Математическая логика и теория алгоритмов

IКурс

1. Возникновение и развитие дисциплины "Теория алгоритмов". Математическое понятие алгоритма.
2. Алфавитный оператор. Общность понятия, способы задания.   
   Примеры простых и кодирующих операторов.
3. Многозначные и однозначные алфавитные операторы.
4. Соотношение между АО и алгоритмами. Стохастические и самоизменяющиеся алгоритмы.
5. Свойства алгоритмов. Понятие алгоритмической системы.
6. Понятие абстрактного вычислителя. Машина Поста.
7. Понятие программы машины Поста. Диаграмма и схема Поста.
8. Понятие абстрактного вычислителя. Машина Тьюринга.
9. Тезис Тьюринга. Полнота по Тьюрингу. Вычислимые функции по Тьюрингу.
10. Универсальная машина Тьюринга. Пример работы.
11. Модификации машины Тьюринга. Примеры работы.
12. Композиции машины Тьюринга: произведение, возведение в степень, итерация. Пример.
13. Понятие абстрактного вычислителя. Машина с бесконечными регистрами.
14. МБР-вычислимые функции. Элементарные вычислимые функции.
15. Порождение вычислимых функций в машине с бесконечными регистрами.
16. Параллельная машина с бесконечными регистрами. Пример.
17. Нормальные алгоритмы Маркова. Определение. Граф-схема алгоритма. Заключительная подстановка. Пример.
18. Обобщенный нормальный алгоритм. Принцип нормализации. Виды композиций нормальных алгоритмов.
19. Универсальный нормальный алгоритм.
20. Алгоритмическая система рекурсивных функций. Понятие рекурсивной функции. Понятие вычислимой функции. Пример.
21. Соотношение элементарной и вычислимой функции. Определение функции по индукции. Пример.
22. Алгоритмическая система рекурсивных функций. Полная система формального описания схем.
23. Понятие примитивно-рекурсивной функции и её связь с вычислимой функцией. Частично рекурсивные функции. Мю- оператор.
24. Понятие логической схемы алгоритма. Процесс реализации. Распределение сдвигов. Понятие подчиненности. Оператора max.
25. Полная система преобразования Янова. Отмеченные функции.
26. Система формул перехода S1. Система скобочных формул S2. Система схемных формул S3.
27. Система преобразований Лазарева-Дьяченко. Переход о системы формул к логической схеме алгоритма. Оптимизация логической схемы алгоритма.
28. Графическая схема алгоритма. Формальное определение. Оптимизация на уровне ГСА. Пример.
29. Матричная схема алгоритма. Формальное определение. Оптимизация на уровне МСА. Пример.
30. Объединение схем алгоритмов. Определяющие конъюнкции. Кодирование схем.
31. Определяющие функции. Процесс доопределения.
32. Алгоритмически неразрешимые проблемы. Примеры.
33. Трудноразрешимые проблемы. Примеры.
34. Трудноразрешимые проблемытеории графов.
35. Подходы к решению трудноразрешимых задач.
36. Асимптотическая оценка сложности алгоритмов. Верхняя, средняя и нижняя оценки.
37. Асимптотическая оценка сложности алгоритмов. Амортизированная оценка.
38. Оценка рекурсивных алгоритмов. Основная теорема.
39. Оценка рекурсивных алгоритмов. Метод деревьев рекурсии.
40. Типовые алгоритмические идеи. Разделяйивластвуй. Meet-in-the-middle.
41. Типовые алгоритмические идеи. Стохастические алгоритмы. Монте-Карло. Лас-Вегас.
42. Типовые алгоритмические идеи. Жадные алгоритмы. Динамическое программирование.
43. Типовые алгоритмические идеи. Эвристические алгоритмы. Предпросчет.
44. Типовые алгоритмические идеи. Приближенные алгоритмы. Структуры данных на основе хэш-функций.
45. Логика и исчисление высказываний. Формальное определение.
46. Исчисление высказываний. Проблемы общезначимости и непротиворечивости.
47. Проверка общезначимости в исчислении высказываний. Алгоритм Квайна. Пример.
48. Проверка общезначимости в исчислении высказываний. Алгоритм редукции. Пример.
49. Задача логического вывода. Подходы к решению.
50. Исчисление высказываний. Метод резолюций.
51. Исчисление высказываний. Метод резолюций для хорновских дизъюнктов.
52. Исчисление высказываний. Метод деления дизъюнктов.
53. Формальное определение исчисления предикатов первого порядка.
54. Преобразование выражения исчисления предикатов первого порядка в конъюнктивную нормальную форму. Пример.
55. Сколемовское преобразование. Операции унификации.
56. Модальные логики. Области применения. Особенности. Примеры.
57. Темпоральная логика линейного времени. Формальное определение. Модальные операторы.
58. Темпоральная логика ветвящегося времени. Формальное определение. Модальные операторы.

**1. Возникновение и развитие дисциплины "Теория алгоритмов". Математическое понятие алгоритма.**

Откуда взялся алгоритм? Существовал арабский математик с сложным именем, Аль Хорезми, жил в Багдаде, где была организация "дом мудрости", занимающаяся развитием науки. Он опубликовал трактат "книга об индийской арифметике", в котором обобщил правила десятичной арифметики: сложение\вычитание\умножение\деление. Последовательности действий как считать стали называть алгоризмами. Возможно, образовалось от Аль Хорезми. Позже появился термин Алгорифм и только во второй половине XXв образовался термин ‘алгоритм’. XX веке алгоритмами стали называть конечную совокупность правил, которые позволяют решать те или иные классы задач.

Определение из БСЭ (Большая советская энциклопедия) плохо тем, что непонятно, что понимается под правилами и классом задач. При усложнении задач было выведено новое определение. Для более сложных задач нужен был какой-то новый инструмент. Понадобилось какое-то формальное определение алгоритма.

Алгоритм в математике.

Алгоритм - конструктивно задаваемое соответствие между словами в абстрактном алфавите.

Абстрактный алфавит - конечная последовательность символов, называемых буквами.

Буква - элемент любого множества

Слово - упорядоченная последовательность букв

Алгоритм — это точное предписание, которое задает вычислительную процедуру. Данная процедура перерабатывает исходный набор данных P (входной объект) и направлена на получение обусловленного этими данными результата Q (объекта на выходе). Она состоит из отдельных, элементарных шагов — тактов работы алгоритма. Каждый шаг заключается в смене одного набора данных другим набором (объектом, состоянием). Переход от предыдущего состояния к последующему происходит по заранее заданному, конечному набору инструкций. Эти инструкции не должны предполагать никаких догадок и вероятностных соображений со стороны человека или машины, нужно только точно их исполнять. Некоторые состояния опознаются как заключительные, при которых процесс вычислений заканчивается. При этом на основе некоторой инструкции определяется, что считать результатом вычислений. Пусть алгоритм A имеет исходный набор данных P. Возможны следующие три случая протекания алгоритмического процесса. 1. На некотором шаге возникает состояние, опознаваемое как заключительное. При этом происходит остановка вычислений и выдается результат Q. 2. Каждое очередное состояние сменяется последующим до бесконечности, т.е. процесс вычислений никогда не останавливается. 3. При некотором состоянии возникает ситуация, когда процесс вычислений обрывается без выдачи результата (например, не срабатывает инструкция для определения результата вычислений). Тем самым нет перехода к следующему шагу и нет результата вычислений. В этом случае говорим, что произошла безрезультатная остановка. Считается, что алгоритм A применим к исходному набору данных P тогда и только тогда, когда выполнен случай 1, т.е. когда процесс вычислений заканчивается и получен результат вычислений Q. Этот результат будем обозначать через AP. В случаях 2 и 3 результат AP не существует.

**2. Алфавитный оператор. Общность понятия, способы задания.   
Примеры простых и кодирующих операторов.**

Алфавитный оператор - или оператор отображения. Любое всякое соответствие слову из входного алфавита слова из выходного алфавита.

Так как Оператор преобразует одно слово в другое можно провести некую параллель между функцией в математике и оператором в теории алгоритмов.

Слово, к которому применяется алгоритм, называется входным. Слово, вырабатываемое в результате применения алгоритма, называется выходным.

Операторы бывают:

● Простые. Данные операторы выполняют простое побуквенное отображение. Букву из входного меняют на букву из выходного. Способ задания простого оператора может быть обычной таблицей соответствий. Таблица соответствий если область определения оператора конечна.

● Кодирующие. Данные операторы заменяют букву из входного на последовательность букв из выходного. Причем можно как кодировать, то и декодировать. Следовательно, закодированное слово не должно являться префиксом другого слова. Каждое слово должно иметь свой код. Способ задания сложного оператора это некий набор правил для разных слов, преобразующий их в коды. Если коды имеют одинаковую длину, кодирование называется Нормальным. Набор правил, когда область определения оператора бесконечна. Правил конечное количество

Примеры:

Простой. Ну например это будет код цезаря. Будем выполнять побуквенное отображение.

Сложный. Как вариант можно привести наверно кодирование десятичных цифр. Перевод в 10сс осуществляется по некому набору правил.

● Многозначные и однозначные алфавитные операторы. Примеры.

Однозначный алфавитный оператор - одному входному слову соответствует не более одного выходного слова.

Многозначный алфавитный оператор - нескольким входным словам соответствует одно, либо наоборот.

Примеры:

● Однозначный. Кодирование, например. Одно входное, одно выходное

● Многозначный. Например, это будет некий набор овощей. На входе название, на выходе ОВОЩ.

● Соотношение между алфавитным оператором и алгоритмом. Стохастические алгоритмы. Самоизменяющиеся алгоритмы.

Алфавитный оператор - всякое соответствие слову из входного алфавита слова из выходного алфавита.

Алгоритм - алфавитный оператор, задаваемый с помощью конечных систем правил.

Если область определения алфавитного оператора конечна, то оператор может быть задан простой таблицей соответствия. В случае бесконечной области определения алфавитного оператора он задается системой правил, позволяющей за конечное число шагов найти выходное слово, соответствующее заданному входному слову.

Два алфавитных оператора называются равными, если они имеют одну область определения и любому входному слову они выдают одинаковые выходные.

Алфавитный оператор, не сопоставляющий данному входному слову аi никакого выходного слова bj (в том числе и пустого), не определен на этом слове. Совокупность всех слов, на которых алфавитный оператор определен, называется областью его определения.

Два алгоритма называются равными, если их алфавитные операторы равны, а системы правил совпадают.

Два алгоритма называются эквивалентными, если их алфавитные операторы равны, а системы правил различны.

Например, алгоритмы сортировки эквиваленты.

Алфавитный оператор - Что получим?

Алгоритм - как получим?

Алгоритм = алфавитный оператор + правила, определяющие его действия.

Стохастический алгоритм - алгоритм, у которого применяемое правило или выходное слово определяется случайным образом. Одному входному слову могут соответствовать несколько выходных слов поскольку существует вероятность получения каждого. Ибо есть некая вероятность применения правила в системе правил. Сумма вероятностей единица.

Самоизменяющийся алгоритм - Алгоритм, в котором получаемое выходное слово зависит не только от входного, но и от предыстории работы алгоритма.

**3. Многозначные и однозначные алфавитные операторы.**

Различают однозначные и многозначные алфавитные операторы (АО).

Под однозначным алфавитным оператором понимается такое алфавитное отображение, которое каждому входному слову ставит в соответствие не более одного выходного слова.

Например:

a1----------------------------------------------b2

a2----------------------------------------------b3

a3----------------------------------------------b1

Многозначный — 1 входной — несколько выходных.

**4. Соотношение между АО и алгоритмами. Стохастические и самоизменяющиеся алгоритмы.**

Алфавитные операторы, которые задаются системой правил принято называть алгоритмами. Необходимо понимать принципиальное различие между понятием алфавитного оператора и понятием алгоритма. В понятии алфавитного оператора существенным является конкретное соответствие, которое устанавливает оператор между словами, а не способ, которым это соответствие устанавливается. А понятии алгоритма — наоборот, основным является способ задания соответствия.

Таким образом, алгоритм — алфавитный оператор вместе с правилами, которые определяют его действие.

Стохастический алгоритм - алгоритм, дающий программу решения задачи несколькими путями или способами, приводящими к вероятному достижению результата

Алгоритмы, ставящие в соответствие одному входному своду одно выходное, называются детерминированными. Кроме детерминированных алгоритмов существуют алгоритмы самоизменяющиеся, которые при обработке различных слов могут изменяться, и случайные, в которых правила могут выбираться случайным образом.

**5.Свойства алгоритмов. Понятие алгоритмической системы.**

1) Дискретность – процесс сопоставления входному слову выходного должен быть представим как последовательное выполнение конечного числа некоторых шагов. Кроме того, для выполнение каждого шага конечное время.

2) Детерминированность – в каждый момент времени следующий шаг процесса сопоставления должен быть однозначно определяем. Кроме того, каждому входному слову P должно соответствовать только одно выходное слово Q. Любой стохастический алгоритм может быть представлен как детерминированный путём включения вероятностных характеристик в состав алгоритма

3) Понятность – все шаги процесса должны быть доступны для осуществления исполнителю

4) Завершаемость – для любого входного слова из области определения алгоритм должен выполнять процесс сопоставления ему выходного слова за конечное число шагов

5) Массовость – алгоритм должен быть применим для решения не только одной задачи, а целого класса однотипных задач (Мощность области определения >1)

6) Результативность – применение алгоритма должно приводить к получению выходного слова

Алгоритмическая система:

Всякий способ задания алгоритмов называется алгоритмической системой

При описании используются специальные формальные средства – формализмы

Два типа:

1) Алгебраические: используют символику, когда алгоритмы рассматриваются в виде линейных текстов (например, рекурсивные функции)

2) Геометрические: алгоритмы строятся в виде множеств, в которых реализуются отображения или бинарные отношения, а также используется математический аппарат теории графов (например, нормальные алгоритмы Маркова, граф-схемы и т.д.)

**6. Понятие абстрактного вычислителя. Машина Поста.**

Машина Поста - алгоритмическая система, которая, с точки зрения структуры, может быть представлена как совокупность двух элементов:

- Лента

- Каретка (считывающая и записывающая головка)

ПЕРЕДВИГАЕТСЯ КАРТЕТКА, А НЕ ЛЕНТА!

Лента поделена на ячейки: x- стоит отметка, в ином случае отметки нет

В начале работы каретка стоит рядом с ячейкой с номером ‘0’

Определение: жёсткое сопоставленные в каждой секции уникальные номера образуют целочисленную базовую систему координат машины Поста.

Определение: Временной системой координат машины Поста называется полученная в результате сдвига на фиксированное число относительно базовой система координат.

Состояние ленты – информация о том, какие секции отмечены, а какие нет

Команды:

L a – каретка влево, переход к команде a

R a – каретка вправо, переход к команде a

P a – постановка отметки, переход к команде a

E a – стирание отметки, переход к команде а

J a B – команда передачи управления. Если напротив каретки находится неотмеченная ячейка, осуществляется переход к команде с номером А, иначе - к команде с номером B

S – остановка

Номер следующей команды, расположенный в конце каждой инструкции, называется отсылкой.

Первый номер следующей команды в инструкции передачи управления (J A B) – называется верхней отсылкой, второй – нижней.

Программой Машины Поста называется конечный непустой список команд, обладающий двумя свойствами. Во-первых, на К-ой позиции списка должна стоять команда с номером K. Во- вторых, отсылка любой из команд списка должна совпадать с номером некоторой (другой или той же самой) команды списка.

Три варианта завершения программы:

1) Дойдёт до невыполнимой команды (например, стирание несуществ. Метки). В таком случае – безрезультатная остановка

2) Команда S

3) Бесконечная работа

Определение: Совокупность положения каретки в базовой системе координат и номер текущей команды в выполняемой программе называется состоянием каретки

Определение: Совокупность состояния ленты и состояния каретки называется состоянием машины Поста.

**7. Понятие программы машины Поста. Диаграмма и схема Поста.**

**Программа машины Поста** - конечный не пустой список команд, обладающий 2 свойствами:

1. На k-ой позиции списка стоит команда с номером k;
2. Отсылка любой из команд списка должна совпадать с номером некоторой другой или той же самой команды списка.

**Отсылка** – номер следующей команды, расположенный в конце каждой инструкции.

**Верхняя отсылка** – первый номер в команде передачи управления.

**Набор команд МП**:

* l α – сдвиг влево
* r α – сдвиг вправо
* p α – поставить отметку
* e α – убрать отметку
* j α β – команда передачи управления (если отметка есть переходим к α, иначе переходим к β)
* s – прекратить выполнение программы

**Замечания**:

* Если в ходе выполнения программы произведен переход к команде с неизвестным номером, выполнение программы прекращается.
* Попытка стереть метку там, где ее нет, или поставить метку повторно считается ошибкой, и машина аварийно останавливается.
* Если в команде переход не указан, то переход происходит на следующую строку.

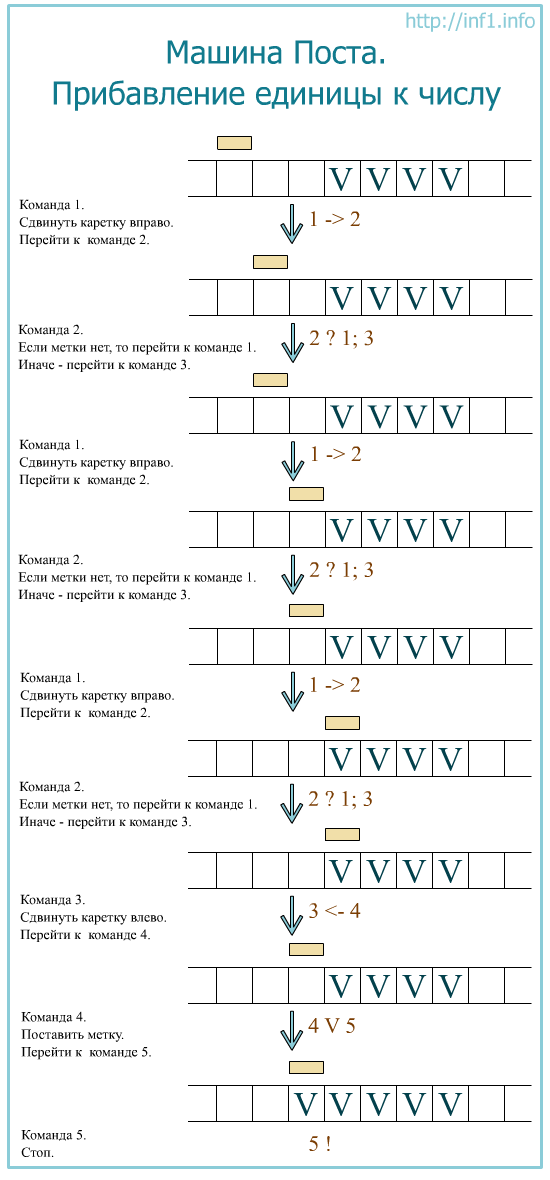
Для работы машины нужно задать программу и ее начальное состояние (т. е. состояние ленты и позицию каретки).

После запуска возможны варианты:

* работа может закончиться невыполнимой командой (стирание несуществующей метки или запись в помеченное поле);
* работа может закончиться командой s;
* работа никогда не закончится.

**Пример**:

Прибавить к некоторому заданному числу 1.

Целое положительное число на ленте машины Поста представимо идущими подряд метками, которых на одну больше, чем кодируемое число. Это связано с тем, что одна метка обозначает ноль, а уже две – единицу, и т.д.

Пусть каретка стоит слева от меток.

1. r 2
2. j 3 1
3. l 4
4. p 5
5. s

**Поглощение** – операция замены в пределах программы одной команды на другую с исключением последней.

1. α заменяем на β;
2. вычеркиваем α;
3. меняем все отсылки α +1, α +2, α +3 на α , α +1, α +2.

**Диаграмма машины Поста**. Способ анализа программы МП. Представляет собой набор кружков. Переход между командами обозначается стрелками или дугами. Сам кружок делится на 2 части. Вверху номер, внизу команда.

2

1

r

2

2

j

2

3

l

2

4

p

2

5

s

**Схема Поста**. Для лучшего понимания узлы разбивают на группы в зависимости от назначения. Группа изображается в виде прямоугольника с номером. В нем список включенных команд + текстовая метка, поясняющая назначение группы. Для соединения стрелки.

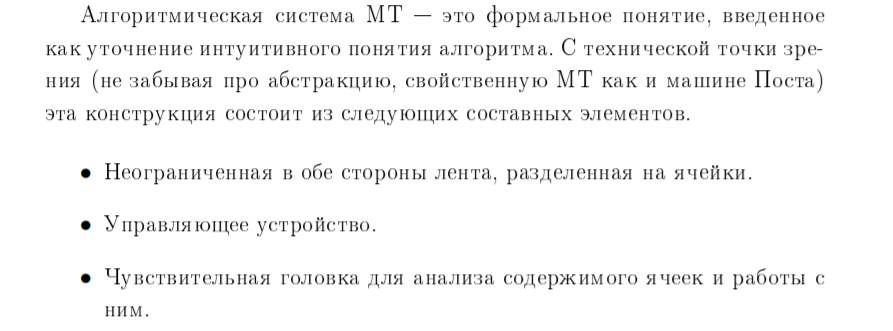


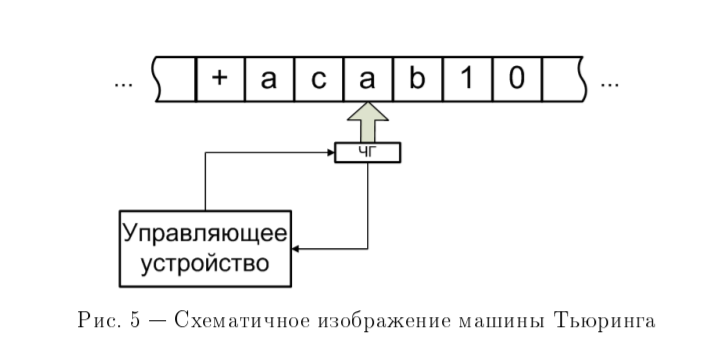
**8. Понятие абстрактного вычислителя. Машина Тьюринга.**

. Абстрактная вычислительная машина - теоретическое построение, с помощью которого вводится строгое, математическое определение алгоритма.

ИЛИ

***Абстрактной машиной***/ ***Абстрактным вычислителем*** (*abstractmachine*) называется математическая формализация, которая моделирует правила выполнения программы (или, иначе, алгоритмы) для реальной вычислительной машины (компьютера).





МТ называется стандартной, если при сдвиге ленты содержимое ячейки меняет значение.

А={S0,S1,S2…Sn} – внешний алфавит, содержит символы, которые находятся на ленте.

Q={q0,q1,q2…qm) – внутренний алфавит, описывает состояние управляющего устройства(q0-нач. состояние, q с точкой – заключительное)

Любая совокупность из последовательности содержимого ячеек МТ и одного из внутренних состояний называется конфигурацией МТ

Программа МТ – это конечная совокупность команд, заданная на одних и тех же внутренних и внешних алфавитов. Также система команд.

**11. Модификации машины Тьюринга. Примеры работы.**

Для некоторых задач составление алгоритмов работы обычной машины Тьюринга представляет значительные трудности, связанные с необходимостью где-то хранить результаты промежуточных вычислений, или производить посимвольное сравнение нескольких групп элементов. В некоторых случаях наличие дополнительной (рабочей) ленты или размещение входных слов на нескольких лентах одновременно позволяет получить более лаконичное решение.

Однако как это ни покажется парадоксальным, вычислительная способность таких машин совершенно не превосходит их одноленточные аналоги. Иными словами, задачи, которые можно решить на многоленточной машине с произвольным количеством лент, всегда решаются и при помощи одноленточной машины.

Функция f(x) вычислима по Тьюрингу, если её значения могут быть вычислены некоторой машиной Тьюринга, на ленте которой изначально было задано стандартное представление аргумента x. Значение функции – запись на ленте после остановки.

S1, i, S1, i+1, q1, S1, i+2

S2, j, S2, j+1, q2, S2, j+2

………………………………

SL, k, SL, k+1, qL, SL, k+2

qi S1 S2 … SL → qj S1’ S2’ … SL’ d1 d2 … dL

12. Композиции МТ: произведение, возведение в степень, итерация.

Произведением двух МТ будем называть такую МТ с общим внешним алфавитом А и с суммарным внутренним алфавитом Q и программой, задаваемой следующим образом. Во всех программах, содержащих переход в заключительное состояние q˙0, производится замена его на qn+1. Во всех командах МТ2 наоборот переход в q˙0 остается неизменным, а все остальные символы заменяются qn+j. MТ=МТ1\*МТ2

Пример: Пусть имеется МТ1, позволяющая отыскать ближайшую слева единицу после группы нулей.

A={0,1}

Q={q0, q1, q2}

МТ1:

1) q1 0 q2 0 R

2) q1 1 q1 1 R

3) q2 0 q2 0 R

4) q2 1 q0. 1 S à q2 1 q3 1 S

Пусть имеется МТ2, позволяющая стирать все единицы слева, если они есть, до ближайшего нуля. Q = {q˙0, q1}

МТ2:

1) q1 0 q0. O S

2) q1 1 q1 0 R à q3 1 q3 0 R

Тогда произведение МТ1 и МТ2 позволит отыскать ближайшую группу единиц и стирать их.

• q1 0 q2 0 R

• q2 0 q2 0 R

• q3 0 ˙q0 0 S

• q1 1 q1 1 R

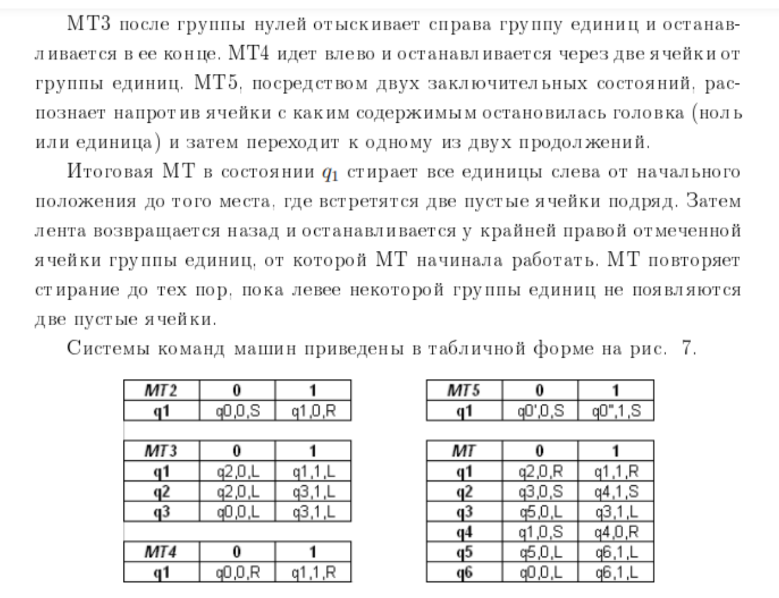
• q2 1 q3 1 S

• q3 1 q3 0 R

*Возведение в степень МТ называется последовательное произведение МТ на саму себя n раз.*

*Операции итерации называется отождествление одного из конечных состояний МТ с начальным.*

Пример: Пусть задана МТ MT = MT˙ 4 · MT5 ( (1) MT3 (2) MT˙2)



**13. Понятие абстрактного вычислителя. Машина с бесконечными регистрами.**

Абстрактная вычислительная машина - вычислитель, которого не может существовать в природе в силу некоторых ограничений.

МНР – это лента, бесконечная в одну сторону и разбитая на ячейки, которые называются *регистрами* и обозначаются *R*1, *R*2,… Каждый регистр содержит некоторое неотрицательное целое число. Содержимое регистра *Rn* обозначается *rn*.

*Программа* МБР – это конечный список *команд* (инструкций) I1,I2,…,I*s* 4-х видов:

* Z(n) - обнулить число, хранящееся в ячейке под номером n;
* S(n) - увеличить число, хранящееся в ячейке под номером n, на единицу;
* T(n, m) - скопировать число, хранящееся в ячейке n, в ячейку m;
* J(n, m, q) - условный переход, работающий так: если число в ячейке с номером n равно числу в ячейке с номером m, то программа переходит к обработке команды, содержащейся в строке q (и далее по списку).

Начальная конфигурация - значения регистров.

Выполнение останавливается, когда попадаем в команду с несуществующим номером. Результат работы обычно записывается в первой ячейке.

Результат работы МБР:

* вычисления сходятся – результативная остановка
* вычисления расходятся – вычисления не закончатся

Пусть *f* – частичная функция на множестве целых неотрицательных чисел. Если задана программа *P(a1, a2, …, as)* и получается некоторый результат b, где *ai* и *b* принадлежат множеству целых неотрицательных чисел, то справедливы следующие утверждения:

* Вычисление *P* от *(a1, a2, …, as)* сходятся к *b*, когда *P(a1, a2, …, as)* прекращается и в заключительном состоянии в *R1:=b*. (Вычисления сходятся к определенному значению, если происходит остановка, и в регистре записано это значение.)
* Программа *P*вычисляет *f,* если для всех*a1, a2, …, as, b*справедливо, что вычисления *P(a1, a2, …, as)* сходится к *b*, когда *a1, a2, …, as*∈*D(f)* и *f(a1, a2, …, as)=b.* (Программа вычисляет функцию, когда для определенной конфигурации выполнение программы сходится к значению этой функции)
* Функция *f*является МБР вычислимой, если есть программа, которая вычисляет *f.*

Класс всех МБР вычислимых функций обозначается через φ, при этом класс всех n местных функций обозначается через φn.

Пример: вычитание второго числа из первого

1. J(1,2,6)
2. S(2)
3. S(3)
4. J(1,2,6)
5. J(1,1,2)
6. T(3,1)
7. (9,7,0)
8. (9,8,0)
9. (9,8,1)

10)(9,9,1)

11)(9,9,2)

12)(2,9,2)

**14. МБР-вычислимые функции. Элементарные вычислимые функции.**

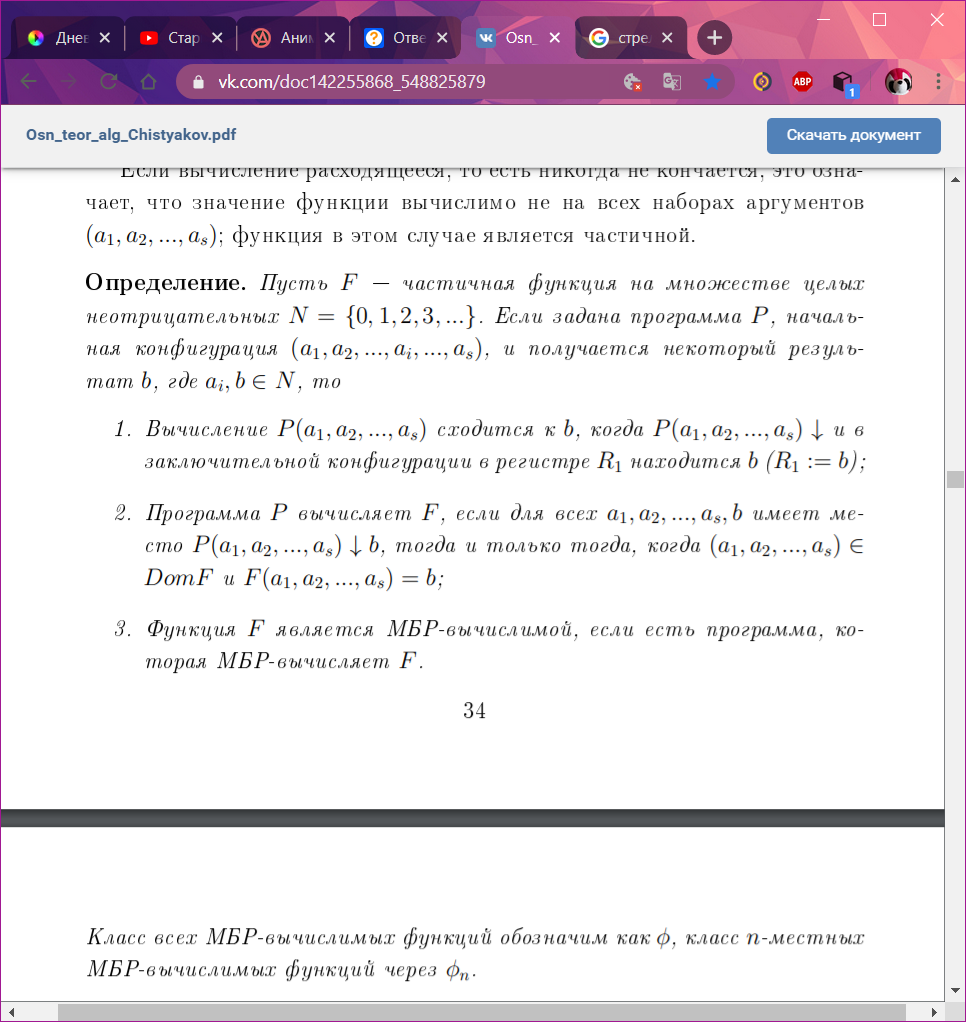
Мбр вычислимые ф-ии. Элементарные вычислимые ф-ии

Пусть дана программа P

В общем случае будем использовать обозначения

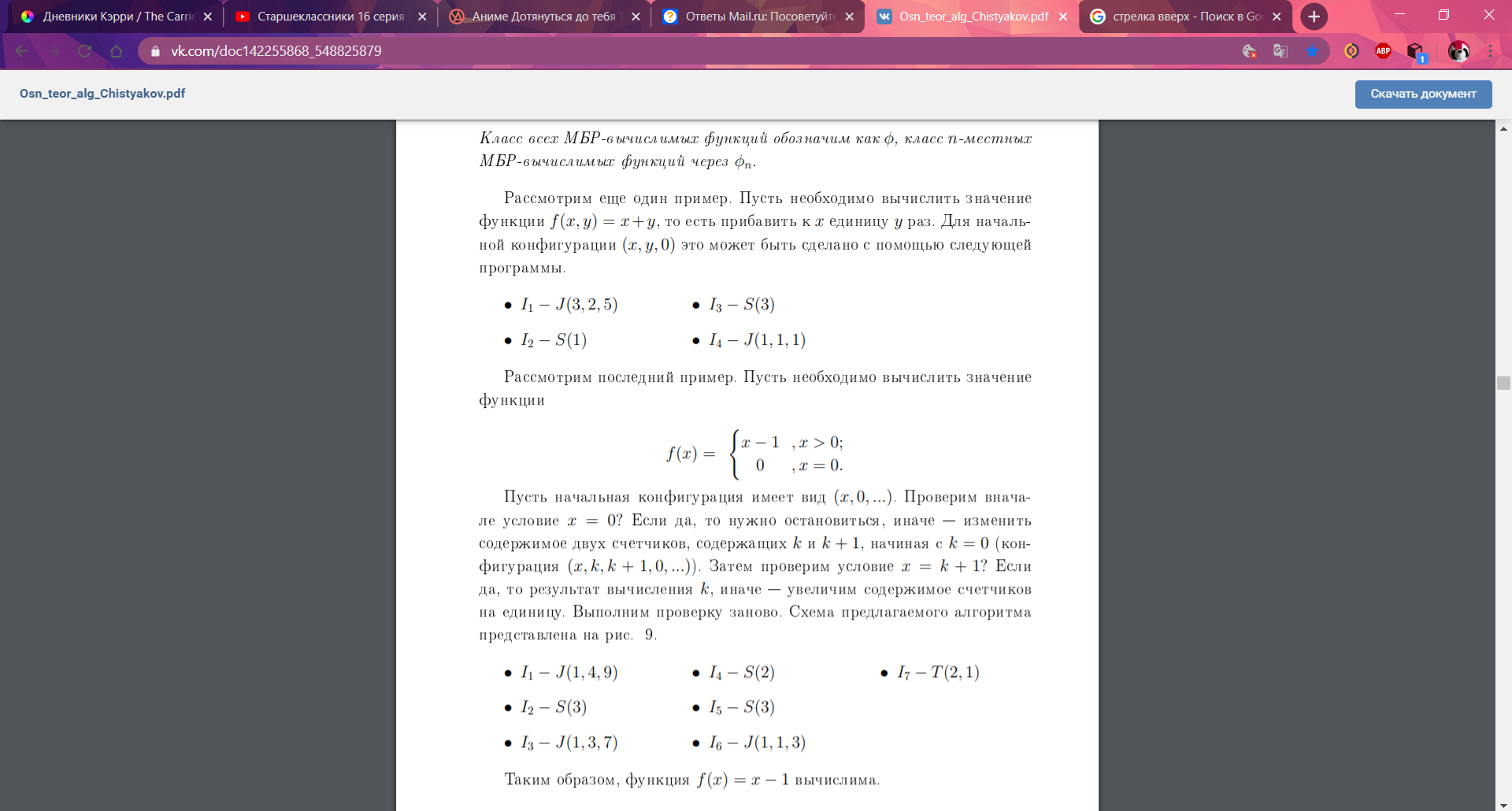
1. P(a1,a2, …, as) – вычисление по программе P с начальной конфирурацией (a1,a2, …, as)
2. P(a1,a2, …, as)↓ - вычисление в определенный момент останавливается(сходится)
3. P(a1,a2, …, as)↑ - вычисление никогда не останавливается(расходится)
4. P(a1,a2, …, as)= P(a1,a2, …, as,0,0,…)

Если вычисление расходящееся, это означает что значении ф-ии вычислимо не на всех наборах аргументов (a1,a2, …, as) . ф-ия в этом случае является частичной



Класс всех МБР-вычислимых функций обозначается *φ* ,а класс n-местных МБР-вычислимых ф-ий – *φn*

Примеры:



**15. Порождение вычислимых функций в машине с бесконечными регистрами.**

Лемма: следующие функции вычислимы:

1. 0(x)=0, для любого X, не принадл. N

2. ¬S (x) = x+1 – функция следования

3. Функция проекции vni(x1, x2,.., xn)= xi , i принадлежит [1; n], n>=1

Элементарные функции вычислимы для МБР, так как им соответствуют команды.

Определение: Программа P=(i1, i2, ..is) имеет стандартный вид, если для всякой команды условие перехода J(m, n, q), q<=S+1

Для построения более сложных функций используются следующие операции:

1. Соединение программ. После завершения работы программы P, начинает свою работу программа Q. Конкатенацией программ называется программа вида P\*Q=набор команд, где к номеру каждой команды из программы Q прибавлено количество S команд в программе P. Тем самым достигается последовательное выполнение программ. В условных переходах программы Q все переходы заменяются на q+s

2. Подстановка программы. **Теорема** (своими словами). Пусть мы имеем функцию f и еще некоторое множество вычислимых функций. Тогда мы можем получить вычислимую функцию, если в качестве аргументов функции f подставим другие вычислимые функции. Например у нас есть функция суммирования f(x,y)=x+y. Она вычислима. Но вместо х мы можем подставить x1+x2, что будет тоже функцией суммирования, а вместо у просто значение, типа х3. Такая функция будет вычислимой.

3. Рекурсия. По определению Рекурсия - способ задания функции путем определения каждого из ее значений в терминах, ранее определенных значений для младших аргументов и возможность использования других уже определенных функций.

Младшие означает, что между аргументами есть частичное отношение порядка, которое определяет какие аргументы являются меньше других.

**Теорема:** У нас есть функции f(x1,x2,…,xn) g(x1,x2,…,xn,y,z) - вычислимые.

Тогда функция H(x1,x2,…,xn,y) полученная из f и gc помощью рекурсий вычислима.

Где в базе у=0 получаем одну функцию, и для больших вызываем эту же функцию для меньших значений у.

4. Неограниченная минимизация. Допустим у нас есть функции f и g. В функции f есть дополнительный аргумент у. В остальном одинаковые аргументы. Определим значение g такое, что оно будет являться минимальным у, при подстановке которого в f мы получим ноль.

**Теорема:** Пусть f(x1…xn, y) вычислима, тогда g(x1, x2,..xn)=ky(f(x1, x2…xn, y)), y=0 также вычислима.

Именно оператор наименьшего корня и следит, при каком значении выбранного аргумента наблюдаемая им функция впервые опустится до нуля. Это значение выбранного аргумента и будет значением оператора наименьшего корня.

**16. Параллельная машина с бесконечными регистрами. Пример.**

Параллельная МБР представляет собой модификацию МБР.

В ней можно выполнять несколько процессов одновременно.

1. Z(n)
2. S(n)
3. T(m,n)
4. J(m,n,q)

Появились новые команды:

● G(n) – команда захвата ресурса (другие входят в состояние ожидания)

● P(n) – разблокировать ранее заблокированный регистр

● s (k,n) – команда запуска программы с номером k сопоставив ей регистр с номером n. (запуск программы 1 и выдать ей регистр с номером 2. Регистр дает нам знать о выполнении программы. После выполнения становится 1)

● I(n) – команда ввода

● O(n) – команда вывода

Появилась поддержка регистровой адресации S(R(2)) - инкрементирует регистр с номером, записанным в регистре 2 (например, цикл)

Пример. f(a,b,c,d)=a\*b+c\*d

Сначала выполняем одновременно 2 умножения, потом складываем.

**17. Нормальные алгоритмы Маркова. Определение. Граф-схема алгоритма. Заключительная подстановка. Пример.**

**Нормальный алгоритм Маркова** – это такие обобщенные нормальные алгоритмы, ГСА которых имеют специальный вид, когда всякий оператор подстановки q1->q2 входит в паре с распознователем.

**ГСА** - конечный связный граф G, удовлетворяющий следующим условиям.

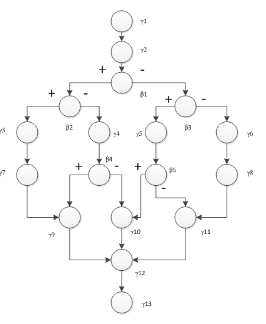
•В каждом графе имеется два отмеченных узла входной, из которого выходит лишь одна стрелка, и выходной, из которого не выходит ни одной стрелки.

•Из любого другого узла выходит либо одна (γ-узел), либо две (β-узел)стрелки.

•Стрелки, исходящие изβ-узла имеют отметки «+» и «-».

•Имеются конечные множества функциональных элементов - множество преобразователей Q={Qi}и множество распознавателей R={Rj},i= 1...n, j= 1... m.

•Каждомуγ-узлу однозначно сопоставлен преобразователь Qi, аβ-узлу распознаватель Rj, при этом некоторые преобразователи и распознаватели могут быть сопоставлены нескольким различным узлам.

Пример  
Последовательность вида ↨ βi1βi2...γj называется - ветвью. Здесь знак ↨ означает («+») или («-»).

Две ветви называются равносильными, если последовательности указного вида совпадают до порядка следования логических операторов.

Два дерева равносильны, если их ветви равносильны.

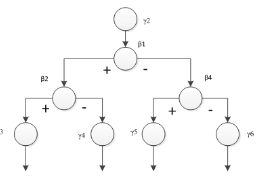
Две ГСА равносильны, если между путями отА0доАkсуществует взаимно однозначное соответствие.

Назовем подграф-схемой Hi ГСАΓнекоторый связный подграф G′ графа G, удовлетворяющий следующим условиям.

•Каждый подграф G′ содержит один входнойγ-узел, один или болеевыходныхγ-узлов иβ-узлы.

•Будем предполагать, что к подграфу относятся и стрелки выходныхγ-узлов.

•Узлам подграфа G′ сопоставлены те же функциональные элементы, что и узлам графа G



**Заключительные подстановки** используются для завершения работы алгоритма. Заключительные подстановки обязательны для реализации сложных конструктивных алфавитных операторов



**18. Обобщенный нормальный алгоритм. Принцип нормализации. Виды композиций нормальных алгоритмов.**

**Обобщенный нормальный алгоритм** - такой алгоритм, заданный ГСА, составленный исключительно из распознавателей слов и операторов подстановки.

Оператор подстановки выполняет замену вхождения слова на другое слово.

При этом предполагается, что каждому оператору подстановки вида p 1 → p 2 подсоединяется только 1 стрелка (стрелка «+» распознавателя p 1 ).

**Принцип нормализации** - для любого конструктивного алгоритма алфавитного отображения в произвольном конечном алфавите можно построить эквивалентный ему Нормальный алгоритм над этим алфавитом.

Принцип сводится к тому, что какой бы ни был сложный алгоритм, мы всего можем записать его в данном виде. Над алфавитом означает, что нам может не хватить символов этого алфавита. Но мы можем расширить этот алфавит своими словами и сможем применять этот алгоритм к словам алфавита А. Марков еще пришел к выводу, что нам достаточно добавить один символ, чтобы построить нормальный алгоритм. ВСЕ АЛГОРИТМЫ НОРМАЛИЗУЕМЫ.

**Принципы композиции**.

1. **Суперпозиция.** Суперпозицией является последовательное выполнение алгоритмов. Когда получая слово из одного алгоритма мы подаем его другому для получения нового слова. B(A(p)) = D(p) – суперпозиция алгоритмов A и B

2. **Объединение алгоритмов.** Объединение алгоритмов, это алгоритм, который в том же алфавите преобразует исходное слово в записанные рядом слова(конкатенация). Два выходных слова, это слова двух объедияемых алгоритмов. P от A⋂P от B🡪C(P)=A(P)\*B(P)

3. **Разветвление**. Композиция трех алгоритмов А В и С.

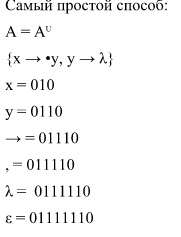


При этом область определения итогового алгоритма будет совпадать с пересечением областей определения алгоритмов А В и С

4. **Повторение(итерация).** В таком случае имеем 2 алгоритма и входное слово. Применяем алгоритм В и смотрим. Если слово не пустое, то подаем его в алгоритм А. Полученное слово снова подаем в алгоритм В то тех пор, пока не получим пустое. Как только получено пустое, значит последнее слово, полученное применением алгоритма А будет выходным.

**19. Универсальный нормальный алгоритм.**

Для построения универсального нормального алгоритма нам необходимо построить изображение алгоритма. Мы можем закодировать алфавит и операции.



Если у нас будет в универсальном алгоритме некоторое входное слово А, то можно построить универсальную систему подстановок, которая разбивала бы это слово на 2 части. Описание алгоритма и само входное слово. Далее алгоритм будет выполняться для этих входных данных.

Теорема: существует такой нормальный алгоритм, называемый универсальным, который для любого алгоритма и слова переводит слово, полученное приписываем изображением этого слова к изображению алгоритма к слову, являющимся изображением выходного слова.

**20. Алгоритмическая система рекурсивных функций. Понятие рекурсивной функции. Понятие вычислимой функции. Пример.**

Данная алгоритмическая система сложилась исторически первой (1931-1934 гг.), ввиду чего получила достаточно полное и всестороннее развитие. В ее основании лежит использование конструктивно определяемых арифметических целочисленных функций, которые получили специальное название – рекурсивные функции.

В основе лежат целочисленные функции, а также особый способ их задания. Идея в том, что какую бы информацию алгоритм не обрабатывал всегда можно работать с целыми числами. Для числа символов n в алфавите можно ввести систему счисления n+1, перевести в 10сс и выполнять операции с целыми числами.

***Рекурсией*** называют способ задания функции, при котором значение определяемой функции для произвольных значений аргументов выражается через значения определяемой функции для меньших значений аргументов.

***Рекурсивная функция* -** функция, значение которой для произвольных аргументов выражается через значение этой же функции для меньших аргументов.

Применение рекурсивных функций в теории алгоритмов основано на идее нумерации слов в произвольном алфавите последовательными числами. Например, логично нумеровать целыми числами слова, располагая их в порядке возрастания длины, а слова одинаковой длины – в произвольном порядке или, например, по алфавиту. Тогда после завершения нумерации входных и выходных слов алфавитный оператор превращается в некоторую функцию вида *y = f(x)*, где как аргумент *x*, так и сама функция *y,* принимают целые неотрицательные значения. Эта функция может быть определена лишь для некоторых значений аргумента *x*, составляющих область определения функции *y*. Подобные, частично определенные, целочисленные функции получили название «арифметических».

Функция *y = f(x1, x2, …, xn)* называется ***алгоритмически вычислимой***, если существует алгоритм, позволяющий определить значение функции при любых значениях переменных *x1, x2, …, xn,* входящих в область определения *f.*

Это определение носит интуитивный характер, поскольку не ясно что понимается под вычисляющим алгоритмом.

Пример рекурсивной функции:

F(0,a)=a

F(1,a)=a+a

F(n+1,a)=a+f(n,a)

**21. Соотношение элементарной и вычислимой функции. Определение функции по индукции. Пример.**

Исследования показали, что любой алгоритм может быть всегда сведён к численному алгоритму. Применение рекурсивных функций в теории алгоритмов основано на идее нумерации слов в произвольном алфавите последовательными числами. Тогда после завершения нумерации входных и выходных слов алфавитный оператор превращается в некоторую функцию, где как аргумент х, так и сама функция у, принимают целые неотрицательные значения. Эта функция может быть определена лишь для некоторых значений аргумента х, составляющих область определения функции у. Подобные, частично определённые, целочисленные функции получили название «арифметических».

Функция y=f(x1, x2, …, xn) называется алгоритмически вычислимой, если существует алгоритм, позволяющий определить значение функции при любых значениях переменных x1, x2, …, xn, входящих в область определения f.

Данное определение носит чисто интуитивный характер, поскольку не ясно, что понимается под вычисляющим алгоритмом.

Назовём «элементарными» арифметические функции, получаемые из целых неотрицательных чисел и переменных с помощью операций сложения, арифметического вычитания, умножения и арифметического деления, а также построения сумм и произведений.

Попробуем ответить на вопрос: все ли вычислительные функции являются элементарными? Другими словами – шире ли класс вычислимых функций класса элементарных?

Оказывается функция ψ(0,a) =a,ψ(1,a) =аа,...,ψ(n+ 1,a) =aψ(n,a) не может быть реализована с помощью элементарных функций. Таким образом, итерация операции возведения в степень позволяет получить неэлементарную функцию. (Док-во смотреть в лекции «рекурсивные функции»)

Это позволяет сделать вывод о том, что класс вычислительных функций шире класса всех элементарных функций.

Обратим внимание на тот факт, что ψ(x,a) была задана по индукции, то есть вначале было определено начальное значение ψ(0,a) — базис, а также указан способ вычисления всех ее значений по предыдущим значе­ниям с помощью вполне доступных операций. Отметим, что определение по индукции может быть введено на любом упорядоченном множестве, где введены понятия «предыдущий» и «следующий».

Обозначим через х' функцию «следование за». Она определяет переход от одного элемента к следующему за ним. Для натурального ряда чисел N = 0,1,2,3,... имеем: 0' = 1,1' = 2,2' = 3 и т.д. Очевидно, в этом случае функция следования совпадает с функцией х + 1. Однако, в зависимости от вида множества приращение может и отличаться от 1.

Уточним общую схему задания функции φ(0) по индукции.

1. Задаем базис φ(0);
2. Для любого х укажем, каким образом значение f(x′) формально опи­сано

φ(0) =q0;

φ(x′) =ψ(x,φ(x)).

В общем случае функция может быть зависимой от п переменных. То­гда схема будет иметь вид

φ(0,x1,x2,...,xn) =g(x1,x2,...,xn);

φ(y′,x1,x2,...,xn) =h(y,φ(y,x1,x2,...,xn),x2,x3,...,xn).

Если g и h известны и вычислимы, тогда с помощью указанной схе­мы можно последовательно вычислять φ(1,x1,x2,...,xn),φ(2,x1,x2,...,xn)и так далее. Таким образом эта схема определяет вычислимую функцию.

**22. Алгоритмическая система рекурсивных функций. Полная система формального описания схем.**

Рекурсией называют способ задания функции, при котором значение определяемой функции для произвольных значений аргументов выражается через значение определяемой функции для меньших значений аргументов.

Исследования показали, что любой алгоритм может быть всегда сведён к численному алгоритму. Применение рекурсивных функций в теории алгоритмов основано на идее нумерации слов в произвольном алфавите последовательными числами.

Среди арифметических функций выделим следующие, особо простые, которые получили название «элементарные арифметические».

1. φ(x) =x′ функция следования, применимая для натурального ряда (сокращенное обозначение S(x));
2. φ(x1,x2,...,xn) =q,q=const — функция константы (сокращенное обозначение Сqn);
3. φ(x1,x2,...,xn) =xi — функция тождества (сокращенное обозначение

Xin);

1. φ(x1,x2,...,xn) =ψ(χ1(x1,x2,...,xn),...,χm(x1,x2,...,xn)) – функция подстановки.

Полная система формального описания схем для получения произволь­ных функций может быть представлена следующим образом.

1. φ(x) =S(x) =x′;
2. φ(x1,x2,...,xn) =Cqn=q;
3. φ(x1,x2,...,xn) =Xin=xi;
4. φ(x1,x2,...,xn) =ψ(χ1(x1,x2,...,xn),...,χm(x1,x2,...,xn));
5. φ(0) =q;

φ(x′) =ψ(x,φ(x)).

1. φ(0,x1,x2,...,xn) =g(x1,x2,...,xn);

φ(y′,x1,x2,...,xn) =h(y,φ(y,x1,x2,...,xn),x2,x3,...,xn).

Схемы под номерами 1-4 задают первоначальные функции, играя роль аксиом, а схимы 5 и 6 можно рассматривать как правила вывода.

**23. Понятие примитивно-рекурсивной функции и её связь с вычислимой функцией. Частично рекурсивные функции. Мю - оператор.**

***Частично рекурсивные функции***. Функция f называется *частично рекурсивной функцией* (ч.р.ф.), если она является одной из простейших функций или может получиться из них с помощью конечного числа применений операторов *суперпозиции*, *примитивной рекурсии* и *минимизации*, т.е. существует последовательность функций f1,f2,..., fn=f, каждая из которых является либо простейшей, либо получена из предыдуших с помощью одного из указанных операторов. Указанная последовательность функций называется частично рекурсивным описанием функции f.

Функция f называется общерекурсивной функцией (о.р.ф.), если она *частично рекурсивна* и всюду определена.

Функция f называется ***примитивно рекурсивной функцией*** (п.р.ф.), если она *частично рекурсивна* и для нее существует частично рекурсивное описание, использующее лишь операторы *суперпозиции* и *примитивной рекурсии*. В таком случае оно называется примитивно рекурсивным описанием функции f.

Нетрудно проверить, что каждая *примитивно рекурсивная функция* всюду определена, т.е. является общерекурсивной (обратное, вообще говоря, неверно).

**Пример 8.1**. Постоянные функции.

Пусть fn(x1,...,xn)=k для всех наборов аргументов (x1,...,xn) и числа . Тогда



**Пример 8.2**. Сложение: +2(x,y)=x+y.

Функция сложения определяется следующей *примитивной рекурсией*.



Следовательно, +2 =R(I11,[s1;I33]).

**Пример 8.3**. Умножение: x2(x,y)=x x y.

Используя сложение, умножение можно задать следующей *примитивной рекурсией*:



Следовательно, x2 =R(o1,[+;I33,I13]).

**Пример 8.4**. Минус 1: .



Нетрудно проверить, что .



**Пример 8.5**. Вычитание : , если x >= y и , если x < y.



Вычитание определяется следующей примитивно рекурсивной схемой:



Следовательно, .



**Пример 8.6**. Предикаты равенства и неравенства нулю:



Примитивная рекурсивность этих функций следует из равенств  и .



**Пример 8.7**. Модуль разности: .



**Пример 8.8**. rm(x,y)= остаток от деления y на x (при x=0 положим rm(0,y)=y ).

Заметим, что



Тогда функцию rm(x,y) можно задать примитивно рекурсивной схемой



Правую часть второго равенства легко представить как функцию g(x,y,rm(x,y)), полученную с помощью *суперпозиции* уже построенных *примитивно рекурсивных функций*.

**Пример 8.9**. Нигде не определенная функция .



Эта функция может быть задана, например, соотношением .



Отметим, что все функции в примерах 8.1 - 8.8 являются примитивно рекурсивными.

*Функция называется примитивно рекурсивной, если она может быть получена из простейших функций  с помощью конечного числа применений операторов суперпозиции и примитивной рекурсии.*

*Функция называется частично рекурсивной, если она может быть получена из простейших функций  с помощью конечного числа применений суперпозиции, примитивной рекурсии и µ-оператора. Если функция всюду определена и частично рекурсивна, то она называется общерекурсивной.* (Отметим, что не всегда частично рекурсивную функцию можно эффективно доопределить до общерекурсивной.)

Ясно, что всякая примитивно рекурсивная функция будет и частично рекурсивной (и даже общерекурсивной, так как каждая примитивно рекурсивная функция всюду определена), поскольку для построения частично рекурсивных функций из простейших используется больше средств, чем для построения примитивно рекурсивных функций. В то же время класс частично рекурсивных функций шире класса примитивно рекурсивных функций, так как все примитивно рекурсивные функции всюду определены, а среди частично рекурсивных функций встречаются и функции не всюду определенные.

Понятие частично рекурсивной функции оказалось исчерпывающей формализацией понятия вычислимой функции. При построении аксиоматической теории высказываний исходные формулы (аксиомы) и правила вывода выбирались так, чтобы полученные в теории формулы исчерпали бы все тавтологии алгебры высказываний. К чему же стремимся мы в теории рекурсивных функций, почему именно так выбрали простейшие функции и операторы для получения новых функций? Рекурсивными функциями мы стремимся исчерпать все мыслимые функции, поддающиеся вычислению с помощью какой-нибудь определенной процедуры механического характера. Подобно тезису Тьюринга, в теории рекурсивных функций выдвигается соответствующая естественно-научная гипотеза, носящая название **тезиса Чёрча:**

*Числовая функция тогда и только тогда алгоритмически (или машинно) вычислима, когда она частично рекурсивна.*

И эта гипотеза не может быть доказана строго математически, она подтверждается практикой, опытом, ибо призвана увязать практику и теорию. Все рассматривавшиеся в математике конкретные функции, признаваемые вычислимыми в интуитивном смысле, оказывались частично рекурсивными.

Теперь мы рассмотрим более подробно классы примитивно рекурсивных функций и частично рекурсивных функций и докажем, что все функции из каждого из этих классов вычислимы на подходящих машинах Тьюринга.

## Примитивно рекурсивные функции

Итак, функция примитивно рекурсивна, если она может быть получена из простейших функций  с помощью конечного числа применений операторов суперпозиции и примитивной рекурсии. Рассмотрим ряд примеров примитивно рекурсивных функций.

**Пример 33.7.** Покажем, как, исходя из простейших функций, с помощью оператора примитивной рекурсии получить следующую функцию, называемую усеченной разностью:

Во-первых, отметим, что функция , получающаяся из функции  фиксированием второго аргумента, удовлетворяет следующим соотношениям: т.е. получена из простейших функций  и  с помощью оператора примитивной рекурсии. Во-вторых, нетрудно понять, исходя из определения усеченной разности, что эта функция удовлетворяет также равенствам: для любых  и . Тождества показывают, что двухместная функция  получена с помощью оператора примитивной рекурсии из простейшей функции  и функции .

**Пример 33.8.** Покажем, что функция  может быть получена из простейших с помощью оператора примитивной рекурсии. Для функции верны следующие тождества:

которые можно записать в виде

 или ,

а это и есть схема примитивной рекурсии, основывающаяся на простейших функциях  и .

**Пример 33.9.** Аналогично операции сложения, очевидные соотношения для операции умножения

говорят о том, что функция  получена из простейшей функции  и функции  с помощью оператора примитивной рекурсии.

**24. Понятие логической схемы алгоритма. Процесс реализации. Распределение сдвигов. Понятие подчиненности. Оператора max.**

Основное достоинство рассматриваемых ниже *логических схем* алгоритмов (*ЛСА*) состоит в том, что, являясь *по* существу разновидностью языка операторных схем, они допускают *запись* алгоритма в строчку, что часто является удобным, т.к. появляется возможность исключить процесс рисования, вычерчивания, как это имеет *место* в *ГСА*. Важным является также наличие развитой системы преобразований *ЛСА* и возможности формального перехода к автоматному отображению.

Основными элементами *ЛСА* являются так же, как и в ГСА, *операторы* и логические условия.

Основные отличия от ГСА состоят в том, что для указания взаимосвязей между операторами и логическими условиями используются верхние и нижние стрелки.

Логической схемой алгоритма называется строчка, составленная из символов операторов , или  и логических условий , а также верхних и нижних стрелок. Иногда верхние и нижние стрелки заменяют на правые и левые полускобки.



Итак, *ЛСА*- строчка, составленная из символов операторов , логических условий  и верхних  и нижних  стрелок, причем:



* Сильная операторная вершина  и одна конечная  ;



* Строка начинается с  и заканчивается  ;



* Не должно быть двух нижних стрелок  с одинаковыми номерами;



* Для каждой нижней стрелки  должна быть по крайней мере одна верхняя;



Переход *по* логическому условию , стоящему в *ЛСА*



осуществляется так:

* Если , то после  выполнится ,



* Если , то после  выполнится .



*Безусловный переход* для ясности может быть обозначен дополнительным символом, например .



*ЛСА* для МП, представленной на [рис. 3.7](https://www.intuit.ru/studies/courses/1031/242/lecture/6230?page=2#image.3.7) выглядит так:



Правило чтения *ЛСА* состоит в следующем.

Вначале анализируется элемент *ЛСА*, следующий непосредственно за оператором . Если рассматриваемым элементом является оператор, то он отмечается (выписывается) и на следующем шаге анализируется стоящий справа элемент (оператор или логическое условие).



Если рассматриваемым элементом является логическое условие  производится проверка этого условия;



*Анализ* *ЛСА* при соблюдении сформулированных правил приводит через некоторое количество шагов к получению строчки операторов, называемой значением *ЛСА* при заданной последовательности наборов логических условий.

Пусть задана *ЛСА*.



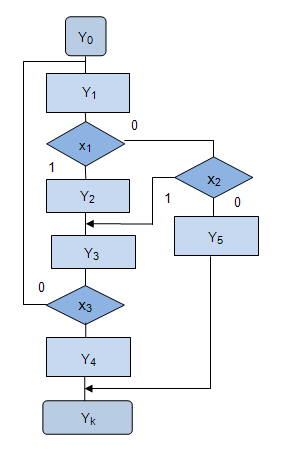
Построим соответствующую ей ГСА. За начальным оператором  следует оператор  и далее логическое условие . Если логическое условие выполняется, то есть , то следующим оператором выполняется . Если логическое условие не выполняется, то есть , то следующим оператором выполняется , то есть оператор, стоящий за нижней стрелкой с номером 1.



Далее в *ЛСА* за оператором  стоит оператор  и . В такой последовательности и изображаем их на ГСА. Далее строим аналогичным образом.



Одной важной особенностью *ЛСА* является возможность неоднозначной записи одного и того же алгоритма.



**Рис. 3.8.**

Так, ГСА на [рис.3.8](https://www.intuit.ru/studies/courses/1031/242/lecture/6230?page=3#image.3.8) может быть описана еще несколькими вариантами *ЛСА*:



Понятие логической схемы алгоритма. Процесс реализации. Распределение сдвигов. Понятие подчиненности. Оператор max.

Микрокомандой является совокупность микроопераций, выполняемых на одном такте.

Микрооперация - одна строчка алгоритма.

Логической схемой алгоритма называют конечную строку из символов операторов и условий, начал и концов стрелок таких, что для каждой стрелки вверх с индексом найдется одна и только одна стрелка вниз с этим же индексом.

Каждый оператор встречается в алгоритме только 1 раз.

Элементарное выражение - операторы и условия.

Выражение - строка, состоящая из элементарных выражений.

Дельта - возможный набор логических переменных.

Процесс реализации ЛСА:

1. Задать начальные значения логических условий. (дельта)

2. Указывается самое левое элементарное выражение

3. Выполняется шаг алгоритма, получается новое значение дельты.

4. Составляется результат выписыванием выражений.

Полученная строка - результат работы ЛСА.

Распределение сдвигов - сопоставленный каждому оператору набор логических переменных, которые могут изменить свое значение после выполнения этого оператора.

● Пустое распределение сдвигов. Оператор ничего не меняет

● Универсальное распределение сдвигов. Оператору сопоставлено все множество логических переменных.

● Нормальное распределение сдвигов. Множество переменных является частью множества всех переменных.

Последовательность распределений сдвигов считается допустимой, когда наборы до и после выполнения оператора отличаются только на переменные из распределения сдвигов.

ЛСА равносильны, если для всякой допустимой последовательности наборов значения этих ЛСА совпадают при заданном распределении сдвигов.

Равносильный преобразования состоит в замене одних выражений другими и в изменении взаимного расположения выражений, при котором ЛСА остаются равносильны исходной.

**25. Полная система преобразования Янова. Отмеченные функции.**

В качестве основной в [6, 23] выдвинута проблема построения системы

эквивалентных преобразований схем, полной в некотором классе схем.

Содержательно под полнотой системы преобразований в классе Ж

понимается следующее ее свойство: для любых двух эквивалентных схем из Ж

существует цепочка преобразований, принадлежащих системе и переводящих

одну схему в другую. Для проверки полноты предъявляемой системы

строится алгоритм, который получает на вход две произвольные схемы из Ж,

в случае их эквивалентности преобразует схемы в изоморфные друг

другу, а при неэквивалентности просто останавливается, убедившись в этом.

Позднее (см. [9]) такой алгоритм был назван алгоритмом канонизации

пар схем из Ж.

Таким образом, проблема построения полной системы фактически

рассматривается в том случае, когда в Ж разрешима проблема

эквивалентности, т. е. если существует алгоритм, распознающий, эквивалентны ли

поступившие на его вход две произвольные схемы из Ж. Этот подход

закрепился и в дальнейшем, в силу чего проблема эквивалентности, имеющая

и самостоятельное значение, приобрела дополнительный интерес.

В [23] проанализирован класс схем, каждая из которых использует

любой операторный символ не более одного раза. Обозначим этот класс

через М0. Основной результат для него дается теоремой 1

Теорема 1 Для любого распределения сдвигов s существует

система L ^эквивалентных преобразований схем, полная в классе Ж0.

Но полнота системы устанавливается алгоритмом канонизации пар

схем, поэтому из теоремы 1 следует разрешимость проблемы

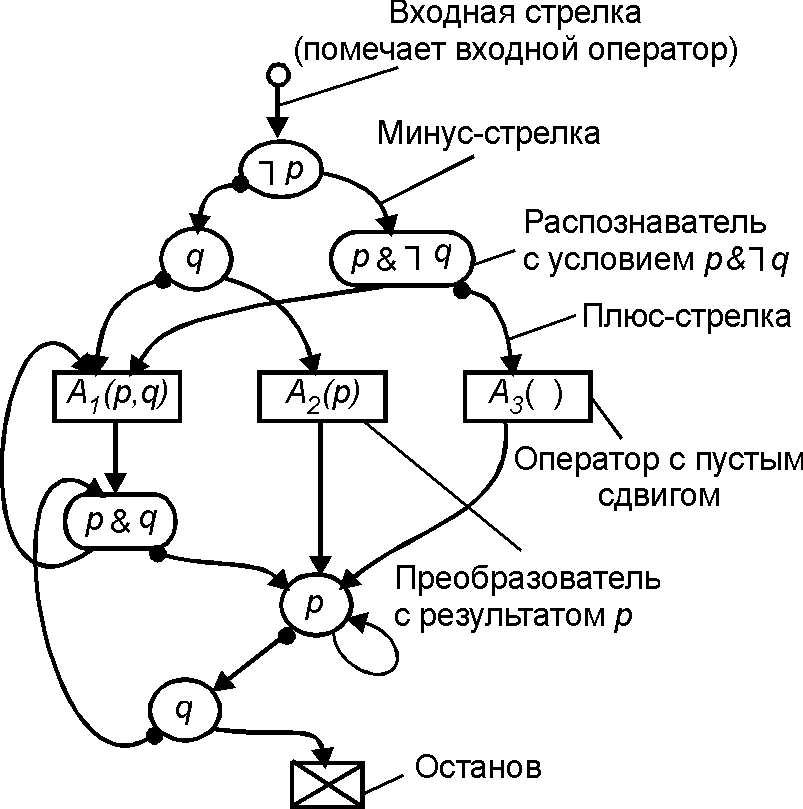
Ьл-эквивалентности в JiQ.

Заметим, что исследованиями, предпринятыми после опубликования

статьи [23], было показано, что утверждение теоремы 1 сохраняет свою

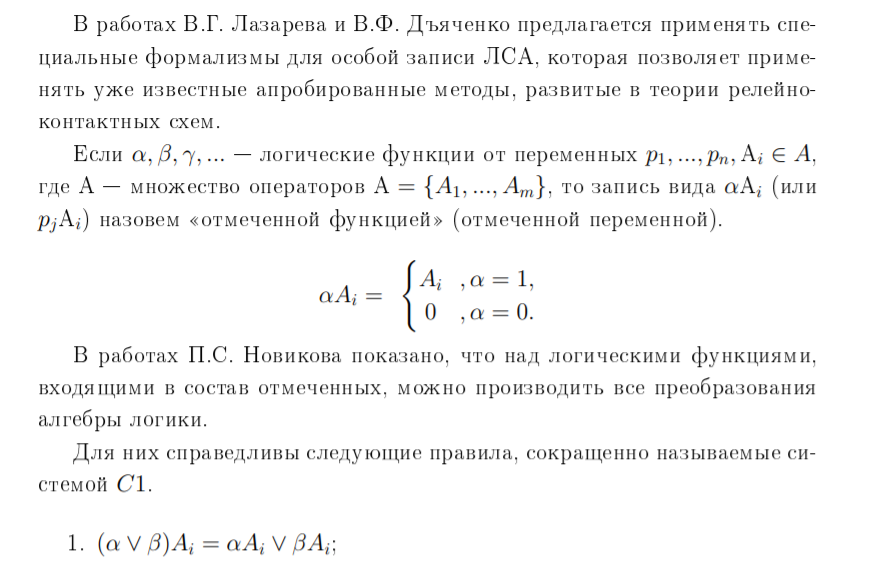
силу при замене Ж0 на множество всех схем над У и Р.

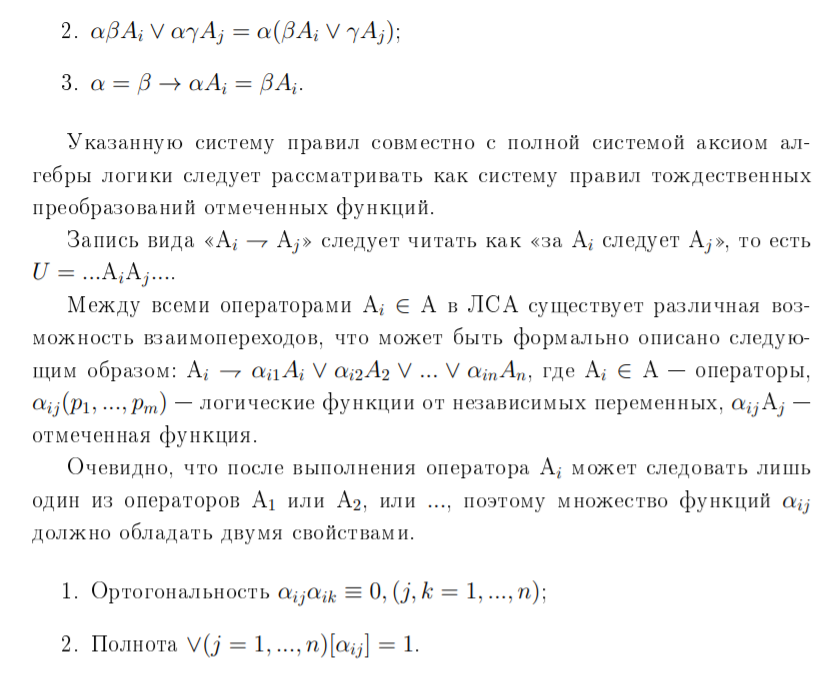
СХЕМА

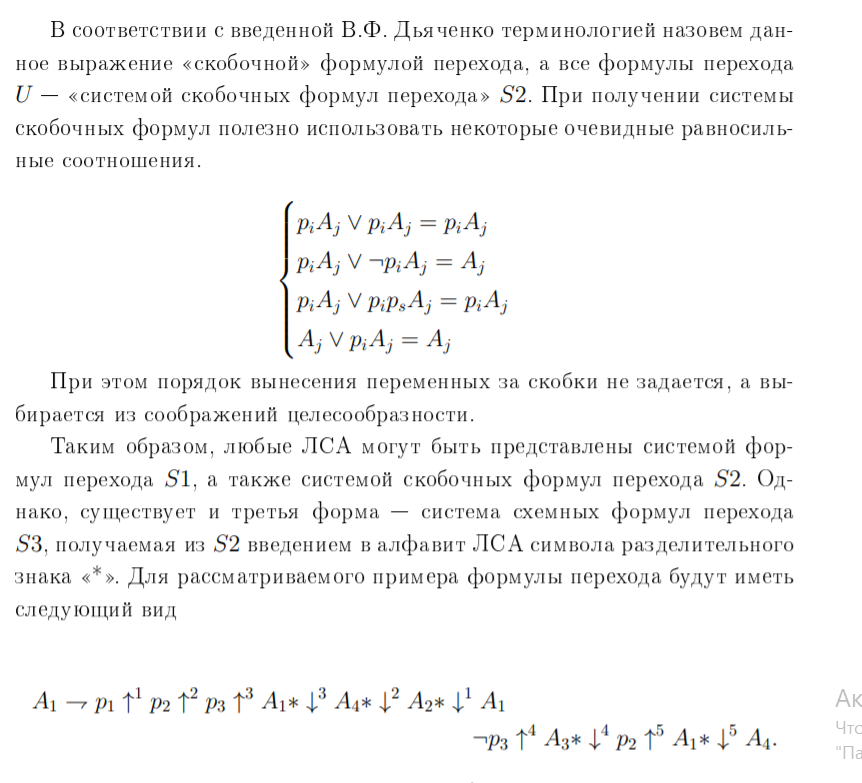
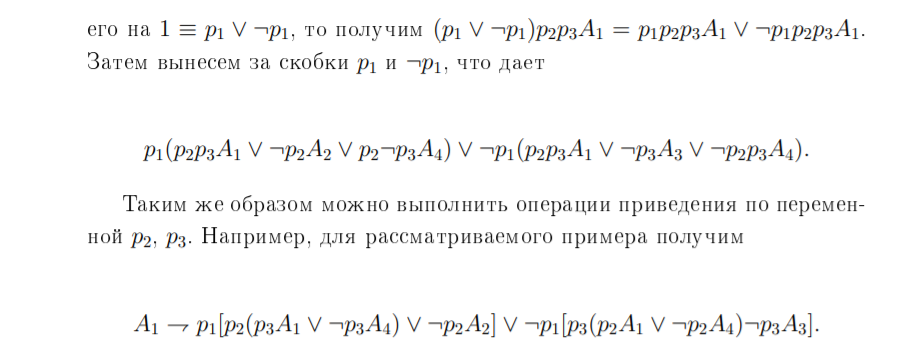
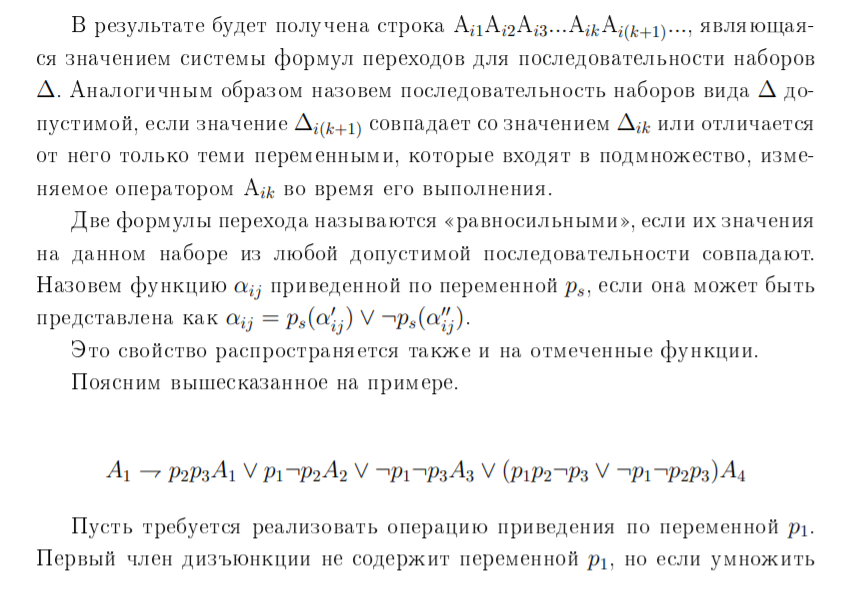
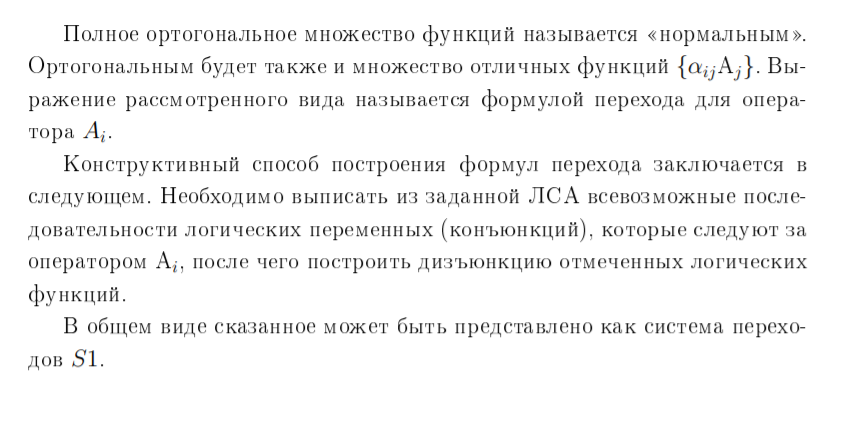


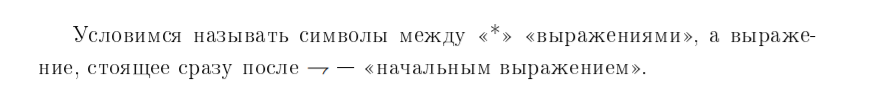
Откуда взял:  
[https://docviewer.yandex.ru/view/140483934/?page=1&\*=iCWnWiiLe3p%&lang=ru](https://docviewer.yandex.ru/view/140483934/?page=1&*=iCWnWiiLe3p%&lang=ru)

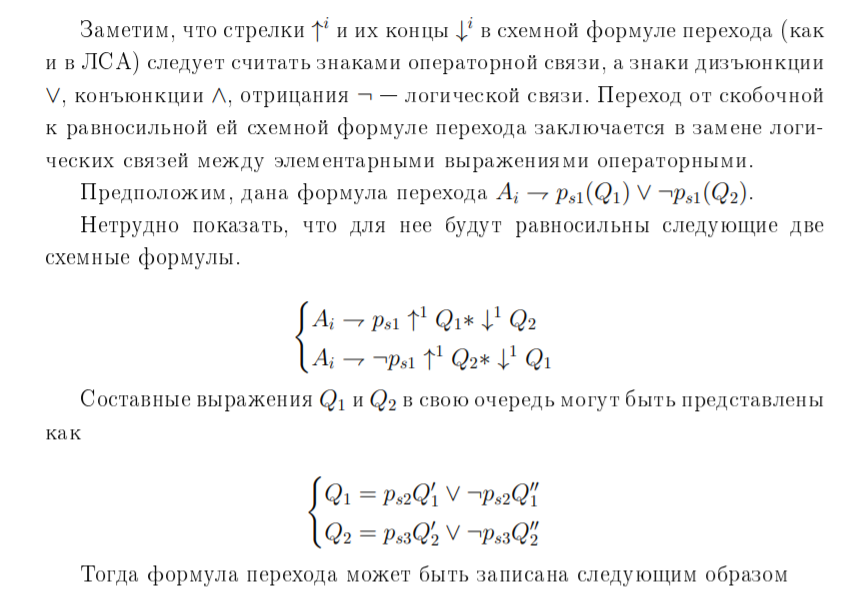
26. Система формул перехода s1, система скобочных формул s2, система схемных формул s3.

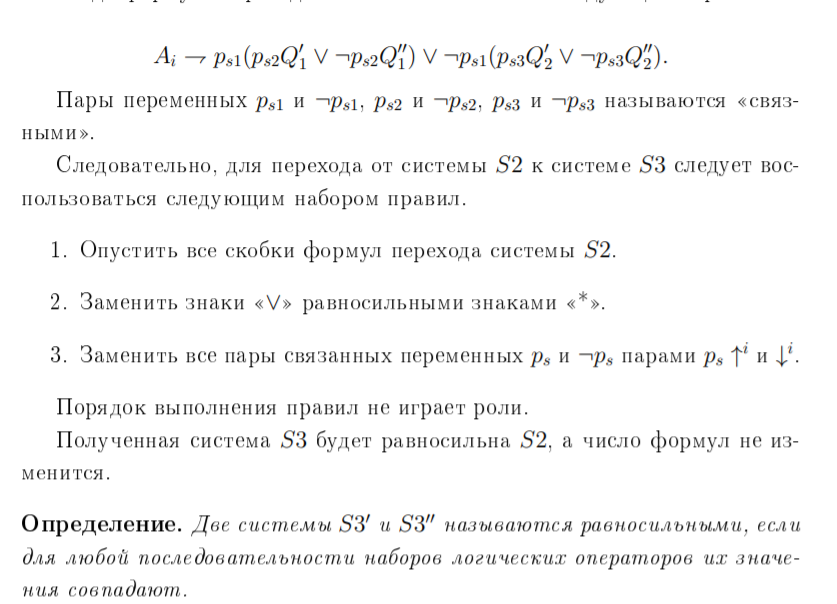






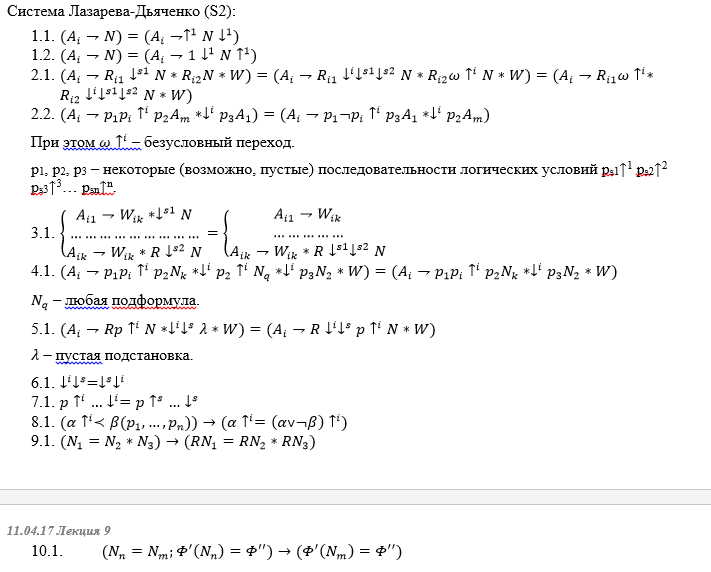






**27. Система преобразований Лазарева-Дьяченко. Переход о системы формул к логической схеме алгоритма. Оптимизация логической схемы алгоритма.**

Система преобразований Лазарева-Дьяченко. Переход от системы формул к логической схеме алгоритма. Оптимизация логической схемы алгоритма.



1 - избавляет от ненужных стрелок

2 - можно перемещать операторы

2.2 - можно инвертировать условия для изменения порядка т.к. мы убираем в S3 часть условий

3 - можно перемещать между переходами в схеме

5 - циклы

6 - можно менять стрелки местами

7 - можно перенумеровать стрелки

Рассматривается способ упрощения логических схем алгоритмов

(ЛСА), основанный на учете наборов значений переменных (логических

условий), которые не встречаются при выполнении алгоритма. Способ

упрощения ЛСА состоит в преобразовании не доопределенных формул перехода и получении общего решения, из которого выбирается частное решение в виде формулы перехода. Затем формулы перехода переводятся в ЛСА.

Учет неиспользуемых наборов позволяет сократить общее число логических

условий и изменить порядок их проверки, что может в некоторых случаях привести к дополнительному объединению одинаковых выражений в ЛСА.

**Алгоритм перехода к ЛСА:**

* 1. Выписать А0 и справа от него начальное выражение из А0 и вычеркнуть его.
  2. Выписать справа от Ai начальное выражение из Ai и вычеркнуть его. Повторять.
  3. Выписать любое невычеркнутое выражение, вычеркнуть его.
  4. Перенести Ак в конец строки, соблюдая отсылки.
  5. Остановка.

**Оптимизация ЛСА:**

* 1. Начинать вынесение за скобку с той переменной, по которой приведено множество конъюнкций.
  2. Если приведено по нескольким переменным, то можно начать с любой переменной.
  3. Если по всем, то безразлично за исключением ситуации когда в формуле есть два и более отмеченных элементарных конъюнкта с одним А.

**28. Графическая схема алгоритма. Формальное определение. Оптимизация на уровне ГСА. Пример.**

Впервые граф-схемы алгоритмов были предложены Л.А.Калужниным.

ГСА - конечный связанный граф, удовлетворяющий условиям:

* 1. Есть отмеченные узлы входной и выходной.
  2. Из каждого узла выходит либо одна стрелка, либо две. γ – одна, β – две.
  3. Если две, то плюс и минус.
  4. Есть множество распознавателей и преобразователей.
  5. Каждой γ сопоставлен преобразователь, каждой β - распознаватель.

Можно выделить подграф-схему из основной. Один вход и может быть много выходов.

Если в подграфе нет контура, то там есть дерево. В деревьях можно выделять поддеревья.

Ветки равносильны, если последовательности совпадают до порядка следования логических условий.

2 дерева равносильны, если равносильны их ветки.

2 ГСА равносильны, если между началом и концом существует взаимно однозначное соответствие (деревья этих ГСА равносильны).

Оптимизация на уровне ГСА видимо состоит в том, чтобы убирать равносильные ветки/деревья.

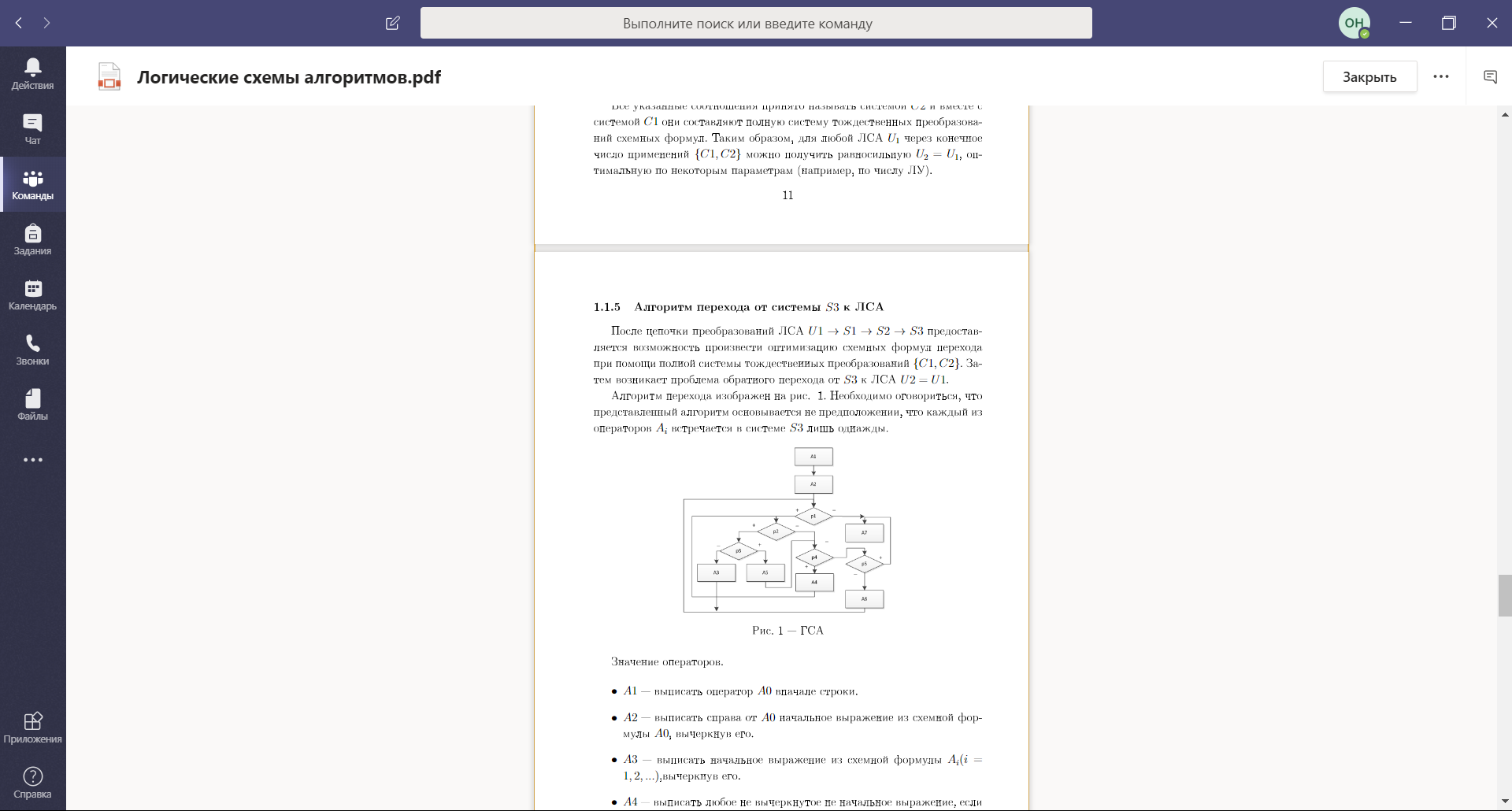
**29. Матричная схема алгоритма. Формальное определение. Оптимизация на уровне МСА. Пример.**

Наглядным способом представления алгоритмов являются матричные схемы, зарекомендовавшие себя как удобный инструмент для анализа систем переходов.

Фактически, МСА является собой табличной формой отображения си-  
стемы S1, формальное изложение которой состоится в ходе следующей  
лекции. Сейчас зафискируйте этот факт. Более формально.  
Определение. Матричной схемой алгоритма называется квадратная  
матрица, строки которой соответствуют операторам A0, A1, ..., An, а  
столбцы — операторам A1, A2, ..., An, Ak. При этом содержимое ячейки  
[i, j] соответствует составному логическому условию, определяющему  
возможность перехода от выполнения оператора Ai к выполнению опе-  
ратораAj  
.  
Естественным образом, каждая строка матрицы, по аналогии с каждой  
формулой перехода, должна отвечать двум требованиям.  
1. Ортогональность AijAik ≡ 0,(j, k = 1, ..., n), то есть конъюнкция  
любых двух элементов строки должна быть тождественно ложна;  
2. Полнота ∨(j = 1, ..., n)[Aij ] = 1, то есть дизъюнкция всех элементов  
строки должна быть тождественно истинна.  
Определим процесс выполнения МСА U для последовательности набо-  
ров ∆.

• Задать значения переменных ∆i1 и определить значение функций  
A1j (j = 1, ..., n, k) в строке 1 матрицы (оператор A0); в связи с требо-  
ванием ортогональности, найдется ровно одна обращающаяся в исти-  
ну логическая функция; выполнить переход к j-ой строке матрицы  
(соответствующей определенному оператору Аj−1).  
• Задать значения переменных ∆ij и определить значение истинной  
функции Aij = 1, при этом найдя строку (и оператор), к которому  
выполняется очередной переход.  
• Процесс продолжается до тех пор, пока не выполнится оператор, сим-  
волизирующий окончание алгоритма.  
В результате будет получена строка Аi1Аi2Аi3...АikАi(k+1)..., являющая-  
ся значением матричной схемы для последовательности наборов ∆.  
Равносильная заданной ЛСА U2 матричная схема U1 может быть полу-  
чена элементарным построением системы формул перехода S1 с последую-  
щим занесением логических функций в ячейки матрицы (с перечислением  
через символ дизъюнкции, если потребуется).

Пример:



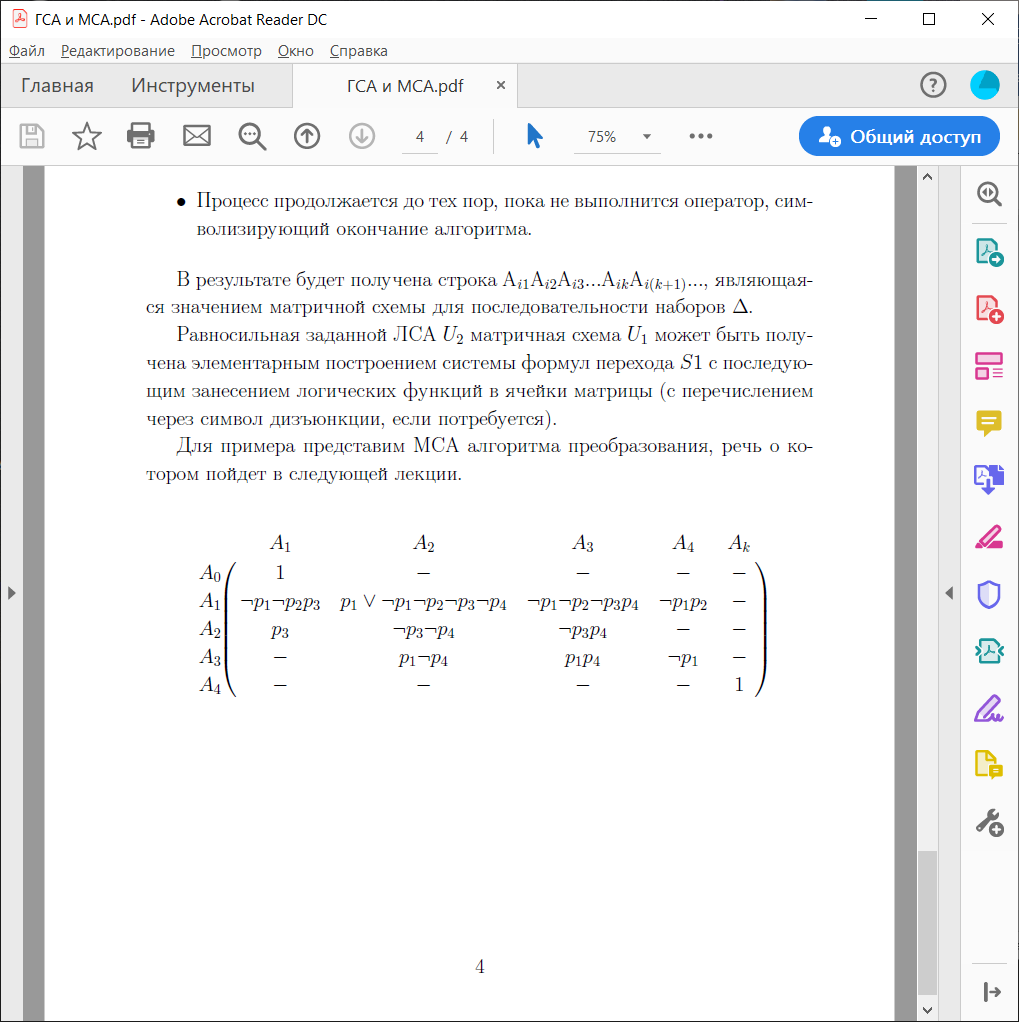
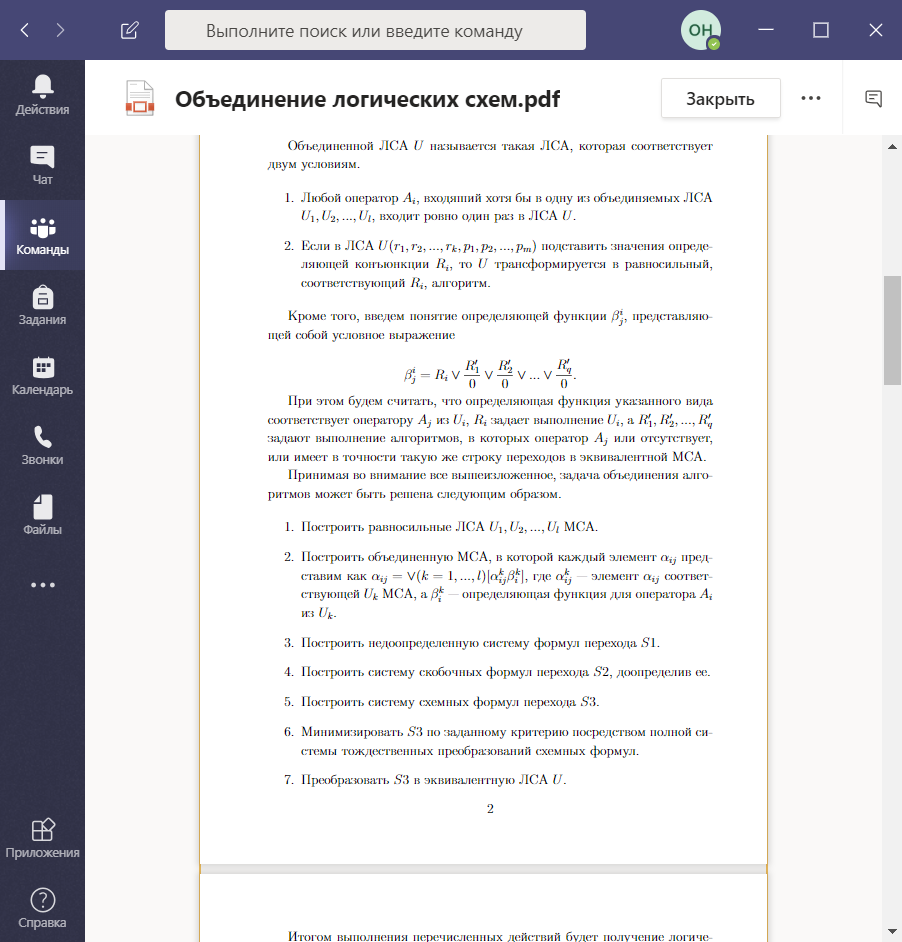
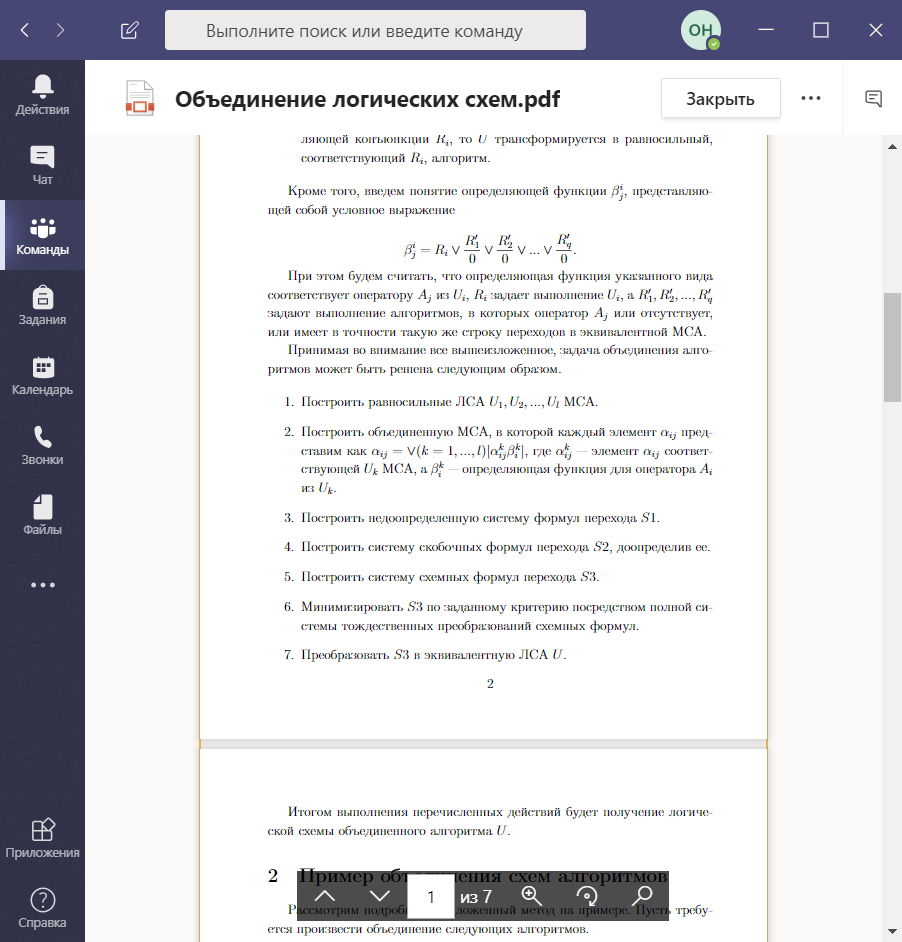


Рис.2 - МСА

**30. Объединение схем алгоритмов. Определяющие конъюнкции. Кодирование схем.**

* Нередко возникает необходимость в реализации устройства, поддержи-  
  вающего выполнение сразу нескольких алгоритмов. Классическим приме-  
  ром подобной задачи является разработка арифметико-логического устрой-  
  ства (АЛУ) процессора ЭВМ. АЛУ должно иметь возможность осуществ-  
  лять как стандартные арифметические (сложение, вычитание, умножение,  
  деление, возведение в степень и так далее), так и элементарные логические  
  операции (например, сравнение чисел). Каждой из перечисленных опера-  
  ций соответствует своя последовательность действий, а все устройство в  
  целом должно обеспечивать выполнение каждого из алгоритмов.  
  В частном случае, объединение может быть произведено путем элемен-  
  тарного выбора конкретного алгоритма перед началом работы: «Если тре-  
  буется осуществить операцию O1, то следует использовать алгоритм U1,  
  если же необходимо осуществить операцию O2, то следует использовать  
  алгоритм U2, и так далее». Однако в более общем случае, такой тривиаль-  
  ный подход приведет к возрастанию накладных расходов(в примере с АЛУ  
  — к увеличению аппаратных затрат) в следствие многократной реализации  
  общих для нескольких алгоритмов фрагментов. Следовательно, для прак-  
  тического применения больше подойдет техника объединения, способная  
  учитывать повторяющиеся последовательности действий.  
  Более формально, пусть имеются алгоритмы U1, U2, ..., Ul  
  , заданные в  
  форме ЛСА, и требуется получить некоторый минимальный по некоторо-  
  му критерию алгоритм U, который, при определенных дополнительных  
  условиях, мог бы быть равносилен любому из U1, U2, ..., Ul  
  .  
  В роли дополнительных могут использоваться специальные логические  
  условия r1, r2, ..., rk, применяемые в схеме U наряду с p1, p2, ..., pm таким  
  образом, чтобы каждая из возможных их конъюнкций R1, R2, ..., R2  
  k соот-  
  ветствовала не более чем одному алгоритму из U1, U2, ..., Ul  
  .  
  Определение. Конъюнкции R1, R2, ..., R2  
  k , сформированные из дополни-  
  тельных условий r1, r2, ..., rk, сопоставленные алгоритмам U1, U2, ..., Ul  
  ,  
  называются определяющими конъюнкциями.  
  Очевидным образом, оптимальным значением k будет такое минимальное, что 2  
  k > l.  
  Объединенной ЛСА U называется такая ЛСА, которая соответствует  
  двум условиям.  
  1. Любой оператор Ai  
  , входяший хотя бы в одну из объединяемых ЛСА  
  U1, U2, ..., Ul  
  , входит ровно один раз в ЛСА U.  
  2. Если в ЛСА U(r1, r2, ..., rk, p1, p2, ..., pm) подставить значения опреде-  
  ляющей конъюнкции Ri  
  , то U трансформируется в равносильный,  
  соответствующий Ri  
  , алгоритм.
* Кроме того, введем понятие определяющей функции βij  
  , представляю-  
  щей собой условное выражение

  
  
.  
При этом будем считать, что определяющая функция указанного вида  
соответствует оператору Aj из Ui  
, Ri задает выполнение Ui, а R’1, R’2, ..., R’q  
задают выполнение алгоритмов, в которых оператор Aj или отсутствует,  
или имеет в точности такую же строку переходов в эквивалентной МСА.  
Принимая во внимание все вышеизложенное, задача объединения алго-  
ритмов может быть решена следующим образом.  
1. Построить равносильные ЛСА U1, U2, ..., Ul МСА.  
2. Построить объединенную МСА, в которой каждый элемент αij пред-

  
3. Построить недоопределенную систему формул перехода S1.  
4. Построить систему скобочных формул перехода S2, доопределив ее.  
5. Построить систему схемных формул перехода S3.  
6. Минимизировать S3 по заданному критерию посредством полной си-  
стемы тождественных преобразований схемных формул.  
7. Преобразовать S3 в эквивалентную ЛСА U.

Итогом выполнения перечисленных действий будет получение логиче-  
ской схемы объединенного алгоритма U.

Соседнее кодирование – для всех пар схем считаем количество переходов между ними ( берем 2 схемы и смотрим построчно совпадение переходов в матричных схемах). Первой схеме сопоставляем произвольную конъюнкцию. Выбираем схему, максимально похожую на первую. Сопоставляем ей конъюнкцию, которая отличается только в 1 элементе. Выбираем из оставшихся схему,максимально похожую на те конъюнкция которых уже была сопоставлена. И тд.

**31. Определяющие функции. Процесс доопределения.**

Определяющие функции. Процесс доопределения

C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image018.png

C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image019.png

Где i - номер оператора

j - номер алгоритма

Либо эта функция определяет алгоритмы, в которых выполняется данный переход.

Процесс доопределения - выбор оптимального кодового набора для представления перехода к оператору данного алгоритма, исходя из определяющей конъюнкции. (Проще говоря, выбор оптимального набора доп условий r)

**32. Алгоритмически неразрешимые проблемы. Примеры.**

**Алгоритмически неразрешимые проблемы -** задачи, для которых нет эффективного алгоритма (дающего результат и не бесконечного)

Неразрешимость связана с невозможностью реализовать алгоритм решения некоторой(конкретной?) задачи на машине тьюринга. Либо с проблемой массовости. Когда алгоритм не может быть применим к классу задач, но применим к конкретной задаче из класса. Либо наоборот. Алгоритм применим к классу задач, кроме каких то случаев. Происходит нарушение свойства массовости алгоритма. Алгоритм можно составить для любой задачи, но для класса задач такой алгоритм придумать не удается.

**Теорема.** Проблем распознавания самоприменимости алгоритма является неразрешимой.

Рассмотрим вариант универсальной МТ, когда на ленте изображен ее собственный шифр в алфавите машины. Естестевенно считать, что если МТ обрабатывает этот шифр – она самоприменима, иначе – несамоприменима.

Предположим, что такая МТ, анализирующая собственный шифр и выносящая решение о самоприменимости, существует. Тогда в этой МТ всякий самоприменимый шифр перерабатывается в некоторый символ «вэ» (положительный ответ), а всякий несамоприменимый шифр в символ «тэ» (отрицательный ответ).

При таком допущении можно построить такую МТ’, которая перерабатывает несамоприменимые шифры «тэ», но неприменима к саприменимым шифрам. Это достигается введением такого изменения таблицы соотвествия, когда после появления символа «вэ» вместо остановки МТ’ бесконечно повторяла бы «вэ», «вэ», «вэ», …

Итак, МТ’ применима ко всякому несамоприменимому шифру и неприменима к самоприменимым шифрам. Это приводит к противоречию.

1. Пусть МТ’ самоприменима, то есть она применима к собственному шифру МТ’ и останавливается, но тогда она несамоприменима, так как вырабатывает «тэ».
2. Пусть МТ’ несамоприменима, что означает ее применимость к её собственному шифру МТ’, то есть она самоприменима.

Указанное противоречие и есть доказательство теоремы, так как оно опровергает утверждение о существовании МТ.

Из этой теоремы вытекает алгоритмическая неразрешимость более широкой проблемы – проблемы распознавания применимости, заключающейся в необходимости установить применимость любой МТ к любой заданной конфигурации.

Определение. Ассоциативным исчислением называется совокупность всех слов в некотором алфавите А с конечной системой подстановок.

Чтобы задать ассоциативное исчисление достаточно указать алфавит А и систему подстановок. Подстановки бывают вида « p< - >q » или «p - >q», где p, q – слова в алфавите А. Неориентированная подстановка позволяет замену в прямом и обратном направлениях, ориентированная – лишь в прямом. Допустим, задан алфавит А = (a, b, c) и одна подстановка «ab<->bcb», а также p = abcbcbab. Заменим вхождения ab: bcbcbcbbcb, а затем используем подстановку в обратном направлении: abcbcbab->aabcbab->aaabab.

Если слово r может быть получено из слова s применением лишь одной подстановки, то говорят, что r и s смежные слова. Если слово r может быть получено из слова s и r применением конечного числа подстановок, то говорят, что смежные слова образуют «дедуктивную цепочку от s к r». Наконец, слова s и r называются «эквивалентными», если существует дедуктивная цепочка от s и r.

Для отношения эквивалентности слов справедливо следующее.

1. Рефлексивность
2. Симметричность
3. Транзитивность

Проблема эквивалентности в ассоциативном исчислении состоит в необходимости определения для любых 2 слов в данном ассоциативном исчислении эквивалентны они или нет.

**Определение.** Логика является полной, если в рамках её можно доказать истинность или ложность каждого утверждения.

**Определение.** Логика является непротиворечивой, если она свободна о противоречий (например, в ней нельзя получить одновременно истинное и ложное утверждения).

**Теорема.** Каждая адекватная непротиворечивая арифметическая логика неполна.

Математический смысл теоремы в том, что в арифметической логике существуют истинные утверждения о целых положительных числах, которые нельзя вывести и доказать средствами этой логики.

**Алгоритмически неразрешимая задача -** задача, для которой не существует алгоритма, который получив входные данные, останавливался и выдавал бы ответ.

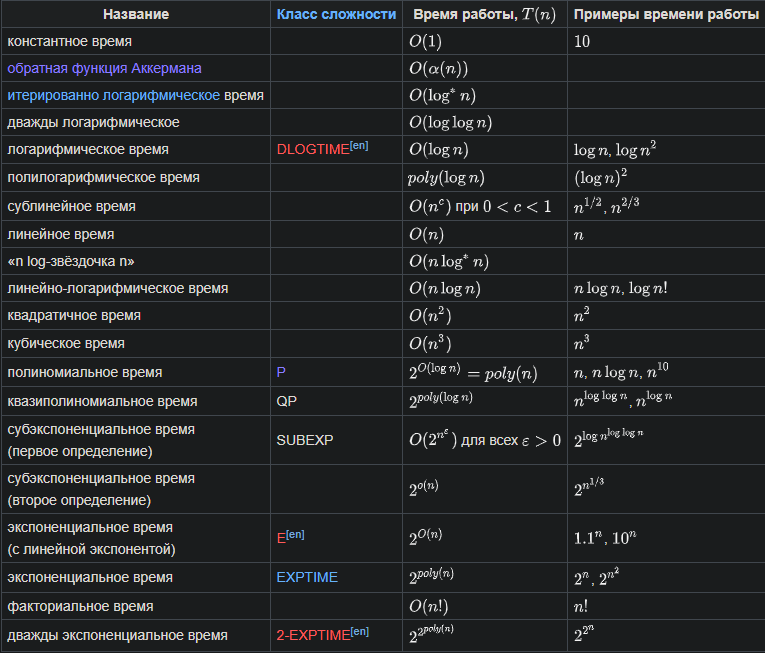
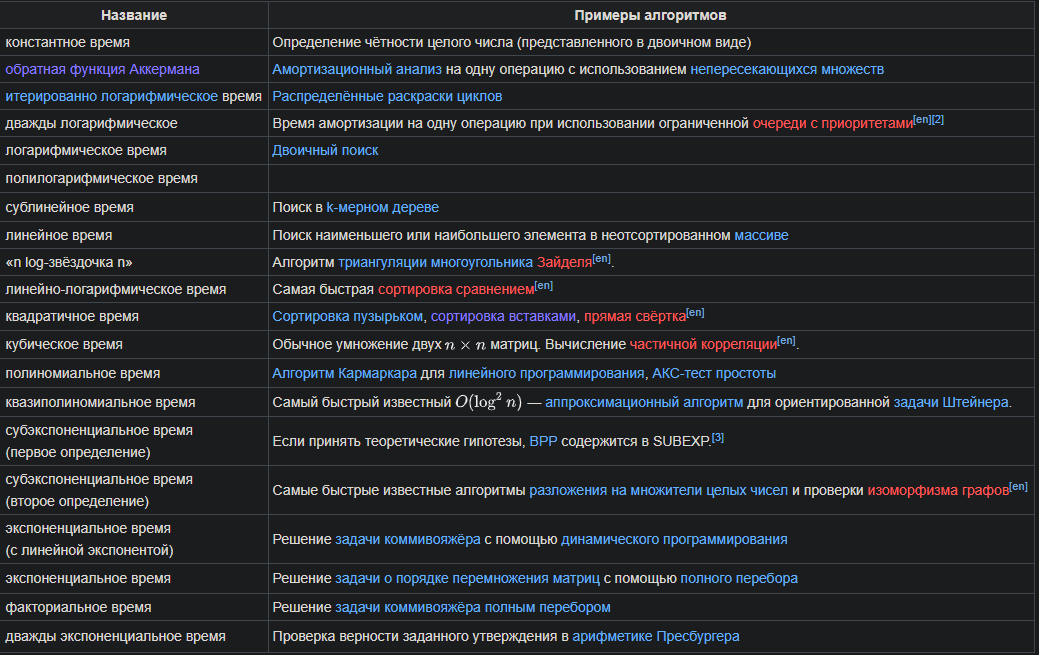
**Пример** проблемы - проблема останова. Проблема определения завершаема работа машины или нет. Ведь если машина не остановилась, это не значит что она не остановится потом. Может быть мы мало ждали.

Для конкретного алгоритма можно провести анализ и узнать, завершаема ее работа или нет. Но нет универсального алгоритма который определял завершаемость любого алгоритма.

В проблеме останова мы же можем взять и провести анализ самостоятельно. Применяя разные подходы. И в конце определить остановится машина или нет. По сути мы же делаем некую последовательность действий по некоторым правилам. Это уже некий алгоритм. Но для каждого случая он свой. Проблема поиска наименьшего корня. Мы вводили функцию наименьшего корня в МБР. Для конкретной функции мы можем провести анализ, но единого универсального алгоритма для всех функций не существует.

**33. Трудноразрешимые проблемы. Примеры.**

Это задачи, у которых нет алгоритмов решения с полиномиальной асимптотической оценкой (где a> 0).

(<https://ru.wikipedia.org/wiki/Временная_сложность_алгоритма#Экспоненциальное_время>)  


*Комментарий: асимптотическая оценка подразумевает оценивание времени (в операциях) и представляет собой временную функцию O(f(n)) от объёма входных данных n.*

**Примеры** трудноразрешимых задач:

* + **Задача независимого множества** (найти максимальный подграф любые две вершины которого не были бы соединены)
  + **Пятнашки**. Решается полным перебором всех возможных движений костяшек.
  + **Задача выполнимости булевых формул (задача SAT или ВЫП).** Можно ли в формуле найти комбинацию переменных такую, при которой она обратилась бы в истину. Формулы записаны в конъюнктивной форме. Задача решается перебором переменных.

**34. Трудноразрешимые проблемы теории графов.**

**Задача о поиске клики**. (Число вершин графа/подграфа, каждая из которых связана с каждой. Пример такого графа например что-то типа треугольника. Каждая вершина соединена с каждой вершиной). Суть в поиске максимальной клики. Жизненный пример, формирование некоторой команды. Работа будет продуктивной если все будут друг с другом знакомы.

* + - **Задача коммивояжера**. Задача о поиске выгодного маршрута между графами, посещая каждую вершину один раз.   
      *Комментарий: Гамильтонов цикл – цикл (замкнутый путь), который проходит через каждую вершину ровно один раз. В этот цикл входят все вершины графа.*
    - **Задача о раскраске графа**. Раскрасить вершины графа в минимальное количество цветов так, чтобы не было двух одинаковых смежных цветов.
    - **Вершинное покрытие**. Минимальное количество вершин, чтобы все ребра графа были соединены хотя бы с одной вершиной.

**35. Подходы к решению трудноразрешимых задач.**

**Трудноразрешимые проблемы** - такие задачи, для которых не существует алгоритмов, имеющих полиномиальную асимптотическую оценку. Полином n^100500.

Хуже полинома экспоненциальная оценка n!, 2^n.

Задачи имеющие экспоненциальную оценку - трудноразрешимы так как для решения этих задач требуются большие временные затраты.

Примеры трудноразрешимых задач:

* + Задача коммивояжера (теория графов. Обойти все вершины графа, побывав в каждой только 1 раз. Гамильтонов цикл. Покрыть)
  + Задача раскраски графа (Определить минимальное число цветов, в которое надо раскрасить граф, чтобы соседние вершины были разных цветов)
  + Поиск клика (каждая вершина подграфа соединена с каждой вершиной этого подграфа)
  + Задача независимого множества (найти максимальный подграф (суммарная стоимость максимально возможная) любые две вершины которого не были бы соединены ребром)
  + Задача о вершинном покрытии
  + Пятнашки. Решается полным перебором всех возможных движений костяшек.
  + Задача выполнимости булевых формул (задача SAT).Можно ли в формуле найти комбинацию переменных такую, при которой она обратилась бы в истину. Формулы записаны в конъюнктивной форме. Задача решается перебором переменных.

Знание этих примеров нужно, чтобы понимать, сводится ли задача к трудноразрешимым.

Трудноразрешимые проблемы теории графов.

* + - Задача коммивояжера. Задача о поиске выгодного маршрута между графами, посещая каждую вершину один раз. Гамильтонов цикл
    - Задача о раскраске графа. Раскрасить вершины графа в минимальное количество цветов так, чтобы не было двух одинаковых смежных цветов. (Планарный граф, прим: географическая карта)
    - Вершинное покрытие. Минимальное количество вершин, чтобы все ребра графа были соединены хотя бы с одной вершиной.
    - Задача клике. (Найти максимальный по размерности подграф т.е. зафиксировать некоторое подмножество, таким образом, чтобы это подмножество вершин образовывало полный подграф т.е. любая пара из этого подмножества вершин была соединена ребром.[Кликой](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B8%D0%BA%D0%B0_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)) в неориентированном [графе](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) называется подмножество вершин, каждые две из которых соединены ребром графа)

Подходы к решению трудноразрешимых задач.

* 1. Задача коммивояжера.
     + Полный перебор. Действие довольно очевидное, но ввиду высокой сложности алгоритма не очень оптимальный
     + Метод ближайшего соседа. Эвристический метод решения задачи коммивояжера. Относится к категории жадных алгоритмов. Заключается в том, что следующий пункт для обхода выбирается ближайшим. Так как алгоритм является эвристическим, и основан на жадности, то не факт, что полученное нами решение будет оптимальным или вообще НЕ худшим.
     + Метод имитации отжига. Относится к стохастическим алгоритмам. По сути в этом подходе мы работаем со случайными наборами. Относится к алгоритмам Монте Карло.
       1. Берем некое начальное значение
       2. Генерируем новое
       3. Если оно лучше, берем, если хуже, то считаем вероятность перехода в это состояние. Чем выше "температура"(вероятность), тем выше шанс перейти.
       4. Дальше либо понижаем температуру, либо ищем новое кандидатное состояние.
     + Муравьиный алгоритм. Моделирует поведение муравьев в колонии. Муравьи способны находить кратчайший путь до пищи, оставляя следы феромонов. На каждом шаге муравей вероятностно выбирает путь. Так же каждое ребро графа имеет некую интенсивность феромонов. При проходе по ребру, он оставляет след, обратно пропорциональный длине ребра. Со временем муравьи могут найти приближенно оптимальное решение.
  2. Задача раскраски графа
     + Полный перебор
     + Жадный алгоритм. Вершины упорядочиваются. Каждой вершине последовательно присваивается наименьший доступный цвет, который не используется для окраски соседней, либо добавляется новый цвет.

* + Эвристические алгоритмы. Алгоритм решения задачи, правильность которого для всех случаев не доказана, но он дает хорошее решение в данном случае. Эвристика не дает лучшего решения, не гарантирует нахождения решения вообще и может дать неверное решение.
  + Алгоритм локального поиска. Это группа алгоритмов. В них поиск ведется только на основании текущего состояния, ранее пройденные не учитываются и не запоминаются. Метод поиска более оптимального решения в окрестности некоторого текущего решения.
  + Метод ветвей и границ. Заключается в разбиении рассматриваемого множества с отсевом решений, заведомо не содержащих оптимальных решений.

**36. Асимптотическая оценка сложности алгоритмов. Верхняя, средняя и нижняя оценки.**

Зачем анализировать: чтобы оценить качественные и количественные свойства алгоритма, что позволит сравнивать алгоритмы между собой.

Критерии:

* Кол-во переданных по сети данных
* Требования к аппаратному обеспечению
* *Временные затраты//базовый*
* *Затраты памяти// базовый*
* Объём вспомогательных данных
* И т.д.

На практике наиболее часто оценивают именно временную сложность алгоритма

* В худшем случае (поиск отсутствующей в БД информации)
* В среднем (чаще всего, для случайных данных)

Сложности при среднем случае

* Средний случай-математическое ожидание времени работы (требуется вероятностный анализ с применением серьёзного мат. аппарата)
* Оценка рандомизированных алгоритмов требует нетривиального мат. Аппарата

«Оценить» порядок роста –значит найти такую функцию, которая бы с ростом значения аргумента увеличивалась точно также, как и время работы алгоритма с увеличением входных данных.

**Асимптотическая оценка(эффективность)** - порядок роста временных затрат в пределе (при бесконечном увеличении входных данных). При выборе подходящей функции можно пренебречь константами и членами с меньшей скоростью роста.

**Асимптотически точная оценка θ(g(n))-оценка в среднем.**

Для некоторой f(n) выражение f(n) = θ(g(n)) означает, что существует некоторые с1, с2 и n0, что 0 <=c1\*g(n)<=f(n)<=c2\*g(n) для всех n=>n0

**Верхняя оценка Оg(n)). -оценка в худшем случае**

Для некоторой f(n) выражение f(n) = О(g(n)) означает, что существует некоторые с и n0, что 0<= f(n) <=c\*g(n) для всех n=>n0

**Нижняя асимптотическая оценка Ω(g(n))-в лучшем случае**

Для некоторой f(n) выражение f(n) = **Ω** (g(n)) означает, что существует некоторые с и n0, что 0<=c\*g(n) <= f(n) для всех n=>n0

Сложности бывают:

* + В худшем случае(О). В основном используется эта оценка ввиду возможно некой простоты ее нахождения и вероятностей худшего случая.
  + В среднем(**θ**). Сложность алгоритма на случайных наборах данных. Это сложность усредненная по всем возможным входным данным.
  + Может применяться в случаях:
    - Когда у нас может быть трудноразрешимая задача в худшем случае, но этот случая очень редкий и мы чаще имеем другие данные, которые дают среднюю сложность.
    - Позволяет выделять более эффективный алгоритм среди других. Например, быстрая сортировка в худшем и дает n^2, но в среднем дает nlogn.
  + В лучшем случае(**Ω**). Нужна для поиска средней.

Строгие оценки

* Верхняя о(g(n)).

Для некоторой f(n) выражение f(n) = о(g(n)) означает, что для любой положительной константы с существует n0,>0, что 0<= f(n) <=c\*g(n) для всех n=>n0

* Нижняя ω(g(n)).

Для некоторой f(n) выражение f(n) = ω (g(n)) означает, что для любой положительной константы с существует n0,>0, что 0<=c\*g(n) <f(n) для всех n=>n0

**37. Асимптотическая оценка сложности алгоритмов. Амортизированная оценка.**

**. Асимптотическая оценка алгоритма** – оценка порядка роста\* временных затрат в пределе (при бесконечном увеличении входных данных). Именно поэтому при выборе подходящей функции *можно* пренебречь *константами и членами с меньшей скоростью роста*

*\*оценить порядок роста – значит найти такую функцию, которая бы с ростом значения аргумента увеличивалась точно так же, как и время работы с увеличением входных данных.*

Асимптотически точная оценка обозначается как O(g(n)).

Для некоторой f(n) выражение f(n)=O(g(n)) означает, что существуют некоторые с1,c2 и n0, что 0<=c1\*g(n)<=f(n)<=c2\*g(n) для всех n>=n0

Верхняя асимптотическая оценка обозначается как O(g(n)) – оценка худшего случая

Для некоторой f(n) выражение f(n)=O(g(n)) означает, что существуют некоторые c иn0, что

0<= f(n)<= c\*g(n) для всех n>=n0

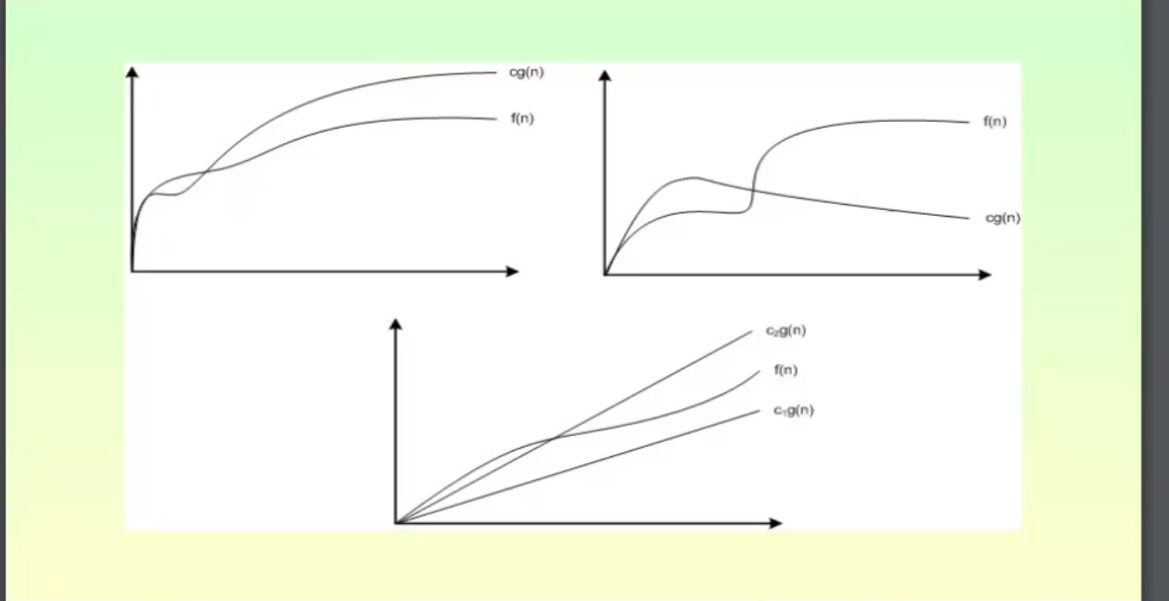
Нижняя асимптотическая оценка обозначается как Ω(g(n)) – оценка лучшего случая

PS-символ читается как «омега»

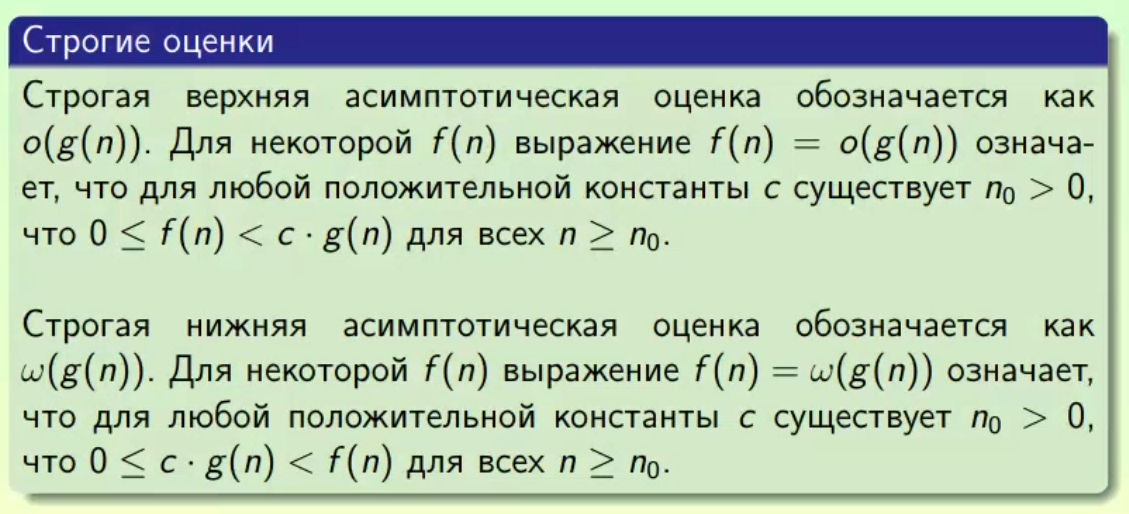
Для некоторой f(n) выражение f(n)=Ω(g(n)), что означает, что существуют некоторые с и n0, что

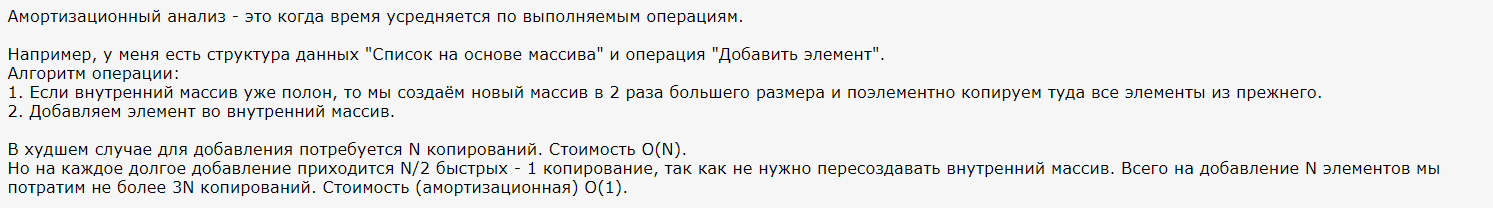
0<=c\*g(n)<=f(n) для всех n>=n0

Нижняя оценка нужна для поиска «средней» оценки. Суть нахождения средней оценки заключается в том, что мы по чуть-чуть опускаем верхнюю и поднимаем нижнюю оценку до тех пор, пока они не сойдутся – это и будет средняя оценка.



Левый верхний график – худший случай. Правый верхний – лучший случай. Нижний – средний случай.

Также в этот пункт включаются: 



Ссылка для подробного изучения амортизированной оценки/анализа, в лекции не было: <https://web.archive.org/web/20090810141015/http://rain.ifmo.ru/cat/view.php/theory/algorithm-analysis/amortized-analysis-2004>

**38. Оценка рекурсивных алгоритмов. Основная теорема.**

**Формула:**

C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image020.png,

Где а - количество подзадач в рекурсии,

n/b - размер каждой подзадачи (Предполагается, что все подзадачи на каждом этапе имеют одинаковый размер.),

f(n) - оценка сложности работы алгоритма без рекурсии. В неё также включается вычислительная стоимость деления на подзадачи и объединения результатов решения подзадач.

* 1. C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image021.png
  2. C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image022.png
  3. C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image023.png

**Суть теоремы**.

C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image024.png

C:\CFA1C625\FE9B1FA5-A704-4C56-BAFF-45DA17AF8B00.files\image025.png

* Если одного порядка, то случай два
* Если больше, то третий случай.

Причем в первом и третьем случаях "зазор" должен быть размера n^e

По сути у нас f(n)=n^c и мы это некоторое с сравниваем с логарифмом. Причем разность видимо должна быть на целое число.

**Разбор теоремы:**

берем алгоритм, смотрим сколько есть вызовов, это будет a, смотрим на сколько частей разбивается число элементов, это будет b.

Убираем рекурсию, считаем сложность остальной части. Это будет f(n).

Считаем Логарифм a по основанию b.  
f(n) представляет собой n^c. Сравниваем c и логарифм.

Ситуации сравнения:

1. С меньше чем логарифм - первый случай
2. С равен логарифму - второй случай
3. С больше логарифма - третий случай

Также см. википедию. Там примеры и выражения, не решаемые основной теоремой.

<https://ru.wikipedia.org/wiki/Основная_теорема_о_рекуррентных_соотношениях>

**39. Оценка рекурсивных алгоритмов. Метод деревьев рекурсии.**

Рекурсивные алгоритмы относятся к классу алгоритмов с высокой ресурсоемкостью, так как при большом количестве самовызовов рекурсивных функций происходит быстрое заполнение стековой области. Кроме того, организация хранения и закрытия очередного слоя рекурсивного стека являются дополнительными операциями, требующими временных затрат. На трудоемкость рекурсивных алгоритмов влияет и количество передаваемых функцией параметров.

Рассмотрим один из методов анализа трудоемкости рекурсивного алгоритма, который строится на основе подсчета вершин рекурсивного дерева. Для оценки трудоемкости рекурсивных алгоритмов строится **полное дерево рекурсии**. Оно представляет собой *граф*, вершинами которого являются наборы *фактических параметров* при всех вызовах функции, начиная с первого обращения к ней, а *ребрами* – пары таких наборов, соответствующих взаимным вызовам. При этом вершины дерева рекурсии соответствуют фактическим вызовам рекурсивных функций. Следует заметить, что одни и те же наборы параметров могут соответствовать разным вершинам дерева. *Корень полного дерева рекурсивных вызовов* – это *вершина* полного дерева рекурсии, соответствующая начальному обращению к функции.

Важной характеристикой рекурсивного алгоритма является **глубина рекурсивных вызовов** – наибольшее одновременное количество рекурсивных обращений функции, определяющее максимальное количество слоев рекурсивного стека, в котором осуществляется хранение отложенных вычислений. Количество элементов полных рекурсивных обращений всегда не меньше глубины рекурсивных вызовов. При разработке рекурсивных программ необходимо учитывать, что глубина рекурсивных вызовов не должна превосходить максимального размера стека используемой вычислительной среды.

При этом **объем рекурсии** - это одна из характеристик сложности *рекурсивных вычислений* для конкретного набора параметров, представляющая собой количество вершин полного рекурсивного дерева без единицы.

Будем использовать следующие обозначения для конкретного входного параметра D:

R(D) – общее число вершин дерева рекурсии,

RV(D) – объем рекурсии без листьев (*внутренние вершины*),

RL(D) – количество *листьев дерева* рекурсии,

HR(D) – глубина рекурсии.

Например, для вычисления n -го члена последовательности Фибоначчи разработана следующая *рекурсивная функция*:

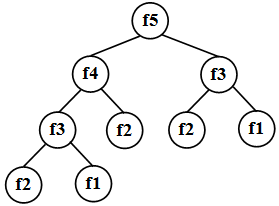
intFib(int n){ //n – номер члена последовательности

if(n<3) return 1; //база рекурсии

returnFib(n-1)+Fib(n-2); //декомпозиция

}

Тогда полное *дерево* рекурсии для вычисления пятого члена последовательности Фибоначчи будет иметь вид ([рис. 34.1](https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=1#image.34.1)):



**Рис. 34.1.**Полное дерево рекурсии для пятого члена последовательности Фибоначчи

Характеристиками рассматриваемого метода оценки алгоритма будут следующие величины.

|  |  |
| --- | --- |
| **D = 5** | **D = n** |
| R(D)=9 | R(D)=2fn-1 |
| RV(D)=4 | RV(D)=fn-1 |
| RL(D)=5 | RL(D)=fn |
| HR(D)=4 | HR(D)=n-1 |

***Пример 1****. Задача о разрезании прямоугольника на квадраты*.

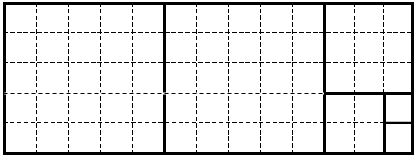
Дан *прямоугольник*, стороны которого выражены натуральными числами. Разрежьте его на минимальное число квадратов с натуральными сторонами. Найдите число получившихся квадратов.

Разработаем рекурсивную триаду.

*Параметризация*: m, n – *натуральные числа*, соответствующие размерам прямоугольника.

*База рекурсии*: для m=n число получившихся квадратов равно 1, так как данный *прямоугольник* уже является квадратом.

*Декомпозиция*: если m \ne  n, то возможны два случая m < n или m > n. Отрежем от прямоугольника наибольший по площади квадрат с натуральными сторонами. *Длина* стороны такого квадрата равна наименьшей из сторон прямоугольника. После того, как квадрат будет отрезан, размеры прямоугольника станут следующие: большая сторона уменьшится на длину стороны квадрата, а меньшая не изменится. Число искомых квадратов будет вычисляться как число квадратов, на которые будет разрезан полученный *прямоугольник*, плюс один (отрезанный квадрат). К получившемуся прямоугольнику применим аналогичные рассуждения: проверим на соответствие базе или перейдем к *декомпозиции* ([рис. 34.2](https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=1#image.34.2)).



**Рис. 34.2.**Пример разрезания прямоугольника 13x5 на квадраты

#include "stdafx.h"

#include <iostream>

using namespace std;

int kv(int m,int n);

int \_tmain(int argc, \_TCHAR\* argv[]) {

inta,b,k;

printf("Введите стороны прямоугольника->");

scanf("%d%d",&a,&b);

k = kv(a,b);

printf("Прямоугольник со сторонами %d и %d можно разрезать

на %d квадратов",a,b,k);

system("pause");

return 0;

}

intkv(intm,intn){ //m,n – стороныпрямоугольника

if(m==n) return 1; //базарекурсии

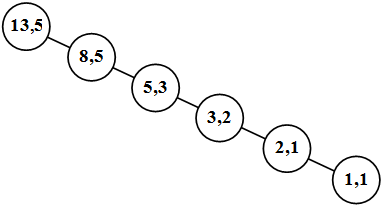
if(m>n) return 1+kv(m-n,n); //декомпозиция для m>n

return 1+kv(m,n-m); //декомпозиция для m<n

}

Характеристиками рассматриваемого метода оценки алгоритма будут следующие величины ([рис. 34.3](https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=1#image.34.3)).

|  |  |
| --- | --- |
| **D = (13, 5)** | **D = (m, n), m \ge n, худший случай** |
| R(D)=6 | R(D)=m |
| RV(D)=4 | RV(D)=m-2 |
| RL(D)=1 | RL(D)=1 |
| HR(D)=6 | HR(D)=m |



**Рис. 34.3.**Пример полного дерева рекурсии для разрезания прямоугольника 13x5 на квадраты

***Пример 2****. Задача о нахождении центра тяжести выпуклого многоугольника*.

Выпуклый многоугольник задан на плоскости координатами своих вершин. Найдите его центр тяжести.

Разработаем рекурсивную триаду.

*Параметризация*: x, y – вещественные массивы, в которых хранятся *координаты* вершин многоугольника; n – это число вершин многоугольника, по условию задачи, n>1 так как минимальное число вершин имеет двуугольник (*отрезок*).

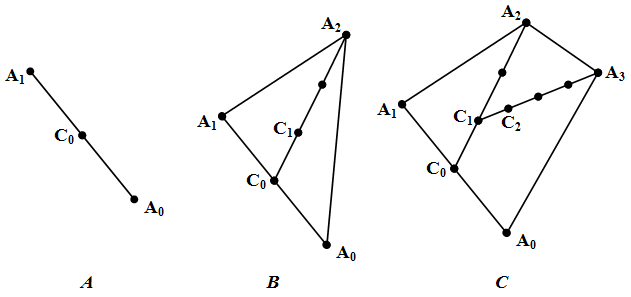
*База рекурсии*: для n=2 в качестве многоугольника рассматривается *отрезок*, центром тяжести которого является его середина ([рис. 34А](https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=2#image.34.4)). При этом середина делит *отрезок* в отношении 1 : 1. Если *координаты* концов отрезка заданы как (x0,y0) и (x1,y1), то *координаты* середины вычисляются по формуле:

cx=\frac{x_0+x_1}{2},\quad cy=\frac{y_0+y_1}{2}.

*Декомпозиция*: если n>2, то рассмотрим последовательное нахождение центров тяжести треугольника, четырехугольника и т.д.

Для n=3 центром тяжести треугольника является точка пересечения его медиан, которая делит каждую медиану в отношении 2 : 1, считая от вершины. Но *основание* *медианы* – это середина отрезка, являющегося стороной треугольника. Таким образом, для нахождения центра тяжести треугольника необходимо: найти центр тяжести стороны треугольника (отрезка), затем разделить в отношении 2 : 1, считая от вершины, *отрезок*, образованный основанием *медианы* и третьей вершиной ([рис. 34B](https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=2#image.34.4)).

Для n=4 центром тяжести четырехугольника является точка, делящая в отношении 3 : 1, считая от вершины, *отрезок*: он образован центром тяжести треугольника, построенного на трех вершинах, и четвертой вершиной ([рис. 34C](https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=2#image.34.4)).



**Рис. 34.4.**Примеры построения центров тяжести многоугольников

Таким образом, для нахождения центра тяжести n -угольника необходимо разделить в отношении (n-1): 1, считая от вершины, *отрезок*: он образован центром тяжести (n-1) -угольника и n -ой вершиной рассматриваемого многоугольника. Если концы отрезка заданы координатами вершины (xn,yn) и центра тяжести (n-1) -угольника (cxn-1,cyn-1), то при делении отрезка в данном отношении получаем *координаты*:

cx_n=\frac{x_n+(n-1)cx_{n-1}}{n},\quad cy_n=\frac{y_n+(n-1)cy_{n-1}}{n}

#include "stdafx.h"

#include <iostream>

using namespace std;

#define max 20

void centr(int n,float \*x, float \*y, float \*c);

int \_tmain(int argc, \_TCHAR\* argv[]){

int m, i=0;

FILE \*f;

if ( ( f = fopen("in.txt", "r") ) == NULL )

perror("in.txt");

else {

fscanf(f, "%d",&m);

printf("\n%d",m);

if( m< 2 || m >max ) //вырожденный многоугольник

printf ("Вырожденный многоугольник");

else {

float \*px,\*py,\*pc;

px = newfloat[m];

py = new float[m];

pc = new float[2];

pc[0] = pc[1] = 0;

while(i<m) {

fscanf(f, "%f %f",&px[i], &py[i]);

printf("\n%f %f",px[i], py[i]);

i++;

}

centr(m,px,py,pc);

printf ("\nЦентр тяжести имеет координаты:

(%.4f, %.4f)",pc[0],pc[1]);

delete [] pc;

delete [] py;

delete [] px;

}

fclose(f);

}

system("pause");

return 0;

}

void centr(int n,float \*x, float \*y, float \*c){

//n - количество вершин,

//x,y - координаты вершин,

//c - координаты центра тяжести

if(n==2){ //база рекурсии

c[0]=(x[0]+x[1])/2;

c[1]=(y[0]+y[1])/2;

}

if(n>2) { //декомпозиция

centr(n-1,x,y,c);

c[0]= (x[n-1] + (n-1)\*c[0])/n;

c[1]= (y[n-1] + (n-1)\*c[1])/n;

}

}

Характеристиками рассматриваемого метода оценки алгоритма будут следующие величины.

|  |  |
| --- | --- |
| **D = 4** | **D = n** |
| R(D)=3 | R(D)=n-1 |
| RV(D)=1 | RV(D)=n-3 |
| RL(D)=1 | RL(D)=1 |
| HR(D)=3 | HR(D)=n-1 |

Однако в данном случае для более достоверной оценки необходимо учитывать емкостные характеристики алгоритма.

***Пример 3****. Задача о разбиении целого на части*.

Найдите количество разбиений *натурального числа* на сумму натуральных слагаемых.

*Разбиение* подразумевает *представление* *натурального числа* в виде суммы натуральных слагаемых, при этом суммы должны отличаться набором чисел, а не их последовательностью. В *разбиение* также может входить одно число.

Например, *разбиение* числа 6 будет представлено 11 комбинациями:

6

5+1

4+2, 4+1+1

3+3, 3+2+1, 3+1+1+1

2+2+2, 2+2+1+1, 2+1+1+1+1

1+1+1+1+1+1

Рассмотрим решение в общем виде. Пусть зависимость R(n,k) вычисляет количество разбиений числа n на сумму слагаемых, не превосходящих k. Опишем свойства R(n,k).

Если в сумме все слагаемые не превосходят 1, то такое *представление* единственно, то есть R(n,k)=1.

Если рассматриваемое число равно 1, то при любом натуральном значении второго параметра *разбиение* также единственно: R(n,k)=1.

Если второй *параметр* превосходит *значение* первого , то имеет *место* *равенство* R(n,k)=R(n,n), так как для представления *натурального числа* в сумму не могут входить числа, превосходящие его.

Если в сумму входит слагаемое, равное первому параметру, то такое *представление* также единственно (содержит только это слагаемое), поэтому имеет *место* *равенство*: R(n,n)=R(n,n-1)+1.

Осталось рассмотреть случай (n>k). Разобьем все представления числа n на непересекающиеся разложения: в одни обязательно будет входить слагаемое k, а другие суммы не содержат k. Первая *группа* сумм, содержащая k, эквивалентна зависимости R(n-k,k), что следует после вычитания числа k из каждой суммы. Вторая *группа* сумм содержит *разбиение* числа n на слагаемые, каждое из которых не превосходит k-1, то есть число таких представлений равно R(n,k-1). Так как обе группы сумм не пересекаются, то R(n,k)=R(n-k,k)+R(n,k-1).

Разработаем рекурсивную триаду.

*Параметризация*: Рассмотрим *разбиение* *натурального числа* n на сумму таких слагаемых, которые не превосходят *натурального числа*k.

*База рекурсии*: исходя из свойств рассмотренной зависимости, выделяются два базовых случая:

при n=1     R(n,k)=1,

при k=1     R(n,k)=1.

*Декомпозиция*: общий случай задачи сводится к трем случаям, которые и составляют декомпозиционные отношения.

при n=k     R(n,k)=R(n,n-1)+1,

при n<k     R(n,k)=R(n,n),

при n>k     R(n,k)=R(n-k,k)+R(n,k-1).

#include "stdafx.h"

#include <iostream>

using namespace std;

unsigned long int Razbienie(unsigned long int n,

unsigned long int k);

int \_tmain(int argc, \_TCHAR\* argv[]){

unsigned long int number, max,num;

printf ("\nВведите натуральное число: ");

scanf ("%d", &number);

printf ("Введите максимальное натуральное слагаемое в

сумме: ");

scanf ("%d", &max);

num=Razbienie(number,max);

printf ("Число %d можно представить в виде суммы с

максимальным слагаемым %d.", number, max);

printf ("\nКоличество разбиений равно %d",num);

system("pause");

return 0;

}

unsigned long int Razbienie(unsigned long int n,

unsigned long int k){

if(n==1 || k==1) return 1;

if(n<=k) returnRazbienie(n,n-1)+1;

return Razbienie(n,k-1)+Razbienie(n-k,k);

}

*Пример 4. Задача о переводе натурального числа в шестнадцатеричную систему счисления*.

Дано *натуральное число*, не выходящее за пределы типа *unsigned* long. Число представлено в десятичной системе *счисления*. Переведите его в систему *счисления* с основанием 16.

Пусть требуется перевести *целое число* n из десятичной в р -ичную систему *счисления* (по условию задачи, р = 16), то есть найти такое k, чтобы выполнялось *равенство* n10=kp.

*Параметризация*: n – данное *натуральное число*, р – *основание* *системы счисления*.

*База рекурсии*: на основании правил перевода чисел из десятичной системы в систему *счисления* с основанием р, *деление* нацело на *основание* системы выполняется до тех пор, пока неполное частное не станет равным нулю, то есть: если целая часть частного n и р равна нулю, то k = n. Данное условие можно реализовать иначе, сравнив n и р: целая часть частного равна нулю, если n < р.

*Декомпозиция*: в общем случае k формируется из цифр целой части частного n и р, представленной в системе *счисления* с основанием р, и остатка от деления n на p.

#include "stdafx.h"

#include <iostream>

using namespace std;

#define maxline 50

void perevod( unsigned long n, unsigned int p,FILE \*pf);

int \_tmain(int argc, \_TCHAR\* argv[]){

unsigned long number10;

unsigned int osn=16;

char number16[maxline];

FILE \*f;

if ((f=fopen("out.txt", "w"))==NULL)

perror("out.txt");

else {

printf ("\nВведите число в десятичной системе: ");

scanf("%ld", &number10);

perevod(number10, osn, f);

fclose(f);

}

if ((f=fopen("out.txt", "r"))==NULL)

perror("out.txt");

else {

fscanf(f,"%s",number16);

printf("\n %ld(10)=%s(16)", number10, number16);

fclose(f);

}

system("pause");

return 0;

}

void perevod(unsigned long n, unsigned int p, FILE \*pf){

char c;

unsigned int r;

if(n >= p) perevod (n/p, p, pf);//декомпозиция

r=n%p;

c=r <10 ? char (r+48) : char (r+55);

putc(c, pf);

}

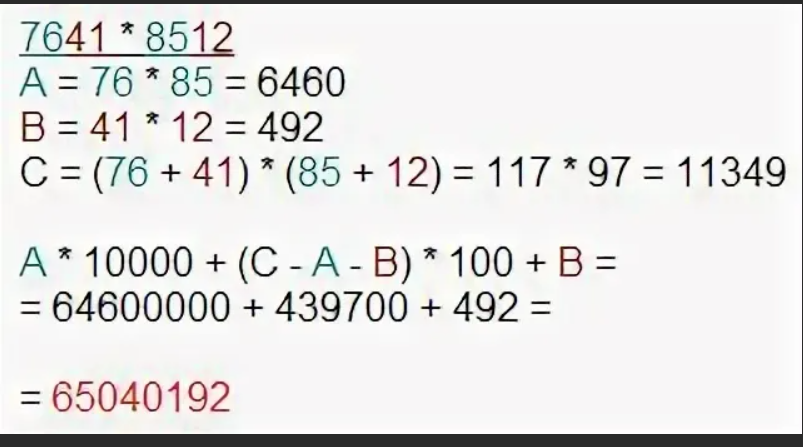
[**https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=1**](https://www.intuit.ru/studies/courses/648/504/lecture/11462?page=1)

[**http://www.tvd-home.ru/recursion**](http://www.tvd-home.ru/recursion)

**40. Типовые алгоритмические идеи. Разделяй и властвуй. Meet-in-the-middle.**

А) Алгоритм «разделяй и властвуй» предполагает, что мы делаем декомпозицию задачи, решаем подзадачи, затем синтезируем всё в общее решение. Примерами являются *алгоритм Быстрой сортировки* и *алгоритм сортировки Слиянием*.

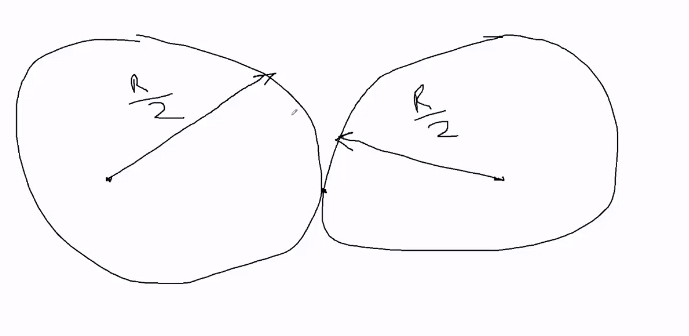
Также следует выделить *алгоритм Карацубы*. Он предназначен для перемножения двух полиномов.



Б) Meet-in-themiddle

Идея состоит в том, что задачу можно решать не только прямым ходом, т.е. сначала, но и с конца.

Предположим, что у нас есть начальное слово и конечное слово, а также N преобразований, позволяющих из начального слова получить конечное. Мы можем пойти как бы задом наперёд. Одновременно из начальной точки и конечной будем выполнять преобразования. При преобразованиях для каждого слова будут получаться так называемые пространства состояния. Если пространства состояния обоих слов «соприкоснутся» (см. рисунок ниже), значит, решение существует.



**41. Типовые алгоритмические идеи. Стохастические алгоритмы. Монте-Карло. Лас-Вегас.**

Иногда нас устраивает получение решения с некоторой вероятностью.

Делятся на 2 группы алгоритмов:

* 1. Монте Карло. Рандомизированный алгоритм, продолжительность которого детерминирована, но чья продукция может быть неправильной с определенной (типично маленькой) вероятностью. В данном алгоритме мы какое-то решение получим. Приближенное. Например, есть сложная фигура. Задача определить ее площадь. Можем определить так же лежит точка в фигуре или нет. Мы можем вписать ее в прямоугольник, накинуть сетку и посчитать количество точек внутри. Через отношение точек внутри к общему найдем приближенное значение площади.

Данный метод производит многократную генерацию случаев (моделей) возможных результатов. При обыгрывании таких моделей или ситуаций любая переменная (фактор), которому свойственна неопределенность, заменяется диапазоном значений — случайных величин, удовлетворяющих условию задачи. После чего выполняются многократные расчеты результатов, в каждом таком случае (итерации) происходит генерирование случайной величины и ситуация как-бы проигрывается снова. С одной стороны, производится многократное воспроизведение случая, что невозможно повторить в реальности, тем самым, вероятность становится относительно детерминированной т.е. появляется некоторая «средняя» ситуация, которую можно предсказать, а значит рассчитать ее числовые характеристики. С другой стороны, чтобы приблизиться к результату с заданной точностью нужно разыграть достаточное количество экспериментов. Резюмируя алгоритм, определим важнейшие шаги реализации:

* **Шаг 1.** Определяются области применения случайных величин в рамках задачи;
* **Шаг 2.** Разыгрывается случайная величина и используется в расчетах задачи;
* **Шаг 3.** Многократное повторение расчетов задачи и сохранение «успешных» результатов;
* **Шаг 4**. Оценка полученного решения.
  1. Лас Вегас. Н**икогда не возвращают неправильный ответ, хотя иногда они не возвращают вообще никакого ответа.** Например, генерация ОЧЕНЬ большого простого числа. Можно либо перебрать все числа, либо просто сгенерировать число и проверить его. И так продолжать пока не получим.

Оптимальный Лас-Вегас

Для того, чтобы сделать алгоритм LasVegas оптимальным, ожидаемое время работы должно быть сведено к минимуму. Это может быть сделано путем:

* Лас - Вегас алгоритм А (х) пробегает несколько раз для некоторого числа т 1 шагов. Если А (х) останавливается во время выполнения программы, то А (х) делается; в противном случае, повторить процесс с самого начала еще т 2 шагов, и так далее.
* Разработка стратегии , которая является оптимальной среди всех стратегий А (х), учитывая полную информацию о распределении Т А (х).

Существование оптимальной стратегии может быть увлекательным теоретическое наблюдение. Однако, это не практично в реальной жизни , потому что это не так легко найти информацию о распределении Т А (х). Кроме того, нет никакого смысла запускать эксперимент несколько раз , чтобы получить информацию о распределении , так как большую часть времени, ответ нужен

**42. Типовые алгоритмические идеи. Жадные алгоритмы. Динамическое программирование.**

Жадный алгоритм.

Жадный алгоритм, это алгоритм, который на каждом своём шаге (т.е. на локальном участке) выбирает оптимальное решение, что должно привести к глобальному оптимуму (верно для матроидов).

Общего критерия оценки применимости жадного алгоритма для решения конкретной задачи не существует, однако для задач, решаемых жадными алгоритмами, характерны две особенности: во-первых, к ним применим Принцип жадного выбора, а во-вторых, они обладают свойством Оптимальности для подзадач.

Говорят, что к оптимизационной задаче применим принцип жадного выбора, если последовательность локально оптимальных выборов даёт глобально оптимальное решение. В типичном случае доказательство оптимальности следует такой схеме:

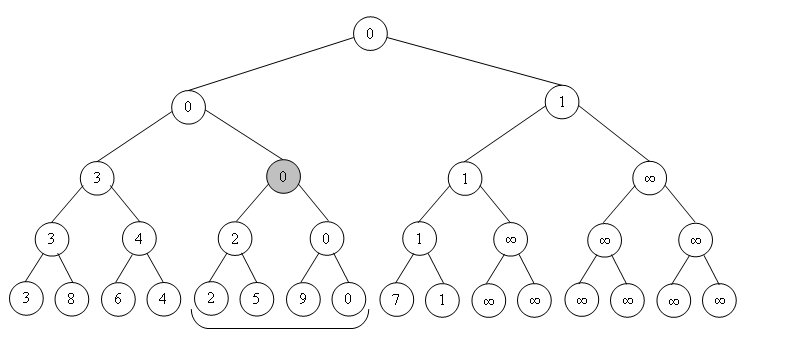
1. Доказывается, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не хуже первого.
2. Показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной.
3. Рассуждение завершается по индукции.

**43. Типовые алгоритмические идеи. Эвристические алгоритмы. Предпросчет.**

**Эвристический алгоритм** - алгоритм решения задачи, основанный на некотором предположении. Он не является гарантированно точным или правильным, но его достаточно для решения поставленной задачи. На некотором наборе не факт что мы будем получать ответ вовсе, а где то ответ может быть неверный. Например правило Ворсдорфа. Это правило для того, чтобы обойти все клетки шахматной доски конем, в каждой побывав один раз. Правило в том, что надо ходить в ту ячейку, из которой меньше всего продолжений. С применением этого правила мы можем обойти доску 250х250 за то же время, что и 8х8 обычным.

**Предпросчет.** Часть нам например надо отвечать на запросы. Причем мы должны отвечать на них быстро. Мы можем просчитать часть информации и использовать ее при формировании ответа. В таком случае мы сможем эту информацию как статическую. Например использование некоторой таблицы

Определение

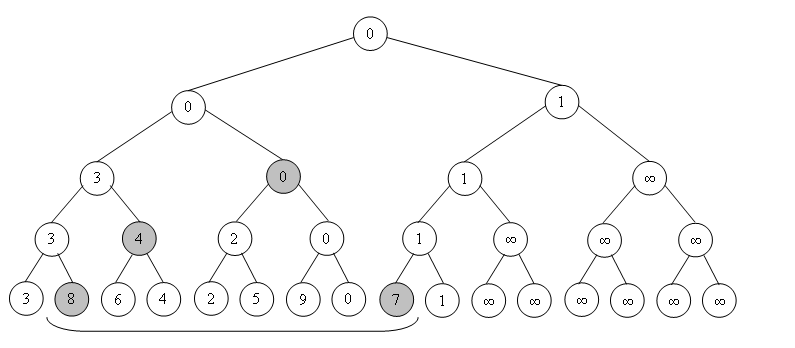
Введём понятие дерева отрезков. Для удобства дополним длину массива до степени двойки. В добавленные элементы массива допишем бесконечности (за бесконечностью стоит понимать, например, число, больше которого в данных ничего не появится). Итак, дерево отрезков это двоичное дерево, в каждой вершине которого написано значение заданной функции на некотором отрезке. Функция в нашем случае – это минимум.  
  
Каждому листу будет соответствовать элемент массива с номером, равным порядковому номеру листа в дереве. А каждой вершине, не являющейся листом, будет соответствовать отрезок из элементов массива соответствующих листам-потомкам этой вершины.  
  
  
  
За кажущейся монструозностью определения скрывается вполне простое понятие – обращаем взгляд на рисунок.  
  
  
  
Поясним определение. Выделенной вершине будет соответствовать отмеченный отрезок, потому как он является объединением всех листов-потомков данной вершины (начиная с этого момента отождествим лист и элемент массива, который он представляет).  
  
Хранить дерево будем подобно двоичной куче. Заведём массив T[2n – 1]. Корень будет лежать в первом элементе массива, а сыновья i-ой вершины будут лежать в элементах с номерами 2i и 2i + 1 – левый и правый соответственно. Сразу можно заметить очевидное свойство: T[i] = min(T[2i], T[2i + 1]) для i-ой вершины, не являющейся листом. Листы, к слову, будут лежать при такой нумерации в элементах с номерами от n до 2n – 1.

Построение

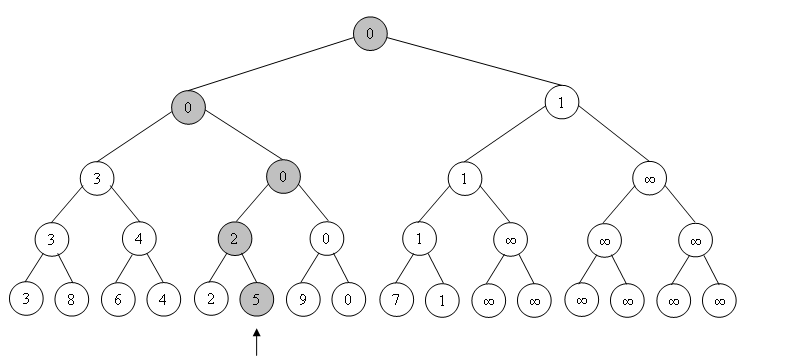
Построим дерево, пробежавшись по элементам с (n – 1)-ого по первый, считая минимум значений в сыновьях для каждой вершины.

Функция build\_tree(V) превращает массив V в дерево отрезков для этого массива. Итак, как теперь по отрезку найти минимум на нём? Для этого введём понятие фундаментального отрезка.

Запрос минимума

Назовём фундаментальным отрезком в массиве такой отрезок, что существует вершина в дереве, которой он соответствует. Разобьём наш отрезк на минимальное количество непересекающихся фундаментальных. Покажем, что на каждом уровне их количество не превосходит 2.  
  
  
  
Возьмём самый большой фундаментальный отрезок в разбиении. Пусть его длина – 2t. Заметим, что фундаментальных отрезков длиной 2t – не более двух (1). Возьмём самый левый из имеющихся максимальных фундаментальных. Будем двигаться от него налево. Заметим, опять же, что длины отрезков будут убывать (2). Так же и с правым из максимальных. Тем самым получим, что фундаментальных отрезков – не более 2t, что не превосходит 2logn. Пункты (1) и (2) в доказательстве я оставлю для самостоятельного осмысления.  
  
Чем нам это помогает? Теперь мы можем реализовать запрос минимума «снизу». Будем подниматься снизу, добавляя к ответу на каждом уровне, если надо, фундаментальный отрезок.  
  
Заведём два указателя – l и r, с помощью которых будем находить очередные фундаментальные отрезки разбиения. Изначально установим l и r указывающими на листы, соответствующие концам отрезка запроса. Заметим, что если l указывает на вершину, являющуюся правым сыном своего родителя, то эта вершина принадлежит разбиению на фундаментальные отрезки, в противном случае не принадлежит. Аналогично с указателем r – если он указывает на вершину, являющуюся левым сыном своего родителя, то добавляем её в разбиение. После этого сдвигаем оба указателя на уровень выше и повторяем операцию. Продолжаем операции пока указатели не зайдут один за другой.  
  
Находя очередной фундаментальный отрезок, мы сравниваем минимум на нём с текущим найденным минимумом и уменьшаем его в случае необходимости. Асимптотика работы алгоритма – O(logn), т. к. на каждом уровне мы выполняем константное число операций, а всего уровней – logn.

Модификация

Теперь научимся изменять значение элемента дерева. Заметим, что для каждого листа есть ровно logn фундаментальных отрезков, которые его содержат – все они соответствуют вершинам, лежащим на пути от нашего листа до корня.  
  
  
  
Значит, при изменении элемента достаточно просто пробежаться от его листа до корня и обновить значение во всех вершинах на пути по формуле T[i] = min(T[2i], T[2i + 1]).

**44. Типовые алгоритмические идеи. Приближенные алгоритмы. Структуры данных на основе хэш-функций.**

**Задача**[**коммивояжёра**](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BC%D0%B8%D0%B2%D0%BE%D1%8F%D0%B6%D1%91%D1%80) (или **TSP** от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Travellingsalesmanproblem*) — одна из самых известных задач [комбинаторной оптимизации](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F), заключающаяся в поиске самого выгодного [маршрута](https://ru.wikipedia.org/w/index.php?title=%D0%9C%D0%B0%D1%80%D1%88%D1%80%D1%83%D1%82_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)&action=edit&redlink=1), проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешёвый, совокупный критерий и тому подобное) и соответствующие матрицы расстояний, стоимости и тому подобного. Как правило, указывается, что маршрут должен проходить через каждый город только один раз — в таком случае выбор осуществляется среди [гамильтоновых циклов](https://ru.wikipedia.org/wiki/%D0%93%D0%B0%D0%BC%D0%B8%D0%BB%D1%8C%D1%82%D0%BE%D0%BD%D0%BE%D0%B2_%D1%86%D0%B8%D0%BA%D0%BB).

В основном задачи решаются полным перебором, но в некоторых случаях можно использовать не совсем правильное решение, а решение, которое близко к правильному. Такой метод используется в том случае, если решать задачу полным перебором очень долго, а примерно близкое решение позволяет выйти к верному ответу в разы быстрее.

**45. Логика и исчисление высказываний. Формальное определение.**

**Разница между логикой высказываний и исчислением высказываний:** исчисление относится к абстракции, логика привязано к конкретной ситуации.

Чтобы работать с логикой, необходимо ввести понятие исчисления.

Условия, определяющие исчисление (они же формально определяют исчисление):

* 1. Имеется алфавит исчисления, элементами которого называются символами. Конечная последовательность символов - слово. W - множество слов
  2. Задано подмножество - множество выражений исчисления. Это формулы.
  3. Выделено множество Ax входящее в W. Это аксиомы.
  4. Множество |R={R1,R2,…} - правила вывода исчисления

Исчисление высказываний статично. Позволяет описать статичные ситуации. Высказывания - исчисление состояний.

Исчислений высказываний бывает множество.

Базис позволяет определиться с алфавитом. Чтобы знать с какими знаками работаем.

Переменные пропозициональные, потому что они являются некоторыми утверждениями.

Пример.

Исчисление Гильбертовского типа.

1)Алфавит.

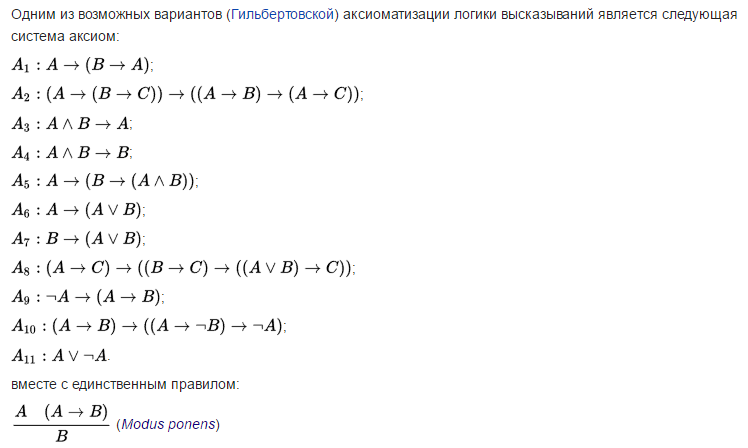
Пропозициональные переменные, логические знаки:И, ИЛИ, НЕ импликация, строгая дизъюнкция, эквивалентность.

2)Формулы.

* + 1. Отдельно стоящая пропозициональная переменная – формула.
    2. Если А - формула, то НЕ(А) - тоже формула.
    3. Если А и В произвольные формулы, то (A -> B),(A V B),(A\*B) тоже формулы.
    4. Что-либо ешё, определяемое формулами 1-3.

3)Аксиомы.

4)Формула вывода - Modusponens(«правило вывода»): если {\displaystyle A}А и {\displaystyle A\to B}А→В — выводимые формулы, то {\displaystyle B}В также выводима.



**46. Исчисление высказываний. Проблемы общезначимости и непротиворечивости.**

Формула, истинная во всякой интерпретации при любой оценке, называется **общезначимой формулой** (тождественно истинной формулой, тавтологией или логическим законом). То, что формула A общезначима, обозначается через A╞.

Очевидно, что общезначимость формулы равносильна общезначимости её замыкания. Поэтому при исследовании вопроса об общезначимости формулы, можно предполагать, что эта формула замкнута.

Суть задачи **общезначимости** сводится к необходимости доказательства истинности некоторого заданного утверждения при любом наборе значений переменных.

**Задача непротиворечивости** - необходимость установления наличия противоречия в наборе некоторых заданных логических утверждений.

**47.** .**Исчисление высказываний Проверка выводимости правильных умозаключений. Алгоритм Квайна. Правило резолюций. Алгоритм Вонга.**

**I Формальные системы** (ФС)



A – [алфавит](https://pandia.ru/text/category/alfavit/) имён

*З* – алфавит специальных разделительных знаков

*ППФ* – правильно построенные формулы из имён и знаков

*А*Î*ППФ* – формулы, которые называются [аксиомами](https://pandia.ru/text/category/aksioma/)

*ПВ*– правила вывода на множестве ППФ вида «если , то »



Исчисление , такое что *I* конструктивно порождается из множества аксиом применением ПВ.



**Пример 1**. Формальные грамматики. , где A – терминальный алфавит; *V* – вспомогательный алфавит; *Р* – правила вывода, вида , где  (все слова из объединённого алфавита); *S* – начальный вспомогательный символ (начало порождения) суть аксиома.



Грамматика *G* порождает (исчисляет) язык , который является множеством слов, выделенных для целей [синтаксического](https://pandia.ru/text/category/sintaksis/) описания моделей некоторых объектов (например, языка программирования ALGOL).



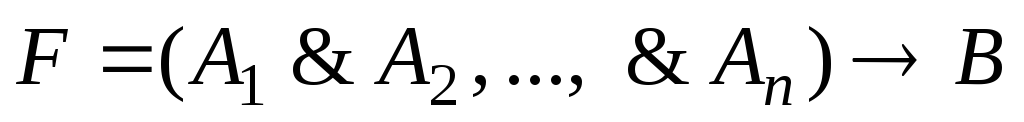
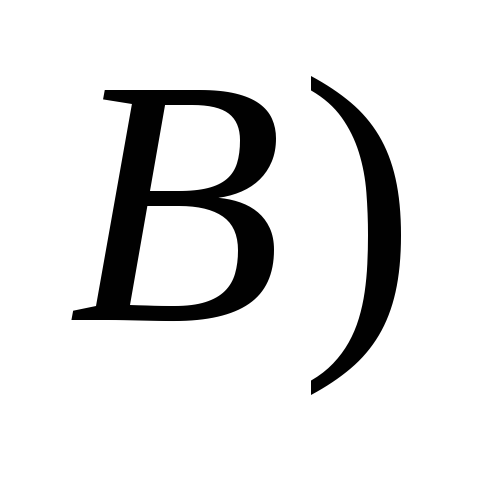
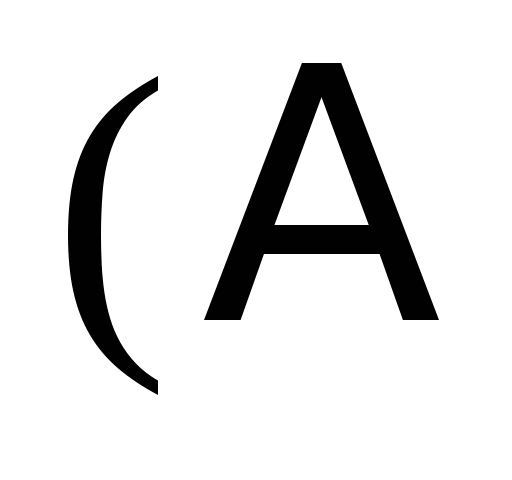
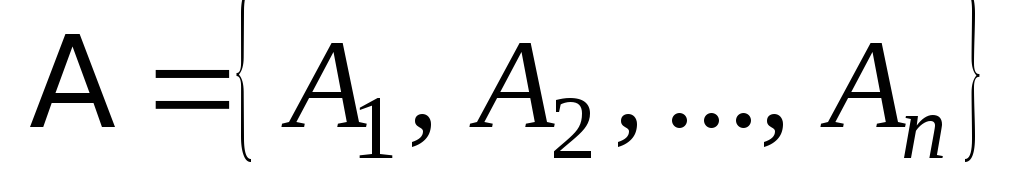
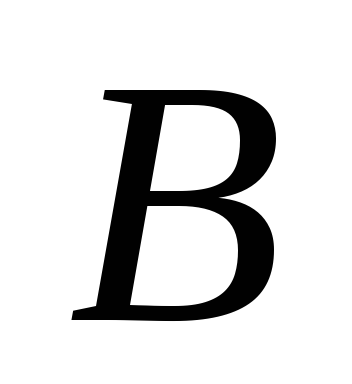
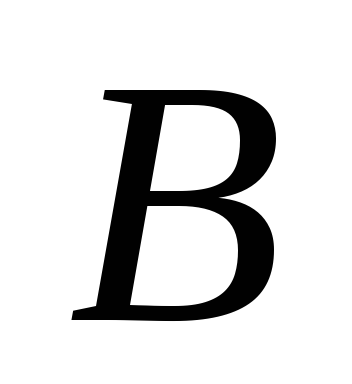
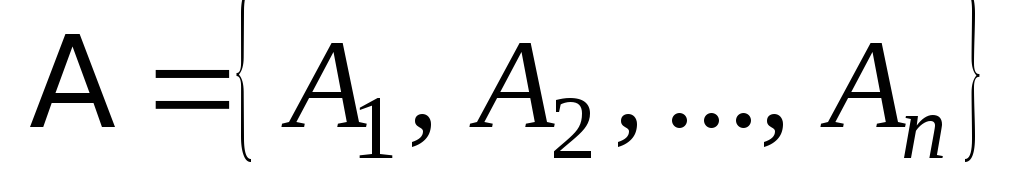
II. Формальная модель высказываний (язык высказываний)

A – алфавит имён =. *Z*– алфавит знаков ={*®*,*&,* ù, *V*, Å, º*,* другие знаки логических операций,  – скобки }. *ППФ* суть скобочные формулы из A и *Z*, построенные по определённым правилам, например  – есть ППФ, а формула  не является ППФ. Буквами *A, B,*C и т. д. обозначаются сложные ППФ.

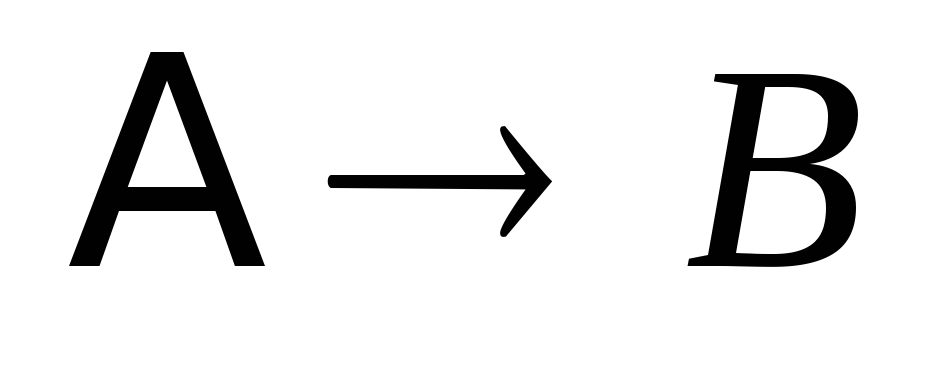


# VI. Методы и алгоритмы проверки выводимости.

Пусть задана совокупность ППФ , которая называется посылками (иногда гипотезами), и ППФ –. «» называется логическим следствием. или выводимой изА (записывается как ⊦), еслиявляется тавтологией (тождественно истинным высказыванием).

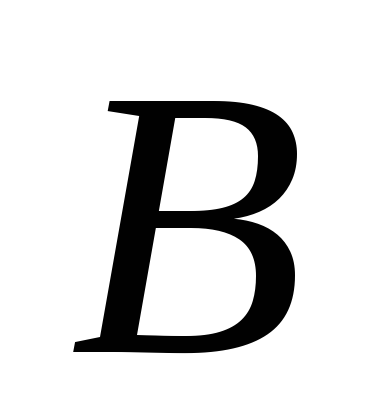
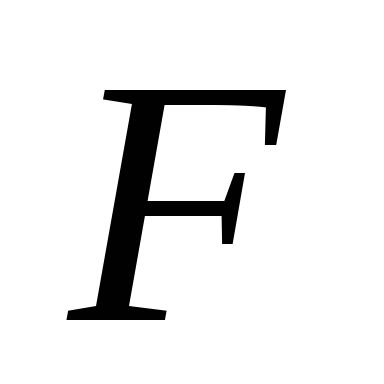
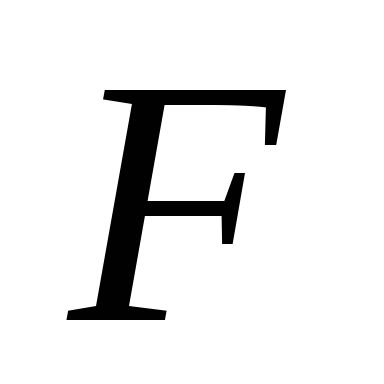
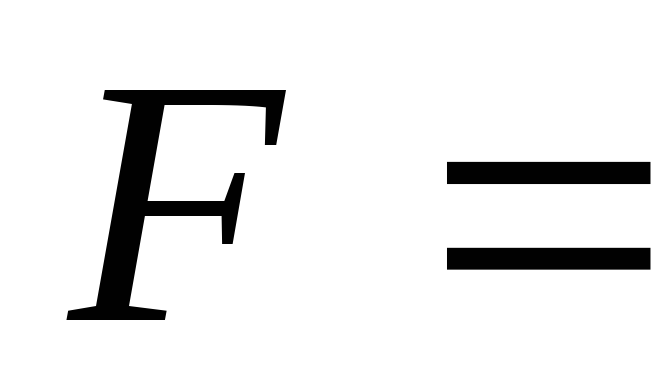
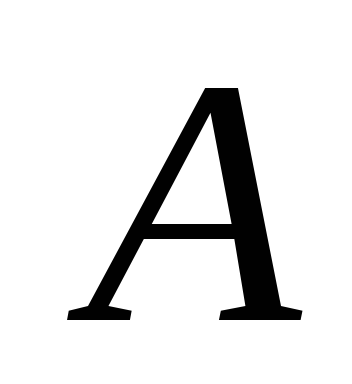
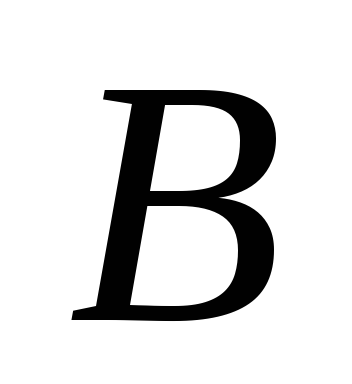
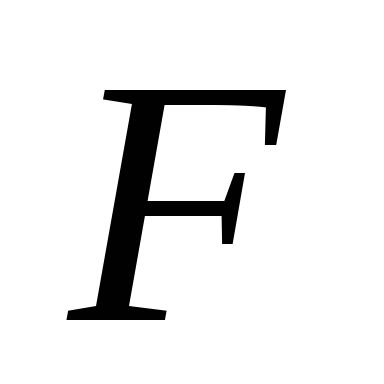
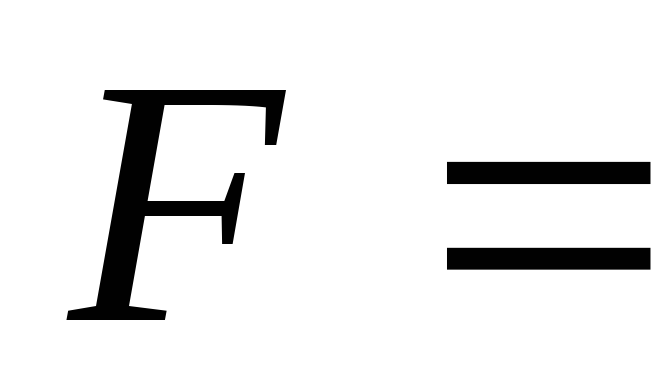
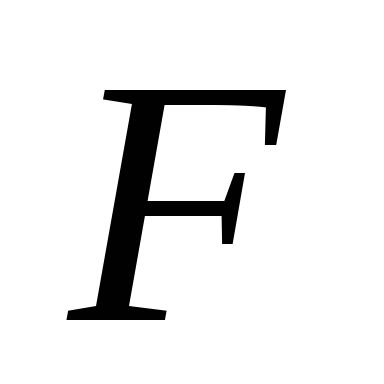


Таким образом, задача проверки выводимости сводится к проверке на тождественную истинность. Существует несколько десятков методов и алгоритмов установления тождественной истинности логической формулы.



1. **Алгоритм истинностных таблиц** (АИТ) **для *F=A*→*B.***

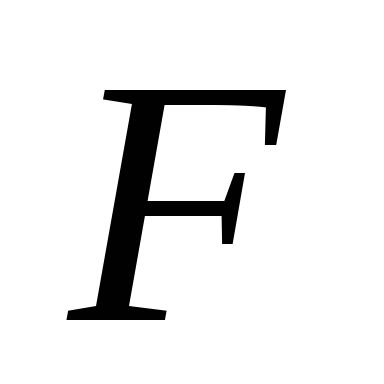
АИТ сводится к последовательной подстановке всех возможных интерпретаций (наборов «истина» и «ложь») переменных, входящих в . Алгоритм останавливается, если значениеложь (не выполнима, а значитне выводима изна всех интерпретациях);истина (выполнима на всех интерпретациях, значитсуть тавтология иА⊦. Такой алгоритм требует в наихудшем случае 2*n* подстановок (2*n* возможных интерпретаций), где «*n*» число переменных, входящих в формулу *F*.



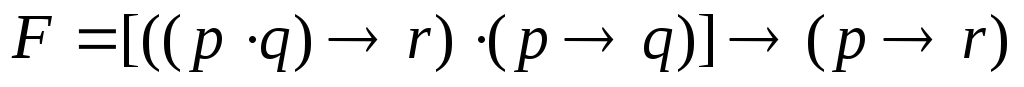
1. **Алгоритм Квайна**(Квайн. 1960 г. США)

Идея: при последовательных подстановках значений переменных можно уменьшить длину формулы, исходя из совокупности проведённых проверок истинности *F*, и тем самым сокращать число переменных для проверки.

Вводится понятие дерева испытаний, которое по сути дела представляет собой граф всех интерпретаций проверяемой формулы . Квайн назвал его «семантическим деревом».



Пример: Пусть необходимо проверить общезначимость формулы



*Алгоритм Квайна*. Напомним, что формула *F* от пропозициональных переменных *А1*, *А2*, …, *Аn* является общезначимой (тождественно истинной) или (что то же самой) доказуемой, если булева функция  , соответствующая формуле *F*, тождественно равна 1. Для проверки значений функции *f* используется так называемое *семантическое дерево*, т.е. бинарное дерево, которое удовлетворяется следующим условиям:



1) каждое ребро соответствует переменной (с отрицанием или без);

2) ребра, выходящие из одной вершины, соответствуют контрарным переменным:  ,  ;



3) ребра соответствуют одной и той же переменной  тогда и только тогда, когда они находятся на одинаковом расстоянии от корня (рис. 1.1).



|  |
| --- |
|  |
|  |  |

Семантическое дерево имеет  висячих вершин и для проверки общезначимости необходимо пройти  маршрутов от корня до этих вершин.



*Алгоритм Квайна* позволяет проходить не все семантическое дерево, а только его часть. Он состоит в том, что пропозициональным переменным  , упорядоченным в набор (*А1*,*А2*, …, *Аn*), последовательно придают значения 0 и 1 и анализируют таблицы истинности формул, содержащих меньшее число переменных.



Пример. Проверить общезначимость формулы

 .



Упорядочим пропозициональные переменные (*А*, *В*, *С*). Придадим первой переменной *А* значение 1, т.е. *А*=1. Тогда формула *F* преобразуется следующим образом:

  .



В полученной формуле переменной *В* придадим значение 1, т.е. *В*=1. Тогда преобразованная формула имеет вид  , т.е. будет общезначимой. В случае *В*=0 имеем  , что также общезначимо. Рассмотрим теперь случай *А*=0. Имеем



 .



Таким образом, все возможные случаи приводят к тождественно истинным (общезначимым) формулам. Следовательно, формула *F* также истинна. На рис. 1.2а изображено семантическое дерево, соответствующее формуле *F*, а на рис. 1.2б показана часть семантического дерева, которая фактически использовалась для проверки общезначимости.

|  |
| --- |
|  |
|  |  |

**48. Проверка общезначимости в исчислении высказываний. Алгоритм редукции. Пример.**

Проверка общезначимости – установление факта истинности значения на любом наборе пропозиционных переменных.

Алгоритм редукции: при большом количестве часть семпликаций упрощает проверку (подбор аргумента, доведение высказывания до абсурда)

**49. Задача логического вывода. Подходы к решению.**

**Задача логического вывода** состоит в получении новых знаний из уже имеющихся знаний. Получение новых формул из аксиом и формул, полученных ранее.

В основу современных методов решения логических задач положены формальные конструкции дедуктивной логики с использованием силлогизмов Аристотеля, основанных на логике утверждений (суждений). Например, из двух утверждений: "Все люди смертны" и "Аристотель – человек" следует вывод: "Аристотель смертен".

Используя дедуктивную логику, из двух или нескольких исходных аксиом, имеющихся в логической базе знаний, можно вывести очередное утверждение-следствие или доказать истинность (ложность) целевого утверждения (теоремы) путем использования определенных правил вывода. Этот процесс получения новых знаний из имеющихся аксиом называют логическим выводом на знаниях.  
  
Источник: <http://5fan.ru/wievjob.php?id=21195>

Подходы:

1. Метод резолюций. Базовая операция - резолюция. Известно, что оба дизъюнкта истинны. Можно построить рехольвенту дизъюнктов. При этом резольвента так же будет истинна.
2. Метод резолюций для хорновских дизъюнктов
3. Метод деления дизъюнктов

**50.** **Исчисление высказываний. Метод резолюций.**

Базовая операция - резолюция. Известно, что оба дизъюнкта истинны. Можно построить резольвенту(**логическое** следствие дизъюнктов и , т. е. ╞ .) дизъюнктов. При этом резольвента так же будет истинна. Потому что при любом значении контрарной литеры одно из двух выражений будет истинно. Следовательно резольвента истинна.

Резольвентой дизъюнктов C1 и C2 называется любая из следующих бинарных резольвент: бинарная резольвента C1 и C2, бинарная резольвента C1 и склейки C2, бинарная резольвента C2 и склейки C1, бинарная резольвента склейки C1 и склейки C2.

Доказуемо Что R1R2=>res(r1,r2)

Если резольвента равна нулю, то доказуемо, что из R1R2 следует ложь.

**Теорема.** Множество дизъюнктов противоречиво только тогда, когда существует резолютивный вывод заканчивающийся нулем.

Метод резолюций состоит в доказательстве целевого утверждения через его отрицание. Если доказали противоречивость от отрицания целевого, значит целевое истинно.

При выводе используются различные стратегии. Стратегия вывода - последовательность построения резольвент.

**Теорема** (полнота метода резолюций). Если множество дизъюнктов S невыполнимо, то существует резолютивный вывод пустого дизъюнкта из S.

Пусть дано множество S, состоящее из следующих дизъюнктов:

1) P(x),

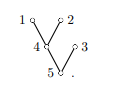
2) ¬P(f(y)) ∨ Q(a),

3) ¬Q(x).

Попробуем вывести пустой дизъюнкт из S:

4) Q(a) — резольвента дизъюнктов 1 и 2 (отрицание литеры P(x) дизъюнкта 1 и литера ¬P(f(y)) дизъюнкта 2 имеют НОУ {f(y)/x}),

5) — резольвента дизъюнктов 3 и 4 (литера ¬Q(x) дизъюнкта 3 и отрицание литеры Q(a) дизъюнкта 4 имеют НОУ {a/x}).

Таким о бразом, S невыполнимо. Дерево вывода дизъюнкта из S имеет следующий вид:

# 51. Метод резолюций для хорновских дизъюнктов

В общем случае метод резолюций требует больших вычислений. Если дизъюнкты имеют специальный вид, являются, так называемыми хорновскими дизъюнктами, то вычисления упрощаются.

Хорновский дизъюнкт - дизъюнкт, содержащий не более одного положительного дизъюнкта.

Литера называется позитивной, если она не содержит отрицания и негативной, если содержит отрицание.

Примеры хорновских дизъюнктов: А, В, А, В, А С В, А В, А С С D. Если все дизъюнкты отрицательны - негативный

Если один положительный - унитарный позитивный

Рассмотрим множество S хорновских дизъюнктов без тавтологий. Невыполнимость можно проверить с помощью следующего алгоритма.

1.Полагаем, что S0= S.

2.Пусть S n-1, n≥1, построено. Для построения S n выбираем из S n-1

дизъюнкты D1 и D2 такие, что:

D1 - унитарный позитивный дизъюнкт, пусть, например, D1=Р; D2 - дизъюнкт, содержащий литеру Р. Вычисляем резольвенту R для дизъюнктов D1 и D2 и полагаем, что S n = (S n-1\{ D2}) {R}. Эту

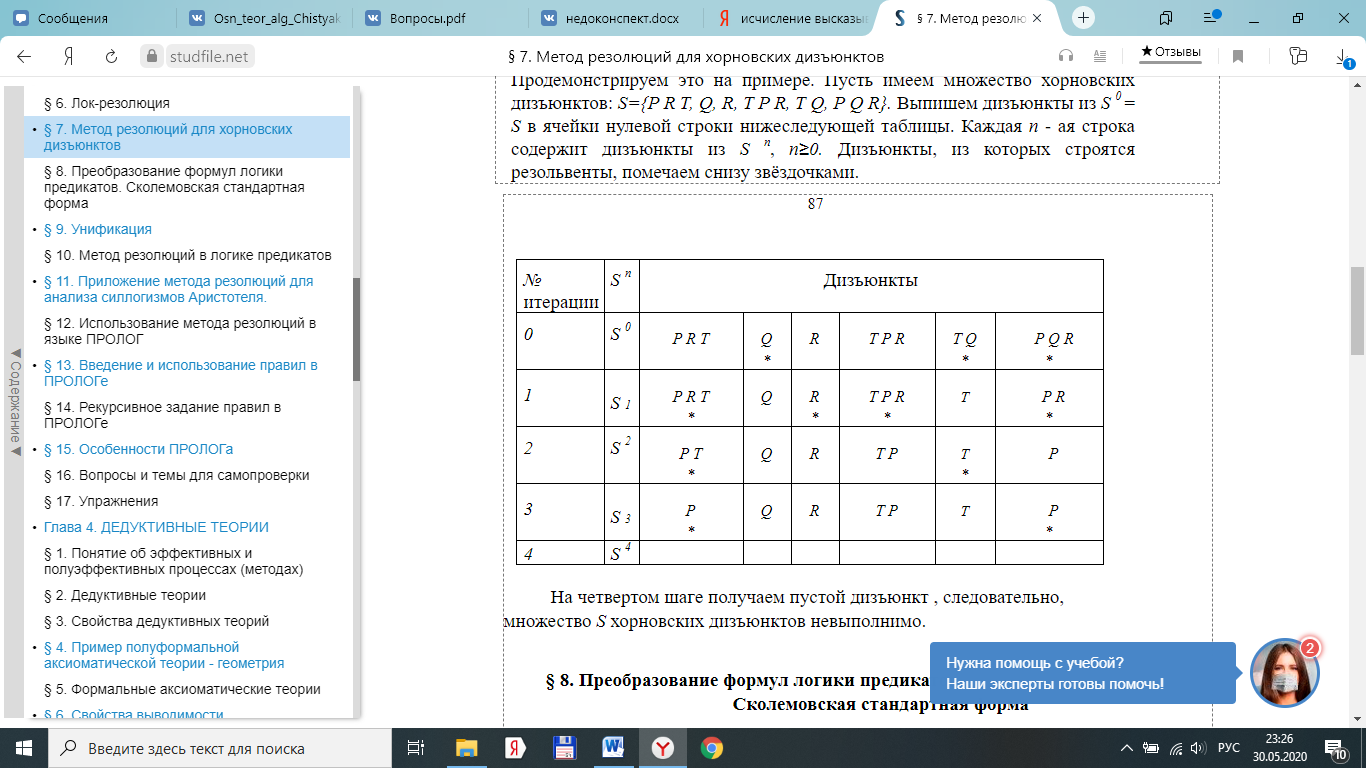
процедуру повторяем до тех пор пока не получим пустой дизъюнкт либо пока не окажется, что в S n-1 не существует дизъюнктов D1 и D2

указанных видов.

Можно доказать, что для приведенного алгоритма появление пустого дизъюнкта означает, что множество S хорновских дизъюнктов невыполнимо. Если же окажется, что S n-1 не содержит дизъюнктов D1 и D2 указанных видов, то исходное множество S хорновских дизъюнктов выполнимо.

Реализацию этого алгоритма проще проводить с помощью таблицы. Продемонстрируем это на примере. Пусть имеем множество хорновских дизъюнктов: S={P R T, Q, R, T P R, T Q, P Q R}. Выпишем дизъюнкты из S 0 = S в ячейки нулевой строки нижеследующей таблицы. Каждая n - ая строка содержит дизъюнкты из S n, n≥0. Дизъюнкты, из которых строятся резольвенты, помечаем снизу звёздочками.

Берем унитарный позитивный дизъюнкт. Подставляем его в выражение, содержащее резольвентную пару. Продолжаем так делать пока не получим ноль. Если получили ноль - противоречие. Если ничего не получили, значит исходное множество непротиворечиво.





**52. Исчисление высказываний. Метод деления дизъюнктов.**

//Из всей инфы что я нашёл в интернете, что-то хоть как то более-менее разборчивое для ума первокурсника нашёл только в виде презентации от ВятГу

Анализ параллелизма в | Полное деление дизъюнктов

hpcc.unn.ru›file.php?id=185

**53. Формальное определение исчисления предикатов первого порядка.**

Формальная система (формальная теория, исчисление) - совокупность чисто абстрактных объектов (объектов, никак не связанных с реальным миром), в которой представлены правила оперирования цепочками символов в исключительно синтаксической трактовке без учета какого-либо смыслового содержания.

Формальная система (теория) **S** определяется как четверка

**S**=<**T**, **F**, **A**, **R**> ,

где  
**T** - *алфавит* теории (конечное множество базовых символов);  
**F**- множество (перечислимое) *формул* (называемых также **п**равильно **п**остроенными **ф**ормулами - сокращенно, **ппф**), построенных из элементов **T** с использованием некоторого набора синтаксических правил;  
**A** - множество формул, называемых *аксиомами*;  
**R** - конечное множество *правил* вывода.

Правило вывода - отношение вида ri(f1, f2, ... fn , g) на множестве **F**. При этом формула g называется *непосредственно выводимой* из множества формул f1, f2, ..., fn по правилу ri; формулы f1, f2, ..., fn называются *посылками*, а g - *следствием* или *заключением*.  
Для записи правил вывода часто используют представленную ниже нотацию:

ri: f1, f2, ... fn |- g .

Различают правила вывода двух типов:

* правила , применяемые к формулам, рассматриваемым как единое целое - *правила продукций*, *продукции*, *продукционные правила*;
* правила, которые могут применяться к любой отдельной части формулы (если эта часть сама является формулой) - *правила переписывания*.

*Пример*. Так, в математике правило вывода

x>y, y>z |- x>z

является правилом продукции.

*Выводом* формулы fm в теории **S** из множества формул {f1, f2, ..., fn} называется последовательность формул fn+1, fn+2, ..., fm такая, что для любого i = n+1...m формула fi либо аксиома, либо непосредственно выводима из множества {f1, f2, ..., fi-1} (или его подмножества) по одному из правил множества **R**.

*Примечание*. Напоминаем, что запись вида {...} используется для перечисления элементов некоторого множества.

*Доказательством* формулы g в **S** называется вывод g, в котором в качестве исходного множества формул используются только аксиомы; g называется *теоремой* **S**, если для нее существует доказательство.

Формальная теория **S** называется *разрешимой*, если существует эффективная процедура, которая для любой формулы определяет, является ли данная формула теоремой или нет.

*Пример*. Простая формальная система:  
Алфавит **T** = {a, b, #};  
Формулы **F** = {символ или любая последовательность символов алфавита **T**};  
Аксиомы **А** = {a#a} (одна единственная аксиома);  
Правила вывода **R** = {*C1*#*C*2 |- b*C*1#*C*2b} (одно единственное правило вывода), *C*1 и *C*2 обозначают любые последовательности символов a и b. Представленное правило вывода является правилом продукции.  
Некоторые формулы теории (ппф): ###, aaa, bababbb, b#a#ab, a.  
Формулы, не являющиеся ппф: abby, bob, baobab.  
В данной теории формула babba#ab непосредственно выводима из формулы abba#a по единственному имеющемуся правилу вывода (но не является теоремой).  
Теоремами здесь будут только следующие формулы:

ba#ab,  
bba#abb,  
bbba#abbb,  
bbbba#abbbb,  
и так далее.

*Пример*. Еще одна формальная система:  
Алфавит **T** = {m, i, u};  
Формулы **F** = {символ или любая последовательность символов алфавита **T**};  
Аксиомы **А** = {mi} (одна единственная аксиома);  
Правила вывода **R** = {

r1: *C*i |- *C*iu (продукция),  
r2: m*C* |- m*CC* (продукция),  
r3: iii |- u (правило переписывания),  
r4: uu |- (пусто) (правило переписывания)  
},

где *C* обозначает любую последовательность символов алфавита **T**.

Наибольшее распространение в системах искусственного интеллекта получила формальная система, носящая название *исчисления предикатов первого порядка* (ИППП).

Алфавит ИППП состоит из:

* *предметных переменных* z, y, x, ... (для их обозначения используются строчные буквы из конца латинского алфавита);
* *предметных констант* a, b, c, ... (для их обозначения используются строчные буквы из начала латинского алфавита);
* n-местных *предикатных букв* Pin, n>0, i=1,2,... (для их обозначения используются прописные буквы из середины латинского алфавита);
* m-местных *функциональных букв* fim, m>0, i=1,2,... (для их обозначения используются строчные буквы из середины латинского алфавита);
* знаков связок: & (коньюнкция), | (дизъюнкция), ~ (отрицание), -> (импликация);
* *кванторов* общности A-1 и существования E-1;
* круглых скобок.

Понятие формулы в ИППП определяется в два этапа, с использованием представленных ниже синтаксических правил.

1. Терм:
   * всякая предметная переменная и константа является термом;
   * если fim - функциональная буква и t1, t2, ..., tm - термы, то fim(t1, t2, ..., tm) - также терм.
2. Формула:
   * если Pin - предикатная буква и t1, t2, ..., tn - термы, то Pin(t1, t2, ..., tn) - *элементарная формула (атом)*, являющаяся частным случаем правильно построенной формулы (ппф);
   * если F и G - ппф, а x - предметная переменная, то ~F, (F&G), (F|G), (F->G), A-1x(F) и E-1x(F) - также ппф.

Элементарная формула (атом) F и ее отрицание ~F называются *литералами* (соответственно, положительным и отрицательным).

В формулах A-1x(F) и E-1x(F) принято F называть *областью действия* квантора, при этом переменная x в области действия квантора характеризуется как *связанная*, вне области действия она *свободна*.  
Формула называется *замкнутой*, если она не содержит свободных переменных.

Рассматриваемое исчисление предикатов является исчислением первого порядка, поскольку в нем допустимы кванторы лишь по переменным (в исчислении второго порядка - дополнительно по предикатным и функциональным буквам).

*Пример*. Несколько правильно построенных формул ИППП:

* P(x) ,
* P(x)&~Q(a,y) ,
* A-1x(E-1y(Q(x,y))) ,
* E-1x(P(x)|Q(x,b))->A-1y(Q(y,b)&R(y,f(x,b),a)) .

Из всех представленных формул только третья является замкнутой.

Если областью действия квантора является вся формула, то охватывающие ее круглые скобки можно опускать, поэтому третья формула может быть записана более кратко в следующем виде:

A-1xE-1yQ(x,y) .

Множество (бесконечное) аксиом *чистого* исчисления предикатов (исчисления, не содержащего фукциональных букв и предметных констант) задается следующим множеством их схем:

A1: F->(G->F)  
A2: (F->G)->((F->(G->H))->(F->H))  
A3: (F&G)->F  
A4: (F&G)->G  
A5: F->(G->(F&G))  
A6: F->(F|G)  
A7: G->(F|G)  
A8: ~~F->F  
A9: (F->G)->((F->~G)->~F)  
A10: (F->H)->((G->H)->((F|G)->H))  
A11: A-1x(F(x))->F(y)  
A12: F(y)->E-1x(F(x))

,где F, G и H - *метапеременные*, которые могут быть заменены произвольными ппф для получения конкретных аксиом, а F(x) в двух последних схемах обозначает любую ппф, содержащую свободные вхождения переменной x, при этом ни одно из них не находится в области действия квантора по y.

*Примечание*. Фомулы F->G и ~F|G, а также F&G и ~(F->~G) эквивалентны, т.е. являются синонимами (подробнее об эквивалентности см. ниже).  
*Примечание*. Для ИППП возможны и другие системы аксиом, например, существует система трех схем аксиом, содержащих только знаки отрицания и импликации.  
*Примечание*. Аксиомы ИППП A1 ... A10 совпадают с аксиомами исчисления высказываний.

В ИППП используются следующие правила вывода.

1. Правило заключения (modusponens)

R1: F, F->G |- G  
(если F и F->G являются выводимыми формулами, то G - выводимая формула).

1. Правило обобщения (правило A-1-введения)

R2: F->G(x) |- F->A-1x(G(x))  
, где G(x) содержит свободное вхождение переменной x, а F их не содержит.

1. Правило E-1-введения

R3: G(x)->F |- E-1x(G(x))->F  
при тех же требованиях к G(x) и F, что и в предыдущем правиле.

*Примечание*. Представленные выше аксиомы даны в форме с метапеременными, это возможно, поскольку существует правило подстановки

R0: U(A) |- U(G)

, где U(A) - выводимая формула, содержащая атомарную формулу A, а U(G) - формула, получающаяся из U(A) заменой всех вхождений A на произвольную ппф G.

На основе схем аксиом и указанных правил вывода могут быть доказаны "*метатеоремы*", пригодные для использования в качестве дополнительных правил вывода.

Двумя полезными правилами-"метатеоремами" являются представленные ниже.

1. Правило переименования свободных переменных

R4: F(x) |- F(y)  
, где F(x) содержит свободные вхождения x, ни одно из которых не находится в области действия квантора по y.  
Доказательство:  
а) |- F(x) (по условию)  
б) F(x)->(G->F(x)) (аксиома A1, где в качестве G выбирается любая доказуемая формула, не содержащая свободных вхождений x)  
в) G->F(x) (результат применения правила заключения R1 к формулам а и б)  
г) G->A-1xF(x) (правило обобщения R2 к формуле в)  
д) A-1xF(x) (правило заключения к формуле г при учете доказуемости G)  
е) F(y) (правило заключения к д и аксиоме A11)

1. Правило переименования связанных переменных

R5': A-1xF(x) |- A-1yF(y)  
R5": E-1xF(x) |- E-1yF(y)  
, где F(x) содержит свободные вхождения x, ни одно из которых не находится в области действия квантора по y, и не содержит свободных вхождений y.  
Доказательство для правила-"метатеоремы" R5':  
а) |- A-1xF(x) (по условию)  
б) A-1x(F(x))->F(y) (аксиома A11)  
в) A-1x(F(x))->A-1y(F(y)) (правило обобщения к б)  
г) A-1yF(y) (правило заключения к а и в)

**54. Преобразование выражения исчисления предикатов первого порядка в конъюнктивную нормальную форму. Пример.**

Для перевода в КНФ выносим кванторы в левую часть, после применяем правила булевой алгебры для перевода в КНФ.

https://studopedia.ru/3\_11520\_preobrazovanie-proizvolnoy-formuli-logiki-predikatov-pervogo-poryadka-v-klauzalnuyu-formu.html

**55. Сколемовское преобразование. Операции унификации.**

Выносим все кванторы. Убираем кванторы существования, заменяем переменные на термы от предшествующих переменных.

Операция унификации. Допустим у нас есть формулы. Надо проверить их на равенство. Для этого берем переменные и пытаемся изменить их значение, чтобы добиться равенства. Процедура обнаружения одинаковых литералов.

Сколоменская стандартная форма

Замкнутая формула находится в сколемовской стандартной форме (ССФ), если I она находится в предварённой нормальной форме и I её кванторная приставка не содержит кванторов ∃:

∀exn(D1 &*...*&D*k*)

### Например, формула

∀x ∀u (P(x)&(¬P(f(x)) ∨ R(x*,*g(x)))&¬R(x*,*u))

находится в сколемовской стандартной форме

Наряду с “находится в ССФ” будем говорить “является ССФ”

Один из этапов метода резолюций — сведение ПНФ к настолько же (не)выполнимой ССФ

Лемма(об удалении квантора существования)

Пусть — замкнутая формула (*n* ≥ 0) и

функциональный символ f не содержится в *χ*. Тогда ||= *ϕ* ⇔ ||= ∀exn(*χ*{xn+1*/*f(exn)}) Доказательство (леммы).

(⇐): ПустьI — модель для формулы *ψ* = ∀exn(*χ*{xn+1*/*f(exn)})

Тогда для любого набора предметов *d*e*n*верно: I |= *χ*{xn+1*/*f(exn)}[*d*e*n*]

Рассмотрим предмет *dn*+1 = f(*d*e*n*)

Пусть будет верно следующее: I |= *χ*[*d*e*n, dn*+1]

Значит, I |= (∃xn+1 *χ*)[*d*e*n*], и I |= ∀exn∃xn+1 *χ*

 — замкнутая формула (*n* ≥ 0) и

функциональный символ f не содержится в *χ*. Тогда ||= *ϕ* ⇔ ||= ∀exn(*χ*{xn+1*/*f(exn)}) Доказательство (леммы).

(⇒): ПустьI — модель для *ϕ*

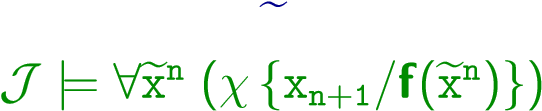
Тогда для любого набора предметов *d*e*n*существует предмет *dn*+1, такой что I |= *χ*[*d*e*n, dn*+1]

По I построим новую интерпретацию J:

если в сигнатуре был функциональный символ f, удалим его добавим в сигнатуру функциональный символ f(*n*)

оценим f так, чтобы выполнялось: f(*d*e*n*) = *dn*+1

Тогда J |= *χ*{xn+1*/*f(xn)}[*d*e*n*],

а значит,H

 — замкнутая формула (*n* ≥ 0) и

функциональный символ f не содержится в *χ*. Тогда ||= *ϕ* ⇔ ||= ∀exn(*χ*{xn+1*/*f(exn)})

Небольшая вольность: если слева от ∃не стоит ни одного ∀, то, согласно лемме, f — 0-местный функциональный символ, то есть константа, и *ψ* имеет вид *χ*{xn+1*/*f}

Сколемизация — это устранение кванторов ∃с введением новых символов с целью получить более простую “хорошую” формулу (*здесь — сохраняющую выполнимость и невыполнимость*)

При устранении ∃на место удаляемой переменной подставляются сколемовские термы (*здесь —* f(xn))

e

### Алгоритм сколемизации ПНФ

*Дано:* ПНФ *ϕpnf*

*Требуется* получить ССФ *Sk*(*ϕpnf*), такую что

*ϕpnf*выполнима ⇔*Sk*(*ϕpnf*) выполнима *Как работает алгоритм: ϕpnf*: ∀x1 *...*∀xk−1 ∃xk∀xk+1 *...*∀xm−1 ∃xm*... χ*

∀x1 *...*∀xk−1 ∀xk+1 *...*∀xm−1 ∃xm*...* (*χ*{xk*/*f(x1*,...,*xk−1)})

∀x1 *...*∀xk−1 ∀xk+1 *...*∀xm−1 *...*

(*χ*{xk*/*f(x1*,...,*xk−1)*,*xm*/*g(x1*,...,*xk−1*,*xk+1*,...,*xm−1)})

*Sk*

*Требования к выбору функциональных символов:* формула *χ* не содержит вхождений символа f символы f и g различны формула *χ* не содержит вхождений символа g, *...*

### Пример

|  |  |
| --- | --- |
| *ϕpnf*: | ∀x ∃z ∃y ∀u (P(x)&(¬P(z) ∨ R(x*,*y))&¬R(x*,*u))  ∀x ∃y ∀u (P(x)&(¬P(f(x)) ∨ R(x*,*y))&¬R(x*,*u)) |
| *Sk*(*ϕpnf*) : | ∀x ∀u (P(x)&(¬P(f(x)) ∨ R(x*,*g(x)))&¬R(x*,*u)) |

Теорема(*о сколемизации*). Если *ϕpnf*— ПНФ, то *Sk*(*ϕpnf*) — ССФ, для которой верно следующее:

||= *ϕpnf* ⇔ ||= *Sk*(*ϕpnf*)

Доказательство. Достаточно конечное число раз применить *лемму об удалении квантора существования* H

Унификация

Унификация атомов *A*, *B* достигается применением к ним *подстановки θ*, такой что *Aθ*= *Bθ*

### Напоминание

*Подстановка* — это отображение *θ* :Var→ Term

Конечная подстановка задаётся множеством *связок*:

{x1*/t*1*, ...,* xn*/tn*}

*Eθ*— это результат применения подстановки *θ* к выражению *E*

Чтобы поставить и решить задачу унификации, исследуем алгебраические свойства подстановок

Композиция подстановок *θ*, *η* — это подстановка *θη*, такая что для любой переменной x верно: x(*θη*) = (x*θ*)*η*

Утверждение. Пусть *θ* = {x1*/t*1*,...,*xn*/tn*} и *η* = {y1*/s*1*,...,*yk*/sk*}. Тогда *θη*= {xi*/tiη*| 1 ≤ *i* ≤ *n, xi*6= *tiη*}

∪{yj*/sj*| 1 ≤ *j* ≤ *k,* yj∈ {*/* x1*,...,*xn}}

Доказательство. Рассмотрим переменную z ∈Var

Если z ∈*/* Dom*θ*∪Dom*η*, то z(*θη*) = (z*θ*)*η*= z*η*= z

Если z = yj∈Dom*η*\ Dom*θ*, то z(*θη*) = (z*θ*)*η*= z*η*= *sj*

Иначе z = xi∈Dom*θ*, и z(*θη*) = (z*θ*)*η*= *tiη* H Пример *θ* = {x*/*f(x*,*c)*,* y*/*g(u)*,* z*/*y} *η* = {x*/*g(y)*,*y*/*z*,*u*/*c}

*θη*= ?

{x*/*f(x*,*c)*η,* y*/*g(u)*η,* z*/*y*η*} ∪ {u*/*c}

{x*/*f(g(y)*,*c)*,* y*/*g(c)*,* z*/*z} ∪ {u*/*c}

{x*/*f(g(y)*,*c)*,* y*/*g(c)} ∪ {u*/*c} *θη*= {x*/*f(g(y)*,*c)*,* y*/*g(c)*,* u*/*c} Подстановка *θ* называется унификатором выражений *E*1, *E*2, если *E*1*θ* = *E*2*θ*

У(*E*1*, E*2) — множество всех унификаторов выражений *E*1, *E*2

Выражения *E*1, *E*2 унифицируемы, если У(*E*1*, E*2) 6= ∅

Подстановка *η* — специализация подстановки *θ*, если существует подстановка *µ*, такая что *η* = *θµ*

Подмножество *S* множества подстановок Θ — полное в Θ, если любая подстановка из Θ является специализацией хотя бы одной подстановки из *S*

Подстановка *θ* — наиболее общая в множестве подстановок Θ, если *θ* ∈Θ и {*θ*} — множество, полное в Θ

НОУ(*E*1*, E*2) — множество всех наиболее общих унификаторов выражений *E*1, *E*2

### Примеры

Подстановка *η* = {y*/*g(g(v))*,*u*/*f(c)*,*v*/*g(v)*,*x*/*c} — унификатор атомов P(f(x)*,*y), P(u*,*g(v)):

P(f(x)*,*y)*η* = P(f(c)*,*g(g(v))) = P(u*,*g(v))*η*

А подстановка *θ* = {y*/*g(v)*,*u*/*f(x)} — *более общий* их унификатор:

P(f(x)*,*y)*θ* = P(f(x)*,*g(v)) = P(u*,*g(v))*θ η* = *θ* {v*/*g(v)*,*x*/*c}

Оказывается, что *θ* — наиболее общий унификатор атомов P(f(x)*,*y), P(u*,*g(v))

(но как это доказать?)

А выражения P(x*,*f(x)), P(g(y)*,*y) неунифицируемы

(а это как доказать?)

формулируется следующим образом:

для заданных выражений *E*1, *E*2

выяснить, унифицируемы ли эти выражения, и если это так, то вычислить

множество унификаторов, полное в У(*E*1*,E*2)

**56. Модальные логики. Области применения. Особенности. Примеры.**

**Модальная логика -** раздел логики, в котором изучаются логические связи модальных высказываний, те, которые включают модальность. Модальность - модальный оператор.

Проще говоря, модальная логика - расширение исчисления высказываний.

Добавляются дополнительные операторы, позволяя расширить исчисление.

**Модальности:**

* 1. Алетические:
     + Необходимо
     + Возможно
     + Случайно
  2. Деонтические:
     + Обязательно
     + Разрешено
     + Запрещено
  3. Аксиологические (этика):
     + Хорошо
     + Плохо
     + Нейтрально
  4. Темпоральные (позволяет определить порядок событий):
     + В будущем
     + В прошлом
     + Сейчас
  5. Пространственные:
     + Там
     + Здесь
     + Нигде

Чаще всего используются временные и пространственные модальности.

Модальные операторы делают описания проще, позволяя описывать некоторые ситуации.

**Сравнение с формальной логикой**

Формальную логику можно упростить до цепочки **истинное знание**→процесс→выводы.

Откуда брать *истинное знание* для формальных логик, если только единичные *истинные знания* универсальны?

Логика должна отвечать на реальные жизненные ситуации, а *универсальных истин* немного.

Модальная логика в широком смысле оперирует:

* знаниями
* предположениями (то, что не знаем)
* вопросами (частично в логике знаний)
* задачами (что сделать, чтобы получить знание)

То есть является более реальным/практичным расширением высказываний и логики первого порядка.

**Примеры:**

Например, модальная логика способна оперировать утверждениями типа «Москва всегда была столицей России» или «Санкт-Петербург, когда-то в прошлом, был столицей России», которые невозможно или крайне сложно выразить в немодальном языке. Кроме временных и пространственных модальностей есть и другие, например, «известно, что» (логика знания) или «можно доказать, что» (логика доказуемости).

Обычно для обозначения модального оператора используется {\displaystyle \Box }и двойственный к нему. {\displaystyle \diamondsui

{\displaystyle \diamondsuit A=\neg \Box \neg A.}Это отражает то, что сказать «Москва когда-то была столицей России» то же самое, что сказать «не верно, что Москва никогда не была столицей России».

**57.Темпоральная логика линейного времени. Формальное определение. Модальные операторы.**

При всей широте использования классической логики в науке, технике и в обычной жизни очевидны ее ограничения. Классическая логика основывается на самой примитивной модели истины, она не позволяет выразить степень уверенности/неуверенности в истинности высказывания. Формулы логики могут принимать значения только "да" и "нет" на подходящей интерпретации, но не могут определить интервал возможных значений в некоторой области. Формулы обычной логики истинны или ложны независимо от времени, в статическом мире. Вследствие этого, аппарат классической логики оказался недостаточно выразительным во многих областях применения. Поэтому неудивительно, что предпринимались многочисленные попытки расширений классической логики в самых различных направлениях, и некоторые из этих попыток оказались весьма успешными.

Если утверждения естественного языка явно или неявно включают зависимость выска¬зываний от времени или от порядка событий во времени, то формализация их в класси¬ческой логике высказываний обычно неадекватна. Например, коммутативность операции конъюнкции (перестановочность ее аргументов А /\ В º В /\ А) не выполняется для следую¬щих предложений:

"Джон умер, и его похоронили"

не эквивалентно предложению

"Джона похоронили, и он умер";

"Джейн вышла замуж и родила ребенка"

не эквивалентно предложению

"Джейн родила ребенка и вышла замуж";

"Сообщение послано в канал, и на него пришло подтверждение"

не эквивалентно предложению

"На сообщение пришло подтверждение, и оно послано в канал".

Анализ этих утверждений в рамках обычной логики высказываний невозможен. Для адекватного формального выражения подобных утверждений нужна логика, позволяющая отразить соотношения моментов времени наступления событий, в естественном языке определяемые такими словами, как "случилось после", "случается иногда", "случается всегда". Это требует формализации высказываний, истинность которых меняется во вре¬мени. Подходящим способом отражения порядка событий во времени является использование расширений классической логики высказываний. К таким расширениям относится обладающая мощными описательными возможностями темпоральная логика линейного времени (LTL).

В дополнение к базовым логическим операторам (∨, ∧, ¬) и константам (true, false) в этой логике введены шесть временных операторов (модальные операторы):

(иными словами, в темпоральных логиках бывает два вида операторов: логические и модальные.)

• next (X) – выражение Xp истинно в текущем состоянии, если в следующий момент времени p истинно;

• future (F) – выражение Fp истинно в текущем состоянии, если в будущем р станет истинно;

• globally (G) – выражение Gp истинно в текущем состоянии, если во всех будущих состояниях р будет истинно;

• until (U) – выражение pUq истинно в текущем состоянии, если в будущем q станет истинно, а до тех пор истинно будет р;

//этот оператор ↓ иногда не указывают т.к. эквивалентно .

weekuntil (W) – выражение pWq истинно в текущем состоянии, если в будущем q станет истинно, а до тех пор истинно будет р, или же p всегда будет истинно;

• release (R) – выражение pRq истинно в текущем состоянии, если в будущем p станет истинно, а до того момента времени включительно истинно будет q, или же q всегда будет истинно.

В темпоральной логике линейного времени будущее рассматривается как последовательность состояний, поэтому формулы LTL – это формулы пути. Специфика LTL такова, что некоторая формула истинна только в том случае, если она истинна на всех возможных путях развития событий.

Темпоральная логика линейного времени нашла широкое применение в области спецификации формальных требований к аппаратным и программным системам. Формулы LTL используются для записи условий, выполнимость которых на модели системы нуждается в проверке.

Темпоральная логика линейного времени — это расширение классической логики высказываний способное отражать порядок событий во времени. Иными словами, средство для учета временных зависимостей.

https://ru.wikipedia.org/wiki/Темпоральная\_логика (тут основы)

https://studopedia.ru/5\_118441\_temporalnaya-logika.html (тут по подробнее)

http://vestnik.tstu.ru/rus/t\_18/pdf/18\_4\_002.pdf (работа Г.А. Чистякова с применением темпоральной логики)

**58.Темпоральная логика ветвящегося времени. Формальное определение. Модальные операторы.**

Формальное определение:

Логика ветвящегося времени — CTL

CTL — ComputationalTreeLogic — это одна из возможных темпоральных логик ветвящегося времени.

Темпоральная логика ветвящегося времени (ветвящаяся темпоральная логика) может быть использована для решения задач обучения, прогнозирования и моделирования в интеллектуальных системах, когда необходимо рассматривать время ветвящимся в будущее.

Используются 4 старых модальности (X, F, G, U) из LTL и 2 новых (A, E).

Aφ - Alongallpart истинно в текущем состоянии если будет на всех путях от текущего состояния истинно будет фи. (для всех путей из данного состояния LTL формула φ истинна)

Eφ - Aongatleastonepart - если фи истинно хотя бы на одном пути. (существует такой путь из данного состояния, на котором LTL формула φ истинна)

Aφ ≡ ¬E ¬φ

Темпоральная логика ветвящегося времени (ветвящаяся темпоральная логика) может быть использована для решения задач обучения, прогнозирования и моделирования в ИС, когда необходимо рассматривать время ветвящимся в будущее. Известно, что временная логика может быть построена на основе модальной логики. Модальный операторы необходимости p может интерпретироваться как «необходимо, что всегда будет p», а модальный оператор возможности p – как «возможно, что будет p», или, другими словами, p истинно в момент времени t, если и только если p истинно при всех t’ на всякой ветви времени, выходящей из t, а p истинно при всех t, если и только если p истинно при некотором t’ хотя бы на одной ветви, выходящей из t.

В работе анализируются различные подходы к темпоральным высказываниям, являющимся утверждениями о будущем. При одном подходе утверждения о будущем рассматриваются аналогично утверждениям о прошлом и настоящем как констатация положения дел (ассерторические утверждения). В этом случае утверждение «когда-нибудь будет p» истинно в момент времени t, если и только если в некоторый момент t’, следующий за t, истинно p. Если будущее состояние однозначно детерминировано настоящим, то утверждения о будущем будут либо истинными, либо ложными. Однако более соответствует реальности ситуация, когда допускается, что будущее не предопределено однозначно настоящим и возможны различные варианты развития событий, т.е. возможно ветвление в будущее. При втором (альтернативном) подходе утверждения о будущих событиях рассматриваются не как ассерторические, а как модальные утверждения. В этом случае выражение типа «всегда будет p», «когда-нибудь будет p», «через n единиц времени будет p», корректные (правильно построенные) в случае первого подхода, не являются корректными.

• Правильно построенными выражения будут:

Gp - «необходимо (при любом развитии событий) всегда будет p»;

G◊p – «возможно (при некотором развитии событий) всегда будет p»;

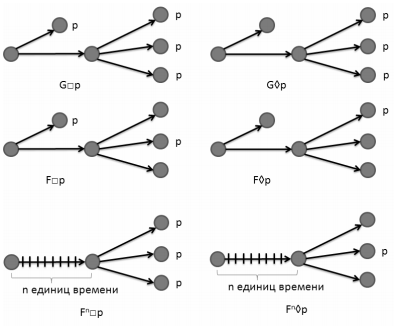
Fp – «необходимо когда-нибудь будет p»;

F◊p – «возможно когда-нибудь будет p»;

Fnp – «через n единиц времени необходимо будет p»;

Fn◊p – «через n единиц времени возможно будет p».

Для каждого приведенного оператора процесс развития событий представляется в виде дерева, ветвящегося в будущее.



Введенные операторы являются едиными, так называемыми модализированнымитемпоральными операторами, а не комбинацией модальных и темпоральных операторов. Данный подход весьма перспективен в смысле выразительности представления, но слабо разработан в плане его практической реализации.

https://www.youtube.com/watch?v=tnySvLjgctk

http://www.isa.ru/aidt/images/documents/2011-01/14\_26.pdf