данном случае наоборот определяется через метод их построения. То есть, Тьюринг придумал свою машину для того, чтобы можно было бы выделить класс выполнимых алгоритмов. Этот метод в теории алгоритмов стал доминирующим. Кроме МТ есть машина Поста, алгорифмы Маркова, рекурсивные функции и т. д.

МТ практически есть пример (частный случай) алгоритмического автомата . Это более широкое понятие чем МТ. Анализ теории автоматов дает возможность рассматривать МТ с многими лентами, многими языками и т. д. Фактически можно назвать автоматом и любую цифровую ЭВМ. Поэтому МТ часто называют примером автомата с памятью, где размер программы и определяет память. Чем больше память, тем более сложную программу можно внести в МТ.

Работа Машины Тьюринга (МТ)

Обычно МТ представляется как объединение бесконечной в обе стороны ленты, разбитой на ячейки, причем в каждую ячейку можно записать только 1 букву внешнего алфавита. В начальный момент времени в ячейке вписаны символы из алфавита А, в остальные ячейки вписаны пустые символы, которые обычно обозначаются пробелами. МТ имеет считывающе-пишущее устройство, которое в любой момент времени просматривает текущую заданную ячейку, воспринимает в этой ячейке символ и в зависимости от своего текущего состояния (буква алфавита Р) выполняет действие. Действие МТ рассматривается как 3-ка следующих действий:

- Изменение буквы. Записать новую букву на место старой в ячейку. Если новая буква совпадает со старой, то считают, что МТ не меняет букву.
- Один из 3-х вариантов движения. Сдвинуться влево d_{-1}, сдвинуться в право d_{+1}, остаться на месте d_0. Это движение устройства вдоль ленты.

- Изменение состояния. Изменить состояние устройства, т. е. изменить букву внутреннего алфавита. Если новая буква состояния совпадает со старой, то считают, что МТ не меняет состояние.

Для МТ нужно определить начальное состояние, так как она в принципе работает бесконечно. Для этого определяют местоположение устройства в начальный момент времени (как правило устройство находится в левой части текста перед текстом или напротив 1-й буквы текста, но в некоторых важных задачах начальное положение устройства определяется особыми условиями).

Работа МТ может быть представлена следующим алгоритмом:

- МТ переходит в начальное состояние и на вход подается исходное слово (будем называть преобразуемый текст словом, даже если это предложение).
- МТ анализирует текущую ячейку и свое текущее состояние, вписывет новую букву (даже если реально буква и не меняется!) в ячейку, меняет свое состояние и наконец делает движение (на 1 клетку влево, вправо или остается на месте).
- Предшествующий шаг повторяется многократно до наступления события остановки.
- Остановка МТ.

Если работа МТ не прекращается, то считается что к данному слову МТ не применима. Если событие остановки наступает за конечное число шагов, то МТ применима к этому слову и дает некий результат.

Проблема остановки МТ

МТ способна работать бесконечно, поэтому особо актуальна проблема ее остановки (проблема конечности работы МТ для конкретной задачи). Выделение класса слов к которым применима данная программа МТ и наоборот построение программы МТ, которая применима к данному классу слов, а так же анализ применимости конкретной программы МТ к конкретному слову составляют основу задачи, которая называется в ТА проблемой останова МТ. Здесь применимость объединяет сразу конечность и результативность алгоритма, описываемого МТ.

О. Остановка МТ. МТ останавливается если в результате очередного действия МТ буква на ленте не изменилась, устройство осталось на месте, состояние МТ не изменилось. Очевидно это устойчивое неизменное положение МТ, которое может продолжаться до бесконечности.

Остановка считается результативной, если при этом необходимый результат получен (текст преобразован необходимым образом) или безрезультативной в других случаях. Такой результат зависит от программирования МТ и исходного данного. Если считать программу МТ частью этой машины, то для заданной МТ (с программой внутри) можно определить класс текстов, для которых МТ работает результативно. Говорят, что в этом случае МТ применима результативно к этому классу текстов.

14.Машина Тьюринга. Программа Машины Тьюринга.

Программа Машины Тьюринга

Если МТ находится в состоянии $P_к$ а символ в обозреваемой ячейке aj и после чтения символа выполняется движение d_n, то говорят что МТ выполняет команду (aj d_n $P_к$). Таким образом, набор троек (aj d_n $P_к$) полностью определяет работу МТ и называется системой команд МТ или программой МТ. Возможные команды МТ составляют конечное множество, так как внешний и внутренний алфавиты конечны, а команд движения всего 3 — это d_1 — движение направо, d_{-1} — движение налево и d_0 — недвижимость головки МТ. Других вариантов движения не предусмотрено (т. е. МТ не двигается на несколько ячеек сразу никогда). Для записи всей программы (множества команд) обычно используют табличную форму (которая наиболее компактна) следующего вида :

Ai	Pj	P0	P1	P2	...	Pn

A1	(Ai dj Pk)				
A2					
A3					
...					
Am					

Здесь строки соответствуют буквам внешнего алфавита Ai, столбцы соответствуют буквам внутреннего алфавита (состояний) Pj. В каждой клетке находятся тройки (команды) типа (Ai dj Pk) в которой есть буква внешнего алфавита Ai, буква внутреннего алфавита Pj и команда движения (одна из 3-х возможных) dj. Действие программы определяется следующими правилами:

1) Определяются буква в ячейке ленты напротив головки и текущее состояние МТ.
2) По таблице программы МТ находят подходящую клетку таблицы по соответствующим строкам и столбцам.
3) Выполняют команду в найденной клетке, заменив букву в ячейке на букву из клетки и изменив состояние МТ на состояние из клетки. Далее выполняют команду движения из клетки, возможно перейдя к другой клетке ленты.
4) Пункты 1-3 последовательно выполняются до выполнения остановки МТ.

Так как алфавиты конечны, то таблица тоже конечна. Кроме того, возможна ситуация, когда команда некоторой клетки ни когда не выполняется. Такие клетки можно не заполнять.

Замечание. Следует учитывать возможность реализации в МТ бесконечного алгоритма, который никогда не выполняет остановку. Обычно такая программа считается не верной или не применимой к текстам, которые заставляют ее выполнять бесконечный алгоритм. Поэтому программирование остановки МТ — особая задача, которую принято реализовать с помощью специального состояния для остановки. В это состояние МТ попадает только для остановки.

15.Машина Тьюринга. Программирование задач. Примеры.

Программирование задач для МТ. Примеры

Для разработки программ МТ необходимо построить таблицу МТ по следующим правилам:

- Фиксируется задание алгоритма и возможный внешний алфавит.
- Определяются варианты замены букв внешнего алфавита для реализации алгоритма. Для этого количество вариантов должно быть конечно. Как правило для каждого варианта нужно отдельное состояние МТ. Если число вариантов бесконечно, то МТ не может реализовать такой алгоритм.
- Как правило для анализа строки с неизвестной длиной необходимо в начале работы алгоритма пройти все ее буквы от начала до конца и начать работу с последней буквы. Последней буквой будет та, после которой будет пустая клетка (пробел). Таким образом пробел (обозначим нижним подчеркиванием _) тоже нужно добавить в внешний алфавит.
- Для остановки выделяется отдельное состояние.
- Последовательно заполняются (по столбцам) клетки таблицы. Последний столбец обычно состояние для остановки.

Рассмотрим построение программы МТ на примере. Пусть имеется двоичное число X к которому прибавляется число 1 (т. е. реализуется функция X+1).

Решение:

Внешний алфавит будет состоять из букв 0, 1, _.

Прибавление 1 приводит к следующим вариантам изменения строки числа:

- Если число заканчивается на 0, то эту букву нужно заменить на 1 и остановить МТ.
- Если число заканчивается на 1, то эту букву нужно заменить на 0 и сдвинуться влево.

- Если после сдвига налево перешли к букве 0, то эту букву нужно заменить на 1 и остановить МТ.
- Если после сдвига налево перешли к букве 1, то эту букву нужно заменить на 0 и сдвинуться влево.
- Если после сдвига налево перешли к пробелу, то записываем 1 и остановить МТ.

Для реализации программы МТ определим 3 состояния — P0 (поиск числа), P1 (пройти все ее буквы от начала до конца), P2(прибавление 1), P3(остановка).

Ai	Pj	P0	P1	P2	P3
_		_ d_{+1} P0	_ d_{-1} P2	1 d_0 P3	-
0		0 d_{+1} P1	1 d_0 P2	1 d_0 P3	-
1		1 d_{+1} P1	0 d_{+1} P1	0 d_{-1} P2	1 d_0 P2

Рассмотрим работу программы МТ на примере числа 111 (выделив цветом активную клетку):

P0	P0	P1	P1	P1	P2	P2	P2	P2	P3
_111_	_111_	_111_	_111_	_111_	_111_	_110_	_100_	_000_	1000_

В состоянии P0 находим число, в P1 проходим его, далее возвращаемся и в P2 меняем 1 на 0 двигаясь влево до начала цифры. Далее меняем пробел на 1 и переходим в P3 для остановки. В результате 111 переходит в 1000=111+1. Программа сработала правильно.

Другие примеры рассматриваются на практических занятиях.

Программирование Машины Тьюринга

Составить программу МТ для прибавления

Задача 1. X+1 в 2-й д/з X+2 в 5-й

Задача 2. X+6 в 8-й д/з X+B в 16-й

Задача 3. X+6 в 4-й д/з X+12 в 9-й

Составить программу МТ для умножения

Задача 4. X*2 в 3-й д/з X*2+1 в 2-й

Задача 5. X*2 в 8-й д/з X*2 в 16-й

Задача 6. 2*X+5 в 7-й д/з 2*X+8 в 9-й

Составить программу МТ для вычитания

Задача 7. X-6 в 7-й д/з 2*X-2 в 8-й

16.Машина Поста. Особенности машины Поста.

Машина Поста. Особенности машины Поста

Машина Поста (МП) — воображаемая алгоритмическая машина преобразующая текстовые константы. Таким образом МП — аналог МТ с некоторыми особенностями, связанными с представлением Поста о работе такой машины и ее программировании. Этот подход альтернатива МТ, но не получила такого широкого признания как МТ. Поэтому рассмотри МП в сравнении с МТ как анализ ее особенностей:

- Машина Поста устроена проще, чем машина Тьюринга, в том отношении, что ее элементарные действия проще, чем элементарные действия машины Тьюринга, и способы записи менее разнообразны, однако именно по этим причинам запись и переработка информации на машине Поста требует большего объема «памяти» в большего числа шагов, чем на машине Тьюринга.
- МП имеет такую же ленту и читающую головку как и в МТ. В каждой секции ленты может быть либо ничего не записано (такая секция называется пустой), либо записана метка X (тогда секция называется отмеченной). Таким образом, внешний алфавит ограничен 2 буквами — пробел и метка. Движения головки (каретки) для МП аналогичны МТ.
- Работа машины Поста состоит в том, что каретка передвигается вдоль ленты и печатает или стирает метки. Эта работа происходит по инструкции определенного вида, называе-

мой программой. В отличие от программы МТ эта программа представляется как последовательность пронумерованных команд 6 вариантов — движение вправо, движение влево, стирание метки, запись метки, ветвление с переходом к команде по ее номеру, команда остановки. Таким образом, программа МП более похожа на программу ЭВМ (в операциональном подходе). Возможность переходов с вариантами выбора определяет возможную сложность реализуемых алгоритмов. Пост доказал (теорема тезис Поста), что его машина имеет примерно тот же класс реализуемых алгоритмов что и у МТ.

Программой машины Поста будем называть конечный непустой (т. е. содержащий хотя бы одну команду) список команд машины Поста, обладающий следующими двумя свойствами:

1) На первом месте в этом списке стоит команда с номером 1, на втором месте (если оно есть) —команда с номером 2 и т. д.; вообще на k-м месте стоит команда с номером k.

2) Переход любой из команд списка возможен только на существующую (по номеру) другой или той же самой команды списка (более точно: для каждого перехода каждой команды списка найдется в. списке такая команда, номер которой равен рассматриваемой ссылке).

Для наглядности программы машины Поста записывают столбиком. Число команд программы называется длиной программы.

Чтобы машина Поста начала работать, надо задать, во-первых, некоторую программу, а во-вторых, некоторое ее (машины) состояние, т. е. как-то расставить метки по секциям ленты (в частности, можно все секции оставить пустыми) и поставить каретку против одной из секций. Секции можно пронумеровать. Как правило, мы будем предполагать, что в начальном (т. е. в задаваемом вначале) состоянии машины каретка ставится всегда против секции с номером (координатой) нуль. При таком соглашении начальное состояние машины полностью определено состоянием ленты.

Если теперь, задавшись программой и каким-либо начальным состоянием, пустить машину в ход, то осуществится один из следующих трех вариантов :

1) В ходе выполнения программы машина дойдет до выполнения невыполнимой команды (печатания метки в непустой секции или стирания метки в пустой секции); выполнение программы тогда прекращается, машина останавливается; происходит так называемая безрезультатная остановка.

2) В ходе выполнения программы машина дойдет до выполнения команды остановки; программа в этом случае считается выполненной, машина останавливается; происходит так называемая результативная остановка.

3) В ходе выполнения программы машина не дойдет до выполнения ни одной из команд, указанных в первых двух вариантах; выполнение программы при этом никогда не прекращается, машина никогда не останавливается; процесс работы машины происходит бесконечно.

Таким образом, МП в отличии от МТ не может печатать метку на месте другой или стирать пустую клетку.

Лекция №7. **Представление алгоритмов с помощью алгорифмов Маркова. Марковская подстановка. Этапы решения задач. Порядок действия алгорифма Маркова. Примеры алгорифмов Маркова. Представление алгоритмов с помощью вычислимых функций. Вычислимые функции. Разрешимые и перечислимые множества. Подходы к определению класса вычислимых функций.**

17.Алгорифмы Маркова. Принцип нормализации. Программирование задач. Примеры.

Представление алгоритмов с помощью алгорифмов Маркова

Еще одним альтернативным определением алгоритмической машины являются алгорифмы предложенные русским математиком Марковым (учитель знаменитого советского математика Колмогорова). Алгорифм — это точное воспроизведение произношения арабского слова, обозначающего алгоритм. Наше слова алгоритм уже имеет современное происхождение как перевод из английского языка. Марков в начале 20 века ссылался на исходное понятие и термин из арабского языка. В отличие от Тьюринга Марков не конкретизировал исполнителя

своих алгорифмов. Это просто задание некого действия с текстом, выполняемого путем замены одних текстов другими. Это роднит Алгорифмы Маркова (АМ) с современными грамматиками, где продукционные правила похожи на марковские подстановки.

О.АМ — последовательность марковских подстановок, в которой местоположение подстановки принципиально важно для результата. Подстановки не нумеруются, но их порядок однозначно определен.

По аналогии с Тьюрингом Марков определил свой **Принцип нормализации** — тезис о выделении класса вычислимых с помощью АМ алгоритмов.

Принцип нормализации — Любой реально выполнимый алгоритм имеет реализующий его АМ.

Марковская подстановка

О.Марковская подстановка (МкП) — пара текстов, связанных между собой. Первая часть МкП должна быть заменена на вторую часть МкП при работе АМ. Записываем МкП в виде A->B. Действие МкП состоит в том, что в исходном тексте ищем подстроку A и заменяем ее на B. Таким образом, АМ — некая функция, на вход которой подается текст, а на выходе получают другой текст. Такие функции называют словарные, их аргументы слова, результат тоже слово (как конечный по размерам текст).

О.МкП называется конечной, если после ее завершения работа АМ прекращается. Для выделения конечных МкП указывают после стрелки точку. Таким образом, A->.B — конечная подстановка.

Для понимания действия АМ нужно выполнения следующих правил:

- МкП выполняются строго по порядку их расположения, который не может быть изменен.
- Если очередная МкП не может быть выполнена, то переходят к следующей по порядку МкП.
- После завершения действия МкП, АМ снова начинает попытку выполнения МкП с начала (т. е. с первой по порядку МкП).

Завершение АМ возможно только в 2-х вариантах. Первый вариант — ни одна МкП не может быть выполнена. Второй вариант — выполнена конечная МкП.

По аналогии с МТ можно ввести понятие применимости АМ.

Этапы решения задач. Порядок действия алгорифма Маркова

Построение алгорифма Маркова должно опираться на определенные этапы:

- Выделение класса алгоритмов (словарных функций) к которым будет применим АМ.
- Определение конечной подстановки или нескольких конечных подстановок (при необходимости). Конечная подстановка должна соответствовать завершению действия АМ как представляется разработчиком АМ.
- Рассмотрение альтернативных вариантов слов которые необходимо обрабатывать и установка порядка их обработки.
- Построение МкП для обработки всех альтернатив.
- Определение последовательности МкП для обработки всех альтернатив.

Это особенно важно, так как АМ постоянно начинает действия подстановок с начала АМ. Поэтому порядок МкП в АМ нужно правильно расставить и тщательно проанализировать на примерах из каждой альтернативы. Есть главное правило — чем реже выполняется МкП тем раньше ее нужно размещать в АМ. Для выставления порядка важен так же пробел, потому что наличие пробела в конце текста говорит о том что это конец слова, а помещение его в начало показывает, что этот текст находится в начале слова.

Характерным для АМ является использование добавочной буквы или текста, который заведомо отсутствует в обрабатываемых словах. Так же как в МТ эта буква пробегает при необходимости все слово либо с начала до завершающего пробела, либо с конца до начального пробела. При этом можно запланировать остановку или остановки на любом этапе перемещения. Это дает возможность выполнения нескольких одинаковых подстановок в одном АМ, либо разделять порядко МкП.

Примеры алгорифмов Маркова

Рассмотрим построение алгорифма Маркова на примере. Пусть имеется восьмеричное число X к которому прибавляется число 1 (т. е. реализуется функция X+1).

В результате действия АМ необходимо получить из записи числа X запись числа X+1. Если X завершается на 0,1,2,3,4,5,6, то в X+1 нужно заменить последнюю букву/цифру и и остановить работу. Если X завершается на 7, то заменим 7 на x0 и используем новую букву x для прибавления 1 к другим разрядам. Результат АМ:

_x->.1
0x->.1
1x->.2
2x->.3
3x->.4
4x->.5
5x->.6
6x->.7
7x->x0
0_->.1
1_->.2
2_->.3
3_->.4
4_->.5
5_->.6
6_->.7
7_->x0

Другие примеры рассмотрены на практических занятиях.

Тема. Алгорифмы Маркова

Задача 1. Дан прямой код числа, построить обратный код д/з дополнительный

Задача 2. Дано 16-е число, построить двоичный код д/з восьмеричный

Задача 3. Дано 8-е число, построить прибавление 1 к числу д/з прибавление 3 из числа

Задача 4. Дано 4-е число, построить вычитание 3 из числа д/з вычитание 3 из 5-го числа

Лекция №8. **Рекурсивные функции. Базовые рекурсивные функции. Операторы суперпозиции и примитивной рекурсии. Определение рекурсивных функций по Черчу. Общерекурсивные функции. Оператор построения по первому нулю (оператор минимизации). Правило минимизации. Тезисы Черча и Клини. Примеры построения рекурсивных функций. Эквивалентность описанных теорий.**

19.Понятие вычислимой и рекурсивной функции. Базовые рекурсивные функции. Общерекурсивные функции.

Представление алгоритмов с помощью вычислимых функций

О. Вычислимой функцией называют функцию имеющую алгоритм вычисления.

Это определение фактически распространяет выделение класса алгоритмов на максимально широкое понятие. Если задуматься, то любая функция матанализа мыслится в конечном итоге вычислимой. С другой стороны такое выделение алгоритмов позволяет использовать классический математический подход к построению вычислимых функций, а следовательно к построению алгоритмов. Любая функция построенная из вычислимых стандартным способом может быть причислена к вычислимым. Остается только определить набор таких стандартных действий. Этим подходом в ТА активно занимались Черч и Клини, последовательно расширяя класс вычислимых функций и введя понятие рекурсивной функции. До настоящего времени этот подход плодотворен, ученые строят все новые и новые классы вычислимых функций усложняя

набор стандартных способов или доказывая возможность построения таких функций стандартным нобором.

Вычислимые функции. Разрешимые и перечислимые множества

Любая функция в математике определяется как отображение 2-х множеств — множества аргументов и множества значений.

Подходы к определению класса вычислимых функций

О.Вычислимой функцией называют функцию имеющую алгоритм вычисления.

Анализ этого определения показывает, что вычислимыми могут быть только целочисленные функции, у которых аргумент и результат — целые числа (в крайнем случае рациональные). Так как действительные числа с точки зрения алгоритмов требуют бесконечных действий даже для их записи. В ЭВМ это препятствие обходят округлением, когда действительное число заменяют парой целых чисел — мантисой и порядком. Поэтому далее рассмотрим только целочисленные функции. По смыслу понятие вычислимых функций ничем не хуже понятия функций реализуемых МТ, МП, АМ. Существую теоремы связывающие классы этих функций. В частности доказывается практическая эквивалентность этих вариантов описания, то есть любая функция реализуемая МТ может быть реализована с помощью МП, АМ и фактически являктся вычислимой. В теории алгоритмов при этом принято выражать класс вычислимых функций через классы рекурсивных функций.

Рекурсивные функции. Базовые рекурсивные функции. Операторы суперпозиции и примитивной рекурсии

Первым классом, относящимся к рекурсивным функциям является класс базовых рекурсивных функций (БРФ). К БРФ относятся :

1) $z(X) = 0$ — нуль функция, функция тождественно равная нулю;

2) $N(X) = x_к + 1$ — функция следования:

3) $I_k(x_1,...,xn) = xk$ функция равная одному из аргументов (функция выбора аргумента, проектирующая функция).

Здесь $X=(x_1,...,xn)$ — вектор неизвестных.

Вычислимость БРФ очевидна. Если заданы значения всех элементов вектора неизвестных, то любую из 3 функций можно вычислить (т. е. алгоритмически определить ее значение).

Из БРФ могут быть образованы другие классы вычислимых функций при помощи специальных алгоритмически определенных операций.

О.Суперпозиция функций A и B — замена одного из аргументов функции A на значение функции B. Алгоритм такой операции очевиден. Записывается это как C(A,B).

Пример : Пусть $A(x)=x+1$, $B(x)=x+1$, тогда функция $P(x)=C(A,B)=x+2$ — тоже алгоритмически вычислима через суперпозицию.

О. Говорят, что функция f зависящие от к+1 переменной получаются применением оператора примитивной рекурсии R к функциям g от к переменных и h от к+1 переменных если выполнено условие f=R(g,h):

1)$f(x_1,...,xn,0) = g(x_1,...,xn)$

2) $f(x_1,...,xn,k+1)=h(x_1,...,xn,k,f(x_1,...,xn,k))$

В данном случае первое выражение позволяет вычислить значение f через g при последней переменной равной нулю (начальное условие для вычисления реккурентной последовательности), а второе выражение дает возможность последовательного вычисления f

через суперпозицию h и f с постоянным увеличением последнего аргумента на 1. Так как все переменные цклочисленные, то на некотором шаге мы получим искомое значение. С точки зрения программирования рекурсия ведется по последней переменной и выражает уже известный способ математической индукции.

О.Функция f называется **примитивно-рекурсивной** если она может быть получена из базовых рекурсивных функций применением конечного числа операторов суперпозиции и примитивной рекурсии. Очевидно, что такие функции можно строить и из других **примитивно-рекурсивных** функций применением конечного числа операторов суперпозиции и примитивной рекурсии.

Класс примитивно-рекурсивных функций часто называют **общерекурсивными** в отличие от частично-рекурсивных. Это связано с тем, что такие функции можно вычислить для любых значений переменных, а вот для частично-рекурсивных функций это не справедливо.

20.Тезисы Черча и Клини. Частично-рекурсивные функции. Операция минимизации.

Общерекурсивные функции. Оператор построения по первому нулю (оператор минимизации). Правило минимизации. Тезисы Черча и Клини. Примеры построения рекурсивных функций

О.Функция f называется **примитивно-рекурсивной** если она может быть получена из базовых рекурсивных функций применением конечного числа операторов суперпозиции и примитивной рекурсии. Очевидно, что такие функции можно строить и из других **примитивно-рекурсивных** функций применением конечного числа операторов суперпозиции и примитивной рекурсии.

Класс примитивно-рекурсивных функций часто называют общерекурсивными в отличие от частично-рекурсивных. Это связано с тем, что такие функции можно вычислить для любых значений переменных, а вот для частично-рекурсивных функций это не справедливо.

Тезис Клини : всякая вычислимая всюду функция является общерекурсивной

Тезис Черча : всякая вычислимая функция является частично рекурсивной

Эти тезисы связывают понятие вычислимых функций с рекурсивными. По тезису Клини, если функция всюду вычислима, то можно построить последовательность конечного числа операций суперпозиции и примитивной рекурсии вычисления этих функций из базовых. Это означает, что алгоритмы их вычисления формализуются аналогично метаматематике с правилами вывода.

Тезис Черча распространяет этот метод на функции определенные (вычислимые) не везде, не для всяких значений переменных. Для этого вводится 3 операция — минимизация или вычисление первого нуля.

О.Функция f(x1,…,xn,z) называется результатом **применения оператора минимизации** (μ-оператор) к функции P(x1,…,xn,y), если функция f равна такому значению y = {0,1…z} при котором значение функции P(x1,…,xn,y) = 0 а для t<y значение предиката P(x1,…,xn,t)<>0. Поэтому y часто называют первым нулем, так как он корень уравнения P(x1,…,xn,y) = 0, причем минимальный среди возможных. Выбор минимального корня обусловлен тем, что если последовательно увеличивать t от 0, то при наличии корней мы всегда найдем однозначно именно минимальный корень. Алгоритм нахождения других корней уже не будет однозначен. С другой стороны корень может совсем отсутствовать, поэтому f(x1,…,xn,z) при некоторых значениях набора (x1,…,xn) не будет определена, такие частично определенные функции и называют **частично рекурсивными.**

О. Частично рекурсивными функциями являются функции полученные с помощью применения оператора минимизации.

Теперь понятно, что тезис Черча опираясь на **частично рекурсивные функции** расширяет класс вычислимых функций на случай, когда функция вычисляется не при любых значениях аргументов.

Математический аппарат рекурсивных функций позволяет анализировать, доказывать существования и оптимизировать алгоритмы вычисления и решения различных задач.

Лекция №9. **Естественные и формальные языки. Формальный язык, алфавит, буква, слово. Символьные цепочки и их свойства. Способы задания языков.**
Лекция №10. **Понятие грамматики языка. Форма Бэкуса-Наура и ее использование. Примеры. Рекурсивность в правилах грамматики. Методы описания грамматик. Классификация языков по Хомскому.**

11. Формальные языки и их построение

Естественные и формальные языки

Языки принято делить на естественные (обычные для человека — русский, английсий языки) и формальные (например языки программирования, язык формальной математики и матлогики). Их главное различие — противоречивость (не однозначность конструкций) естественных языков (омонимы, антонимы, многозначность смысла и т. д.). Формальные языки наоборот должны иметь однозначно описываемые и понимаемые конструкции.

Естественными являются языки человеческого общения, сложившиеся исторически при естественном развитии человеческого мышления (русский, английский, французский, греческий, хинди и т. д.). Для их применения необходима интеллектуальная деятельность, то есть это языки уровня искусственного интеллекта, для работы с ЭВМ и описания работы автоматов желательно иметь языки более просто описываемые и формализуемые. Так стали создаваться формальные языки (язык матлогики, алгебры, языки программирования и т. д.).

Элементы теории формальных языков и грамматик

Теория формальных языков важна для информатики прежде всего с точки зрения осуществления трансляции. В любой системе программирования необходимо перекодирование из языка программирования в язык кодов ЭВМ. Легко транслируются только языки правильной (формальной) структуры, в которой (в отличие от нашего естественного языка) синтаксис и семантика строго фиксированы. Один из основных механизмов определения языков – грамматики. Грамматики – правила построения синтаксиса (а в некоторых случаях и семантики) конструкций языка. В теории формальных языков введены базовые понятия и даны определения, связанные с одним из основных механизмов определения языков - грамматиками, созданы системы классификации грамматик. Далее мы рассмотрим наиболее распространенную классификацию грамматик по Хомскому. Выделим классы контекстно-свободных грамматик и, в частности, их важному подклассу - регулярным грамматикам. Грамматики этих классов широко используются при трансляции языков программирования. Наиболее простым и важным примеров реализации грамматик на практике являются нотации Бекуса-Наура.

Способы задания языков

Известно несколько различных способов описания формальных языков. Один из вариантов использует порождающие грамматики. Именно этот способ описания языков рассмотрим подробнее.

Формальный язык может быть определён:

- Простым перечислением слов, входящих в данный язык (словарь). Этот способ, в основном, применим для определения конечных языков и языков простой структуры.
- Словами, порождёнными некоторой формальной грамматикой.
- Словами, порождёнными регулярным выражением.
- Словами, распознаваемыми некоторым конечным автоматом (например машина Тьюринга).
- Словами, порождёнными формами Бекуса-Наура.

Формальная грамматика или просто **грамматика** в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита. Различают *порождающие* и *распознающие* (или *аналитические*) грамматики — первые задают правила, с помощью которых можно построить любое слово языка, а вторые позволяют по данному слову определить, входит ли оно в язык или нет.

Регуля́рные выраже́ния — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов. Для поиска используется строка-образец («шаблон» или «маска»), состоящая из символов и метасимволов и задающая правило поиска. Для манипуляций с текстом дополнительно задаётся строка замены, которая также может содержать в себе специальные символы. Регулярные выражения состоят из строковых констант и операторов, которые определяют множества строк и множества операций на них соответственно.

Многие современные языки программирования имеют встроенную поддержку регулярных выражений. Среди них ActionScript, Perl, Java, PHP, JvaScript, Python, C++(с 2011 года), Delphi и другие. Регулярные выражения используются текстовыми редакторами для поиска и подстановки текста, обработки текстовой информации по шаблонам.

Построение порождающей грамматики.

Словами языка, заданного такой грамматикой, являются все последовательности терминальных символов, выводимые (порождаемые) из начального нетерминала по правилам вывода. Чтобы задать грамматику, требуется задать алфавиты терминальных и нетерминальных символов, набор продукционных правил вывода других конструкций, а также выделить в множестве нетерминальных символов некий начальный нетерминальный символ.

Итак, грамматика определяется набором:

- Σ — набор (алфавит) терминальных символов
- N — набор (алфавит) нетерминальных символов
- P — набор правил вида типа «левая часть» → «правая часть», где:
    - «левая часть» — непустая последовательность терминалов и нетерминалов, содержащая хотя бы один нетерминал
    - «правая часть» — любая последовательность терминалов и нетерминалов
- S — стартовый (или начальный) символ грамматики из набора нетерминалов.

Выводом новой конструкции или сущности будем называть последовательность строк, состоящих из терминалов и нетерминалов, где первой идет строка, состоящая из одного стартового нетерминала, а каждая последующая строка получена из предыдущей путём замены некоторой подстроки по одному (любому) из правил вывода. Конечной строкой является строка, полностью состоящая из терминалов, и следовательно являющаяся словом языка. Существование вывода для некоторого слова является критерием его принадлежности к языку, определяемому данной грамматикой.

Классификация языков

Формальные языки классифицируются в соответствии с типами грамматик, которыми они задаются. Однако, один и тот же язык может быть задан разными грамматиками, относящимися к разным типам. В таком случае, считается, что язык относится к наиболее простому из них. Так, язык, описанный грамматикой с фразовой структурой, контекстно-зависимой и контекстно-свободной грамматиками, будет контекстно-свободным. Так же, как и для грамматик, сложность языка определяется его типом. Наиболее сложные — языки с фразовой структурой (сюда можно отнести естественные языки), далее — КЗ-языки, КС-языки и самые простые — регулярные языки.

22.Грамматика формальных языков.

Формальная грамматика или просто **грамматика** в теории формальных языков — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита. Различают *порождающие* и *распознающие* (или *аналитические*) грамматики — первые задают правила, с помощью которых можно построить любое слово языка, а вторые позволяют по данному слову определить, входит ли оно в язык или нет.

Построение порождающей грамматики.

Словами языка, заданного такой грамматикой, являются все последовательности терминальных символов, выводимые (порождаемые) из начального нетерминала по правилам вывода. Чтобы задать грамматику, требуется задать алфавиты терминальных и нетерминальных символов, набор продукционных правил вывода других конструкций, а также выделить в множестве нетерминальных символов некий начальный нетерминальный символ.

Итак, грамматика определяется набором:

- Σ — набор (алфавит) терминальных символов
- N — набор (алфавит) нетерминальных символов
- P — набор правил вида типа «левая часть» → «правая часть», где:
  - «левая часть» — непустая последовательность терминалов и нетерминалов, содержащая хотя бы один нетерминал
  - «правая часть» — любая последовательность терминалов и нетерминалов
- S — стартовый (или начальный) символ грамматики из набора нетерминалов.

Выводом новой конструкции или сущности будем называть последовательность строк, состоящих из терминалов и нетерминалов, где первой идет строка, состоящая из одного стартового нетерминала, а каждая последующая строка получена из предыдущей путём замены некоторой подстроки по одному (любому) из правил вывода. Конечной строкой является строка, полностью состоящая из терминалов, и следовательно являющаяся словом языка. Существование вывода для некоторого слова является критерием его принадлежности к языку, определяемому данной грамматикой.

Формальный язык, алфавит, буква, слово

Основные понятия и определения

Определение: формальный язык должен быть построен так, чтобы его синтаксис и семантика имели бы строгие однозначные правила построения конструкций и их анализа.

Определение: синтаксис — правила построения конструкций языка, семантика — правила анализа смысла (семантического) конструкций языка.

Определение: *алфавит* - это конечное множество символов. Предполагается, что термин "символ" имеет достаточно ясный интуитивный смысл и не нуждается в дальнейшем уточнении. При этом часто для пояснения понятия символ определяют терминальные и нетерминальные символы.

Терминальный символ — объект, непосредственно присутствующий в словах языка и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов ASCII — латинские буквы, цифры и спецсимволы.

Нетерминальный символ — объект, обозначающий какую-либо *сущность* языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

Символьные цепочки и их свойства

Определение: *цепочкой символов в алфавите V* называется любая конечная последовательность символов этого алфавита.

Определение: цепочка, которая не содержит ни одного символа, называется *пустой цепочкой*. Для ее обозначения будем использовать символ ε.

Более формально цепочка символов в алфавите V определяется следующим образом:

Определение: если α и β - цепочки, то цепочка αβ называется *конкатенацией (*или *сцеплением)* цепочек α и β. Например, если α = ab и β = cd, то αβ = abcd. Для любой цепочки α всегда αε = εα = α.

Определение: *обращением (*или *реверсом)* цепочки α называется цепочка, символы которой записаны в обратном порядке.

Обращение цепочки α будем обозначать α^R. Например, если α = abcdef, то α^R = fedcba. Для пустой цепочки: $\varepsilon = \varepsilon^R$.

Определение: n-ой степенью цепочки α (будем обозначать α^n) называется конкатенация n цепочек α.

$\alpha^0 = \varepsilon$; $\alpha^n = \alpha\alpha^{n-1} = \alpha^{n-1}\alpha$.

Определение: *длина цепочки* - это число составляющих ее символов.

Например, если α = abcdefg, то длина α равна 7.

Длину цепочки α будем обозначать | α |. Длина ε равна 0.

Определение: *язык* в алфавите V - это подмножество цепочек конечной длины в этом алфавите.

Определение: обозначим через V^* множество, содержащее все цепочки в алфавите V, включая пустую цепочку ε.

Например, если V={0,1}, то V^* = {ε, 0, 1, 00, 11, 01, 10, 000, 001, 011, ...}.

Определение: обозначим через V^+ множество, содержащее все цепочки в алфавите V, исключая пустую цепочку ε.

Следовательно, $V^* = V^+ + \{\varepsilon\}$.

Ясно, что каждый язык в алфавите V является подмножеством множества V^*.

Определение: *декартовым произведением* A × B множеств A и B называется множество пар (a,b), где a принадлежит A, b принадлежит B.

Описание грамматики языка

Определение: *порождающая грамматика* G - это четверка (VT, VN, P, S), где

VT - алфавит *терминальных символов (терминалов)*,

VN - алфавит *нетерминальных символов (нетерминалов)*, не пересекающийся с VT,

P - конечное подмножество множества $(VT \setminus VN)^+ \times (VT \setminus VN)^*$; элемент (α, β) множества P называется продукционным *правилом вывода* и записывается в виде $\alpha \rightarrow \beta$,

S - *начальный символ (цель)* грамматики, $S \setminus VN$.

Для записи правил вывода с одинаковыми левыми частями

$\alpha \rightarrow \beta_1$ $\alpha \rightarrow \beta_2$ ... $\alpha \rightarrow \beta_n$ будем пользоваться сокращенной записью $\alpha \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$.

Каждое β_i, i= 1, 2, ... ,n , будем называть *альтернативой* правила вывода из цепочки α.

Определение: цепочка β *выводима* из цепочки α в грамматике $G = (VT, VN, P, S)$, если существуют цепочки $\gamma_0, \gamma_1, ..., \gamma_n (n \geq 0)$, такие, что $\alpha = \gamma_0 \rightarrow \gamma_1 \rightarrow ... \rightarrow \gamma_n = \beta$.

Определение: последовательность $\gamma_0, \gamma_1, ..., \gamma_n$ называется *выводом длины n*.

Определение: *языком, порождаемым грамматикой* $G = (VT, VN, P, S)$, называется множество всех цепочек в алфавите VT, которые выводимы из S с помощью P.

Рекурсивность в правилах грамматики. Методы описания грамматик

Использование нотаций Бекуса для описания продукций в грамматиках показывает главный принцип построения грамматики — рекурсивность, т.е формулировка правил через самих себя. Такие конструкции (которые принято называть генераторами форм) очень компактны и дают широкие возможности для описания различных классов конструкций языка. Фактически это компактная форма записи алгоритмов построения (генерации) новых форм. Это так называемая продукционная форма представления знаний (теория представления знаний ИИ), которую принято относить к алгоритмической форме представления знаний (в отличие от описательной декларативной формы). Таким образом грамматики компактно формируются из базовых множеств (которые как правило достаточно компактны) и достаточно небольшого количества продукционных правил рекурсивной формы.

Другой способ описания правил грамматик — использование автоматов, например Машины Тьюринга. В этом случае программа МТ выступает в качестве генератора форм грамматики.

18.Нотации Бекуса-Наура. Построение нотаций. Примеры.

Форма Бэкуса — Наура (сокр. БНФ, Бэкуса — Наура форма) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик. Существует расширенная форма Бэкуса — Наура, отличающаяся лишь более ёмкими конструкциями.

Используется для описания синтаксиса языков программирования, данных, протоколов и т. д. (причём как грамматики, так и регулярной лексики, поскольку регулярные грамматики являются подмножеством контекстно-свободных).

Форма Бэкуса-Наура и ее использование

Системы продукций, то есть правил вида: “Если A, то B”, задающих элементарные шаги преобразований и умозаключений могут служить основой для представления в грамматиках языков. Там продукционные правила строятся как синтаксические генераторы новых форм. Классическим примером продукционных правил являются нотации Бекуса-Наура, разработанные

в 60-е годы для описания алгоритмов работы синтаксических анализаторов. Практическим применением синтаксических анализаторов стало создание трансляторов для языков программирования (например Алгол).

Правило нотации имеет простой вид:

<A>::=C_1| C_2| C_3| C_4| C_5| C_6…. Здесь А – новая форма, знак ::= означает «по определению есть», знак | - означает «или», C_i – варианты. Таким образом, новая форма выражается через описанные ранее формы или текстовые константы. При этом есть возможность разветвления логики логическим оператором «или». Вариант оператора «и» здесь заменяется негласным правилом образования новых форм путем соединения нескольких старых. Таким образом, варианты C_i в нотации могут быть текстовой константой, уже описанной формой или соединением текстовых констант и описанных форм в любом порядке и количестве. Таким образом, продукционные правила служат для генерирования новых форм на основе констант и других описанных форм. Этот механизм очень удачен для описания грамматик различных языков и является возможной формой для описания таких структур как знания (область представления знаний искусственного интеллекта).

Рекурсивность в нотациях Бекуса-Наура

Использование нотаций Бекуса демонстрирует главный принцип построения новых синтаксических форм — рекурсивность, т.е формулировка правил через самих себя. Такие конструкции (которые принято называть генераторами форм) очень компактны и дают широкие возможности для описания различных классов конструкций языка. Фактически это компактная форма записи алгоритмов построения (генерации) новых форм. Это так называемая продукционная форма представления знаний (теория представления знаний ИИ), которую принято относить к алгоритмической форме представления знаний (в отличие от описательной декларативной формы). Таким образом нотации компактно формируются из базовых множеств (которые как правило достаточно компактны) и достаточно небольшого количества продукционных правил рекурсивной формы. Кроме того, нотации легко строятся в обычных процедурных языках программирования. Например в Паскале — нотация это функция, аргументом которой является анализируемая строка, а результатом — логическое значение (истина, если строка идентифицируется нотацией и ложь, если строка не соответствует правилу нотации). Альтернативы правила легко реализуются оператором паскаля case, а рекурсия правила определяется рекурсивным вызовом этой же функции. Поэтому нотации были выбраны в качестве основы программирования трансляторов, где наиболее сложная часть алгоритма — разбор текста программы. Описывая анализируемые конструкции программы с помощью нотаций, далее легко записать алгоритм этого анализа уже на языке программирования.

Пример. Нотации Бекуса для описания натуральных чисел:

<Ch>::=1|2|3|4|5|6|7|8|9

<NCh>::=<Ch>|<NCh>0|<NCh><Ch>

Другие примеры рассмотрены на практических занятиях.

Тема: Нотации Бекуса

Задача 1. Построить нотацию натуральных и целых чисел д/з рациональные

Задача 2. Построить нотацию действительных чисел д/з комплексных

Задача 3. Построить нотацию идентификатора переменной д/з заголовок процедуры

Задача 4. Построить нотацию для определения номера автомобиля д/з почтового адреса

Задача 5. Построить нотацию для определения четных чисел в 10-й системе д/з делящихся на 5

Задача 6. Построить нотацию для определения чисел кратных 4 в 8-й системе д/з делящихся на 2

12.Классификация формальных языков по Хомскому.

Классификация языков по Хомскому

Иерархия Хомского — классификация формальных языков и формальных грамматик, согласно которой они делятся на 4 типа по их условной сложности. Предложена профессором Массачусетского технологического института, лингвистом Ноамом Хомским.

(грамматики классифицируются по виду их правил вывода)

Классификация грамматик

Согласно Хомскому, формальные грамматики можно разделить на четыре типа. Для отнесения грамматики к тому или иному типу необходимо соответствие *всех* её правил (продукций) некоторым схемам.

ТИП 0: Грамматика G = (VT, VN, P, S) называется *грамматикой типа 0*, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики), описывает языки с фразовой струтурой. Каждый тип грамматики описывает как частный случай грамматики с большим номером в классификации. Поэтому Языки типа 0 сдержат все типы языков, языки типа 1 содержат как подмножества типы 2 и 3 и т. д.

ТИП 1: Грамматика G = (VT, VN, P, S) называется *неукорачивающей грамматикой*, если каждое правило из P имеет вид $\alpha \rightarrow \beta$, и $| \alpha | \leq | \beta |$.

Грамматика G = (VT, VN, P, S) называется *контекстно-зависимой (КЗ)*, если каждое правило из P имеет вид $\alpha \rightarrow \beta$, где $\alpha = \xi_1 A \xi_2$; $\beta = \xi_1 \gamma \xi_2$;. *Грамматику типа 1* можно определить как неукорачивающую либо как контекстно-зависимую. Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых неукорачивающими грамматиками, совпадает с множеством языков, порождаемых КЗ-грамматиками.

ТИП 2: Грамматика G = (VT, VN, P, S) называется *контекстно-свободной (КС)*, если каждое правило из P имеет вид $A \rightarrow \beta$. *Грамматику типа 2* можно определить как контекстно-свободную либо как укорачивающую контекстно-свободную. Возможность выбора обусловлена тем, что для каждой УКС-грамматики существует почти эквивалентная КС-грамматика.

ТИП 3: Грамматика G = (VT, VN, P, S) называется *праволинейной*, если каждое правило из P имеет вид $A \rightarrow tB$ либо $A \rightarrow t$. Грамматика G = (VT, VN, P, S) называется *леволинейной*, если каждое правило из P имеет вид $A \rightarrow Bt$ либо $A \rightarrow t$. *Грамматику типа 3 (регулярную, Р-грамматику)* можно определить как праволинейную либо как леволинейную. Выбор определения не влияет на множество языков, порождаемых грамматиками этого класса, поскольку доказано, что множество языков, порождаемых праволинейными грамматиками, совпадает с множеством языков, порождаемых леволинейными грамматиками.

Соотношения между типами грамматик:

(1) любая регулярная грамматика является КС-грамматикой;

(2) любая регулярная грамматика является УКС-грамматикой;

(3) любая КС-грамматика является КЗ-грамматикой;

(4) любая КС-грамматика является неукорачивающей грамматикой;

(5) любая КЗ-грамматика является грамматикой типа 0.

(6) любая неукорачивающая грамматика является грамматикой типа 0.

Замечание: УКС-грамматика, содержащая правила вида $A \rightarrow \varepsilon$, не является КЗ-грамматикой и не является неукорачивающей грамматикой.

Определение: язык L(G) является *языком типа k*, если его можно описать грамматикой типа k.

Соотношения между типами языков:

(1) каждый регулярный язык является КС-языком, но существуют КС-языки, которые не являются регулярными (например, $L = \{a^n b^n \mid n>0\}$).

(2) каждый КС-язык является КЗ-языком, но существуют КЗ-языки, которые не являются КС-языками (например, $L = \{a^n b^n c^n \mid n>0\}$).

(3) каждый КЗ-язык является языком типа 0.

Замечание: УКС-язык, содержащий пустую цепочку, не является КЗ-языком.

Замечание: следует подчеркнуть, что если язык задан грамматикой типа k, то это не значит, что не существует грамматики типа k' (k'>k), описывающей тот же язык. Поэтому, когда говорят о языке типа k, обычно имеют в виду максимально возможный номер k.

Дополнительно по классификации грамматик и языков по Хомскому

Тип 0 — неограниченные

Грамматика с фразовой структурой G — это алгебраическая структура, упорядоченная четвёрка (V_T, V_N, P, S),

К типу 0 по классификации Хомского относятся неограниченные грамматики — грамматики с фразовой структурой, то есть все без исключения формальные грамматики. Правила можно записать в виде: $\alpha \rightarrow \beta$

где α — любая непустая цепочка, содержащая хотя бы один нетерминальный символ, а β — любая цепочка символов из алфавита. Практического применения в силу своей сложности такие грамматики не имеют.

Тип 1 — контекстно-зависимые

К этому типу относятся контекстно-зависимые (КЗ) грамматики и неукорачивающие грамматики. Для грамматики $G(V_T, V_N, P, S)$, все правила имеют вид:

- $\alpha A \beta \rightarrow$ где $\alpha, \beta \in V^*, \gamma \in V^+, A \in V_N$. Такие грамматики относят к контекстно-зависимым.
- $\alpha \rightarrow \beta$, где $\alpha, \beta \in V^+, 1 \leq |\alpha| \leq |\beta|$. Такие грамматики относят к неукорачивающим.

Эти классы грамматик эквивалентны. Могут использоваться при анализе текстов на естественных языках, однако при построении компиляторов практически не используются в силу своей сложности. Для контекстно-зависимых грамматик доказано утверждение: по некоторому алгоритму за конечное число шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет.

Тип 2 — контекстно-свободные

К этому типу относятся контекстно-свободные (КС) грамматики. Для грамматики $G(V_T, V_N, P, S)$, $V = V_T \cup V_N$ все правила имеют вид:

- $A \rightarrow \beta$, где $\beta \in V^+$ (для неукорачивающих КС-грамматик) или $\beta \in V^*$ (для укорачивающих), $A \in V_N$. То есть грамматика допускает появление в левой части правила только нетерминального символа.

КС-грамматики широко применяются для описания синтаксиса компьютерных языков.

Тип 3 — регулярные

К третьему типу относятся регулярные грамматики (автоматные) — самые простые из формальных грамматик. Они являются контекстно-свободными, но с ограниченными возможностями. Все регулярные грамматики могут быть разделены на два эквивалентных класса, которые для грамматики вида III будут иметь правила следующего вида:

- $A \to B\,\gamma$ или $A \to \gamma$, где $\gamma \in V\,T\,*$, $A, B \in V\,N$ (для леволинейных грамматик).
- $A \to \gamma\,B$; или $A \to \gamma$, где $\gamma \in V\,T\,*$, $A, B \in V\,N$ (для праволинейных грамматик).

Регулярные грамматики применяются для описания простейших конструкций: идентификаторов, строк, констант, а также языков ассемблера, командных процессоров и др.

Классификация языков

Формальные языки классифицируются в соответствии с типами грамматик, которыми они задаются. Однако, один и тот же язык может быть задан разными грамматиками, относящимися к разным типам. В таком случае, считается, что язык относится к наиболее простому из них. Так, язык, описанный грамматикой с фразовой структурой, контекстно-зависимой и контекстно-свободной грамматиками, будет контекстно-свободным. Так же, как и для грамматик, сложность языка определяется его типом. Наиболее сложные — языки с фразовой структурой (сюда можно отнести естественные языки), далее — КЗ-языки, КС-языки и самые простые — регулярные языки.