МОСКОВСКАЯ ГОСУДАРСТВЕННАЯ АКАДЕМИЯ ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ

Кафедра
“Персональные компьютеры и сети”

Ульянов М.В., Шептунов М.В.

МАТЕМАТИЧЕСКАЯ ЛОГИКА И ТЕОРИЯ АЛГОРИТМОВ

Часть II

ТЕОРИЯ АЛГОРИТМОВ

Учебное пособие

Москва
2003

УДК 519.713

Ульянов М.В., Шептунов М.В. Математическая логика и теория алгоритмов, часть 2: Теория алгоритмов. – М.: МГАПИ, 2003. – 80 с.

ISBN 5-8068-02 68 - X

Рекомендовано Ученым Советом МГАПИ в качестве учебного пособия для специальности 2201.

Рецензенты: к.т.н., проф. Зеленко Г.В.
к.т.н., проф. Рощин А.В.

Предлагаемое издание рекомендуется в качестве учебного пособия для подготовки студентов различных специальностей, изучающих математическую логику и теорию алгоритмов.

Для специальности 2201 «Вычислительные машины, комплексы, системы и сети» это издание может быть использовано в качестве учебного пособия по разделу «Теория алгоритмов» дисциплины «Математическая логика и теория алгоритмов».

Во второй части учебного пособия рассмотрены основы таких разделов теории алгоритмов как: классическая теория алгоритмов (машина Поста, машина Тьюринга, алгоритмически неразрешимые задачи), асимптотический анализ сложности алгоритмов, сложностные классы и практический сравнительный анализ вычислительных алгоритмов.

$Л \frac{240\ 402\ 0000}{ЛР020418 - 97}$

СОДЕРЖАНИЕ

В В Е Д Е Н И Е

Теория алгоритмов - это наука, изучающая общие свойства и закономерности алгоритмов, разнообразные формальные модели их представления. На основе формализации понятия алгоритма возможно сравнение алгоритмов по их эффективности, проверка их эквивалентности, определение областей применимости.

Разработанные в 1930-х годах разнообразные формальные модели алгоритмов (Пост, Тьюринг, Черч), равно как и предложенные в 1950-х годах модели Колмогорова и Маркова, оказались эквивалентными в том смысле, что любой класс проблем, разрешимых в одной модели, разрешимы и в другой.

В настоящее время полученные на основе теории алгоритмов практические рекомендации получают всё большее распространение в области проектирования и разработки программных систем. В связи с этим в государственный стандарт введёна специальная дисциплина “Математическая логика и теория алгоритмов”. Описание государственного стандарта регламентирует включение в состав дисциплины ряда понятий и методов теории алгоритмов, которые и отражены в настоящем учебном пособии.

Во второй части учебного пособия рассмотрены основы таких разделов теории алгоритмов как: классическая теория алгоритмов (машина Поста, машина Тьюринга, алгоритмически неразрешимые проблемы), асимптотический анализ сложности алгоритмов, включая сложность рекурсивных реализаций, сложностные классы P, NP, NPC, включая проблему P = NP и практический сравнительный анализ алгоритмов. На примере криптосистемы RSA показана практическая важность результатов теории алгоритмов.

1. ВВЕДЕНИЕ В ТЕОРИЮ АЛГОРИТМОВ

1.1 Исторический обзор

Первым дошедшим до нас алгоритмом в его интуитивном понимании – конечной последовательности элементарных действий, решающих поставленную задачу, считается предложенный Евклидом в III веке до нашей эры алгоритм нахождения наибольшего общего делителя двух чисел (алгоритм Евклида). Отметим, что в течение длительного времени, вплоть до начала XX века само слово «алгоритм» употреблялось в устойчивом сочетании «алгоритм Евклида». Для описания пошагового решения других математических задач использовалось слово «метод».

Начальной точкой отсчета современной теории алгоритмов можно считать работу немецкого математика Курта Гёделя [4] (1931 год - теорема о неполноте символических логик), в которой было показано, что некоторые математические проблемы не могут быть решены алгоритмами из некоторого класса. Общность результата Геделя связана с тем, совпадает ли использованный им класс алгоритмов с классом всех (в интуитивном смысле) алгоритмов. Эта работа дала толчок к поиску и анализу различных формализаций алгоритма.

Первые фундаментальные работы по теории алгоритмов были опубликованы независимо в 1936 году годы Аланом Тьюрингом, Алоизом Черчем и Эмилем Постом. Предложенные ими машина Тьюринга, машина Поста и лямбда-исчисление Черча были эквивалентными формализмами алгоритма. Сформулированные ими тезисы (Поста и Черча-Тьюринга) постулировали эквивалентность предложенных ими формальных систем и интуитивного понятия алгоритма. Важным развитием этих работ стала формулировка и доказательство алгоритмически неразрешимых проблем.

В 1950-е годы существенный вклад в теорию алгоритмов внесли работы Колмогорова и Маркова.

К 1960-70-ым годам оформились следующие направления в теории алгоритмов:

- Классическая теория алгоритмов (формулировка задач в терминах формальных языков, понятие задачи разрешения, введение сложностных классов, формулировка в 1965 году Эдмондсом проблемы *P=NP*, открытие класса *NP*-полных задач и его исследование) [6];
- Теория асимптотического анализа алгоритмов (понятие сложности и трудоёмкости алгоритма, критерии оценки алгоритмов, методы получения асимптотических оценок, в частности для рекурсивных алгоритмов, асимптотический анализ трудоемкости или времени выполнения), в развитие которой внесли существенный вклад Кнут, Ахо, Хопкрофт, Ульман, Карп [1, 2, 5];
- Теория практического анализа вычислительных алгоритмов (получение явных функции трудоёмкости, интервальный анализ функций, практические критерии качества алгоритмов, методика выбора рациональных алгоритмов), основополагающей работой в этом направлении, очевидно, следует считать фундаментальный труд Д. Кнута «Искусство программирования для ЭВМ» [5].

1.2 Цели и задачи теории алгоритмов

Обобщая результаты различных разделов теории алгоритмов можно выделить следующие цели и соотнесенные с ними задачи, решаемые в теории алгоритмов:

- формализация понятия «алгоритм» и исследование формальных алгоритмических систем;
- формальное доказательство алгоритмической неразрешимости ряда задач;
- классификация задач, определение и исследование сложностных классов;
- асимптотический анализ сложности алгоритмов;
- исследование и анализ рекурсивных алгоритмов;
- получение явных функций трудоемкости в целях сравнительного анализа алгоритмов;

- разработка критериев сравнительной оценки качества алгоритмов.

1.3 Практическое применение результатов теории алгоритмов

Полученные в теории алгоритмов теоретические результаты находят достаточно широкое практическое применение, при этом можно выделить следующие два аспекта:

Теоретический аспект: при исследовании некоторой задачи результаты теории алгоритмов позволяют ответить на вопрос – является ли эта задача в принципе алгоритмически разрешимой – для алгоритмически неразрешимых задач возможно их сведение к задаче останова машины Тьюринга. В случае алгоритмической разрешимости задачи – следующий важный теоретический вопрос – это вопрос о принадлежности этой задачи к классу *NP*–полных задач, при утвердительном ответе на который, можно говорить о существенных временных затратах для получения точного решения для больших размерностей исходных данных.

Практический аспект: методы и методики теории алгоритмов (в основном разделов асимптотического и практического анализа) позволяют осуществить:

- рациональный выбор из известного множества алгоритмов решения данной задачи с учетом особенностей их применения (например, при ограничениях на размерность исходных данных или объема дополнительной памяти);
- получение временных оценок решения сложных задач;
- получение достоверных оценок невозможности решения некоторой задачи за определенное время, что важно для криптографических методов;
- разработку и совершенствование эффективных алгоритмов решения задач в области обработки информации на основе практического анализа.

1.4 Формализация понятия алгоритма

Во всех сферах своей деятельности, и частности в сфере обработки информации, человек сталкивается с различными способами или методиками

решения задач. Они определяют порядок выполнения действий для получения желаемого результата – мы можем трактовать это как первоначальное или интуитивное определение алгоритма. Некоторые дополнительные требования приводят к неформальному определению алгоритма:

Определение 1.1: *Алгоритм* - это заданное на некотором языке конечное предписание, задающее конечную последовательность выполнимых элементарных операций для решения задачи, общее для класса возможных исходных данных.

Пусть D – область (множество) исходных данных задачи, а R – множество возможных результатов, тогда мы можем говорить, что алгоритм осуществляет отображение $D \rightarrow R$. Поскольку такое отображение может быть не полным, то вводятся следующие понятия:

Алгоритм называется частичным алгоритмом, если мы получаем результат только для некоторых $d \in D$ и полным алгоритмом, если алгоритм получает правильный результат для всех $d \in D$.

Несмотря на усилия исследователей отсутствует одно исчерпывающе строгое определение понятия алгоритм, в теории алгоритмов были введены различные формальные определения алгоритма и удивительным научным результатом является доказательство эквивалентности этих формальных определений в смысле их равномощности.

Варианты словесного определения алгоритма принадлежат российским ученым А.Н. Колмогорову и А.А. Маркову [10]:

Определение 1.2 (Колмогоров): *Алгоритм* – это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Определение 1.3 (Марков): *Алгоритм* – это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

Отметим, что различные определения алгоритма, в явной или неявной форме, постулируют следующий ряд требований:

- алгоритм должен содержать *конечное* количество элементарно выполнимых предписаний, т.е. удовлетворять требованию конечности записи;
- алгоритм должен выполнять конечное количество шагов при решении задачи, т.е. удовлетворять требованию конечности действий;
- алгоритм должен быть единым для всех допустимых исходных данных, т.е. удовлетворять требованию универсальности;
- алгоритм должен приводить к правильному по отношению к поставленной задаче решению, т.е. удовлетворять требованию правильности.

Другие формальные определения понятия алгоритма связаны с введением специальных математических конструкций (машина Поста, машина Тьюринга, рекурсивно-вычислимые функции Черча) и постулированием тезиса об эквивалентности такого формализма и понятия «алгоритм».

Рассмотрению машины Поста и машины Тьюринга посвящены следующие два раздела учебного пособия.

1.5 Вопросы для самоконтроля

1) Исторические аспекты создания и разработки теории алгоритмов;
2) Цели и задачи классической теории алгоритмов;
3) Цели и задачи теории асимптотического анализа алгоритмов;
4) Цели и задачи практического анализа алгоритмов;
5) Теоретический и практический аспекты применения результатов теории алгоритмов;
6) Формализация алгоритма, определения Колмогорова и Маркова;
7) Требования к алгоритму, связанные с формальными определениями;

2. МАШИНА ПОСТА

2.1 Основные понятия и операции

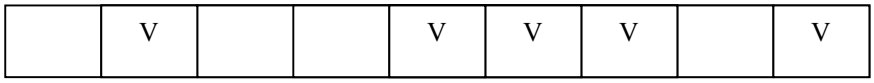
Одной из фундаментальных статей, результаты которой лежат в основе современной теории алгоритмов является статья Эмиля Поста (Emil Post), «Финитные комбинаторные процессы, формулировка 1», опубликованная в 1936 году в сентябрьском номере «Журнала символической логики» [10].

Пост рассматривает общую проблему, состоящую из множества конкретных проблем, при этом решение общей проблемы это такое решение, которое доставляет ответ для каждой конкретной проблемы.

Например, решение уравнения *3*x+9=0* – это одна из конкретных проблем, а решение уравнения *a*x+b=0* – это общая проблема, тем самым алгоритм (сам термин «алгоритм» не используется Постом) должен быть универсальным, т.е. должен быть соотнесен с общей проблемой.

Основные понятия алгоритмического формализма Поста – это пространство символов (язык *L*) в котором задаётся конкретная проблема и получается ответ, и набор инструкций, т.е. операций в пространстве символов, задающих как сами операции, так и порядок выполнения инструкций.

Постовское пространство символов – это бесконечная лента ячеек (ящиков):



Каждый ящик или ячейка могут быть помечены или не помечены.

Конкретная проблема задается «внешней силой» (термин Поста) пометкой конечного количества ячеек, при этом, очевидно, что любая конфигурация начинается и заканчивается помеченным ящиком. После применения к конкретной проблеме некоторого набора инструкций решение представляется так же в виде набора помеченных и непомеченных ящиков, распознаваемое той же внешней силой.

Пост предложил набор инструкций (элементарных операций), которые выполняет «работник», отметим, что в 1936 году не было еще ни одной электронной вычислительной машины. Этот набор инструкций является, очевидно, минимальным набором битовых операций:

1. пометить ящик, если он пуст;
2. стереть метку, если она есть;
3. переместиться влево на 1 ящик;
4. переместиться вправо на 1 ящик;
5. определить помечен ящик или нет, и по результату перейти на одну из двух указанных инструкций;
6. остановиться.

Отметим, что формулировка инструкций 1 и 2 включает защиту от неправильных ситуаций.

Программа представляет собой нумерованную последовательность инструкций, причем переходы в инструкции 5 производятся на указанные в ней номера других инструкций.

2.2 Финитный 1 – процесс

Программа (набор инструкций в терминах Поста) является одной и той же для всех конкретных проблем, поэтому соотнесена с общей проблемой – таким образом, Пост формулирует требование универсальности.

Далее Пост вводит следующие понятия:

- набор инструкций *применим* к общей проблеме, если для каждой конкретной проблемы не возникает коллизий в инструкциях 1 и 2, т.е. никогда программа не стирает метку в пустом ящике и не устанавливает метку в помеченном ящике;
- набор инструкций *заканчивается* (за конечное количество инструкций), если выполняется инструкция (6);
- набор инструкций задаёт *финитный 1 – процесс*, если набор применим, и заканчивается для каждой конкретной проблемы;

- финитный 1 – процесс для общей проблемы есть 1 – решение, если ответ для каждой конкретной проблемы правильный (это определяется внешней силой).

2.3 Способ задания проблемы и формулировка 1

По Посту проблема задаётся внешней силой путем пометки конечного количества ящиков ленты. В более поздних работах по машине Поста [10] принято считать, что машина работает в единичной системе счисления (*0=V; 1=VV; 2=VVV; 3=VVVV*), т.е. ноль представляется одним помеченным ящиком, а целое положительное число – помеченными ящиками в количестве на единицу больше его значения.

Поскольку множество конкретных проблем, составляющих общую проблему счетное, то можно установить взаимно однозначное соответствие (биективное отображение) между множеством положительных целых чисел $\boldsymbol{N}$ и множеством конкретных проблем.

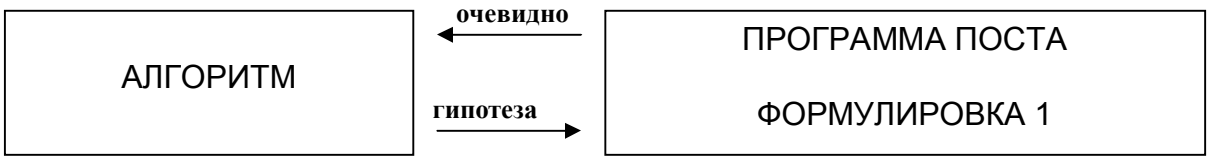
Общая проблема называется по Посту *1-заданой,* если существует такой финитный 1 – процесс, что, будучи, применим к $n \in \boldsymbol{N}$ в качестве исходной конфигурации ящиков, он задает *n-ую* конкретную проблему в виде набора помеченных ящиков.

Если общая проблема 1-задана и 1-разрешима, то, соединяя наборы инструкций по заданию проблемы, и ее решению мы получаем ответ по номеру проблемы – это и есть в терминах статьи Поста *формулировка 1*.

Эмиль Пост завершает свою статью следующей фразой [10]: «Автор ожидает, что его формулировка окажется логически эквивалентной рекурсивности в смысле Геделя — Черча. Цель формулировки, однако, в том, чтобы предложить систему не только определенной логической силы, но и психологической достоверности. В этом последнем смысле подлежат рассмотрению всё более и более широкие формулировки. С другой стороны, нашей целью будет показать, что все они логически сводимы к формулировке 1. В настоя-

щий момент мы выдвигаем это умозаключение в качестве *рабочей гипотезы*. … Успех вышеизложенной программы заключался бы для нас в превращении этой гипотезы не столько в определение или аксиому, сколько в закон природы».

Таким образом, гипотеза Поста состоит в том, что любые более широкие формулировки в смысле алфавита символов ленты, набора инструкций, представления и интерпретации конкретных проблем сводимы к формулировке 1.



Следовательно, если гипотеза верна, то любые другие формальные определения, задающие некоторый класс алгоритмов, эквивалентны классу алгоритмов, заданных формулировкой 1 Эмиля Поста.

Обоснование этой гипотезы происходит сегодня не путем строго математического доказательства, а на пути эксперимента — действительно, всякий раз, когда нам указывают алгоритм, его можно перевести в форму программы машины Поста, приводящей к тому же результату.

2.4 Вопросы для самоконтроля

1) Понятие общей и конкретной проблемы по Посту;
2) Пространство символов и примитивные операции в машине Поста;
3) Понятие финитного 1-процесса в машине Поста;
4) Способы задания проблем и формулировка 1;
5) Гипотеза Поста;

3. МАШИНА ТЬЮРИНГА И АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ ПРОБЛЕМЫ

3.1. Машина Тьюринга

Алан Тьюринг (Turing) в 1936 году опубликовал в трудах Лондонского математического общества статью «О вычислимых числах в приложении к проблеме разрешения», которая наравне с работами Поста и Черча лежит в основе современной теории алгоритмов.

Предыстория создания этой работы связана с формулировкой Давидом Гильбертом на Международном математическом конгрессе в Париже в 1900 году неразрешенных математических проблем. Одной из них была задача доказательства непротиворечивости системы аксиом обычной арифметики, которую Гильберт в дальнейшем уточнил как «проблему разрешимости» - нахождение общего метода, для определения выполнимости данного высказывания на языке формальной логики.

Статья Тьюринга как раз и давала ответ на эту проблему - вторая проблема Гильберта оказалась неразрешимой. Но значение статьи Тьюринга выходило далеко за рамки той задачи, по поводу которой она была написана.

Приведем характеристику этой работы, принадлежащую Джону Хопкрофту [4]: «Работая над проблемой Гильберта, Тьюрингу пришлось дать четкое определение самого понятия метода. Отталкиваясь от интуитивного представления о методе как о некоем алгоритме, т.е. процедуре, которая может быть выполнена механически, без творческого вмешательства, он показал, как эту идею можно воплотить в виде подробной модели вычислительного процесса. Полученная модель вычислений, в которой каждый алгоритм разбивался на последовательность простых, элементарных шагов, и была логической конструкцией, названной впоследствии машиной Тьюринга».

Машина Тьюринга является расширением модели конечного автомата, расширением, включающим потенциально бесконечную память с возможно-

стью перехода (движения) от обозреваемой в данный момент ячейки к ее левому или правому соседу [4].

Формально машина Тьюринга может быть описана следующим образом:

Пусть заданы:

- конечное множество состояний – Q, в которых может находиться машина Тьюринга;
- конечное множество символов ленты – Γ;
- функция δ (функция переходов или программа), которая задается отображением пары из декартова произведения $Q \times \Gamma$ (машина находится в состоянии q_i и обозревает символ γ_i) в тройку декартова произведения $Q \times \Gamma \times \{L,R\}$ (машина переходит в состояние q_j, заменяет символ γ_i на символ γ_j и передвигается влево или вправо на один символ ленты) – $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$
- один символ из $\Gamma \rightarrow e$ (пустой);
- подмножество $\Sigma \in \Gamma \rightarrow$ определяется как подмножество входных символов ленты, причем $e \in (\Gamma - \Sigma)$;
- одно из состояний – $q_0 \in Q$ является начальным состоянием машины.

Решаемая проблема задается путем записи конечного количества символов из множества $\Sigma \in \Gamma$ – $Si \in \Sigma$ на ленту:

e	S1	S2	S3	S4	……	Sn	e

после чего машина переводится в начальное состояние и головка устанавливается у самого левого непустого символа – $(q_0, \uparrow\omega)$, после чего в соответствии с указанной функцией переходов $(q_i, S_i) \rightarrow (q_j, S_k, L \text{ или } R)$ машина начинает заменять обозреваемые символы, передвигать головку вправо или влево и переходить в другие состояния, предписанные функций переходов.

Остановка машины происходит в том случае, если для пары (q_i, S_i) функция перехода не определена.

Алан Тьюринг высказал предположение, что любой алгоритм в интуитивном смысле этого слова может быть представлен эквивалентной машиной Тьюринга. Это предположение известно как тезис Черча–Тьюринга. Каждый компьютер может моделировать машину Тьюринга (операции перезаписи ячеек, сравнения и перехода к другой соседней ячейке с учетом изменения состояния машины). Следовательно, он может моделировать алгоритмы в любом формализме, и из этого тезиса следует, что все компьютеры (независимо от мощности, архитектуры и т.д.) эквивалентны с точки зрения принципиальной возможности решения алгоритмических задач.

3.2. Алгоритмически неразрешимые проблемы

За время своего существования человечество придумало множество алгоритмов для решения разнообразных практических и научных проблем. Зададимся вопросом – а существуют ли какие-нибудь проблемы, для которых невозможно придумать алгоритмы их решения?

Утверждение о существовании алгоритмически неразрешимых проблем является весьма сильным – мы констатируем, что мы не только сейчас на знаем соответствующего алгоритма, но мы не можем принципиально никогда его найти.

Успехи математики к концу XIX века привели к формированию мнения, которое выразил Д. Гильберт – «в математике не может быть неразрешимых проблем», в связи с этим формулировка проблем Гильбертом на конгрессе 1900 года в Париже была руководством к действию, констатацией отсутствия решений в данный момент.

Первой фундаментальной теоретической работой, связанной с доказательством алгоритмической неразрешимости, была работа Курта Гёделя – его известная теорема о неполноте символических логик. Это была строго формулированная математическая проблема, для которой не существует решающего ее алгоритма. Усилиями различных исследователей список алгоритмически

неразрешимых проблем был значительно расширен. Сегодня принято при доказательстве алгоритмической неразрешимости некоторой задачи сводить ее к ставшей классической задаче – «задаче останова».

Имеет место быть следующая теорема (доказательство в [4]):

Теорема 3.1. Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий, т.е. невозможностью предсказать, что для любых исходных данных решение будет получено за конечное количество шагов.

Тем не менее, можно попытаться сформулировать причины, ведущие к алгоритмической неразрешимости, эти причины достаточно условны, так как все они сводимы к проблеме останова, однако такой подход позволяет более глубоко понять природу алгоритмической неразрешимости:

а) Отсутствие общего метода решения задачи

Проблема 1: Распределение девяток в записи числа π [10];

Определим функцию $f(n) = i$, где n – количество девяток подряд в десятичной записи числа π, а i – номер самой левой девятки из n девяток подряд:

$\pi=3,141592\ldots$ $\qquad\qquad$ $f(1) = 5.$

Задача состоит в вычислении функции $f(n)$ для произвольно заданного n.

Поскольку число π является иррациональным и трансцендентным, то мы не знаем никакой информации о распределении девяток (равно как и любых других цифр) в десятичной записи числа π. Вычисление $f(n)$ связано с вычислением последующих цифр в разложении π, до тех пор, пока мы не обнаружим n девяток подряд, однако у нас нет общего метода вычисления $f(n)$, поэтому для некоторых n вычисления могут продолжаться бесконечно – мы даже не знаем в принципе (по природе числа π) существует ли решение для всех n.

Проблема 2: Вычисление совершенных чисел;

Совершенные числа – это числа, которые равны сумме своих делителей, например: *28 = 1+2+4+7+14*.

Определим функцию *S(n) = n-ое* по счёту совершенное число и поставим задачу вычисления *S(n)* по произвольно заданному *n*. Нет общего метода вычисления совершенных чисел, мы даже не знаем, множество совершенных чисел конечно или счетно, поэтому наш алгоритм должен перебирать все числа подряд, проверяя их на совершенность. Отсутствие общего метода решения не позволяет ответить на вопрос о останове алгоритма. Если мы проверили *M* чисел при поиске n-ого совершенного числа – означает ли это, что его вообще не существует?

Проблема 3: Десятая проблема Гильберта;

Пусть задан многочлен *n*-ой степени с целыми коэффициентами – *P*, существует ли алгоритм, который определяет, имеет ли уравнение *P=0* решение в целых числах?

Ю.В. Матиясевич [4] показал, что такого алгоритма не существует, т.е. отсутствует общий метод определения целых корней уравнения *P=0* по его целочисленным коэффициентам.

б) Информационная неопределенность задачи

Проблема 4: Позиционирование машины Поста на последний помеченный ящик [10];

Пусть на ленте машины Поста заданы наборы помеченных ящиков (кортежи) произвольной длины с произвольными расстояниями между кортежами и головка находится у самого левого помеченного ящика. Задача состоит установке головки на самый правый помеченный ящик последнего кортежа.

Попытка построения алгоритма, решающего эту задачу приводит к необходимости ответа на вопрос – когда после обнаружения конца кортежа мы сдвинулись вправо по пустым ящикам на M позиций и не обнаружили начало следующего кортежа – больше на ленте кортежей нет или они есть где-то пра-

вее? Информационная неопределенность задачи состоит в отсутствии информации либо о количестве кортежей на ленте, либо о максимальном расстоянии между кортежами – при наличии такой информации (при разрешении информационной неопределенности) задача становится алгоритмически разрешимой.

в) Логическая неразрешимость (в смысле теоремы Гёделя о неполноте)

Проблема 5: Проблема «останова» (см. теорема 3.1);

Проблема 6: Проблема эквивалентности алгоритмов;

По двум произвольным заданным алгоритмам (например, по двум машинам Тьюринга) определить, будут ли они выдавать одинаковые выходные результаты на любых исходных данных.

Проблема 7: Проблема тотальности;

По произвольному заданному алгоритму определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи – является ли частичный алгоритм P всюду определённым?

3.3. Проблема соответствий Поста над алфавитом Σ

В качестве более подробного примера алгоритмически неразрешимой задачи рассмотрим проблему соответствий Поста [1] (Э. Пост, 1943 г.). Мы выделили эту задачу, поскольку на первый взгляд она выглядит достаточно «алгоритмизуемой», однако она сводима к проблеме останова и является алгоритмически неразрешимой.

Постановка задачи:

Пусть дан алфавит Σ: $|\Sigma| \geq 2$ (для одно-символьного алфавита задача имеет решение) и дано конечное множество пар из $\Sigma^+ x\ \Sigma^+$, т.е. пары непустых цепочек произвольного языка над алфавитом Σ: $(x_1,y_1), \ldots\ldots, (x_m,y_m)$.

Проблема: Выяснить существует ли конечная последовательность этих пар, не обязательно различных, такая что цепочка, составленная из левых подцепочек, совпадает с последовательностью правых подцепочек – такая последовательность называется решающей.

В качестве примера рассмотрим $\Sigma = \{a,b\}$

1. Входные цепочки: (abbb, b), (a, aab), (ba, b)

Решающая последовательность для этой задачи имеет вид:

(a,aab) (a,aab) (ba,b) (abbb,b), так как : a a ba abbb ≡ aab aab b b

2. Входные цепочки: (ab,aba), (aba,baa), (baa,aa)

Данная задача вообще не имеет решения, так как нельзя начинать с пары *(aba,baa)* или *(baa,aa)*, поскольку не совпадают начальные символы подцепочек, но если начинать с цепочки *(ab,aba),* то в последующем не будет совпадать общее количество символов «а», т.к. в других двух парах количество символов «а» одинаково.

В общем случае мы можем построить частичный алгоритм, основанный на идее упорядоченной генерации возможных последовательностей цепочек (отметим, что мы имеем счетное множество таких последовательностей) с проверкой выполнения условий задачи. Если последовательность является решающей – то мы получаем результативный ответ за конечное количество шагов. Поскольку общий метод определения отсутствия решающей последовательности не может быть указан, т.к. задача сводима к проблеме «останова» и, следовательно, является алгоритмически неразрешимой, то при отсутствии решающей последовательности алгоритм порождает бесконечный цикл.

В теории алгоритмов такого рода проблемы, для которых может быть предложен частичный алгоритм их решения, частичный в том смысле, что он возможно, но не обязательно, за конечное количество шагов находит решение проблемы, называются *частично разрешимыми проблемами.*

В частности, проблема останова так же является частично разрешимой проблемой, а проблемы эквивалентности и тотальности не являются таковыми.

3.4 Вопросы для самоконтроля

1) Формальное описание машины Тьюринга;
2) Функция переходов в машине Тьюринга;
3) Понятие об алгоритмически неразрешимых проблемах
4) Проблема позиционирования в машине Поста;
5) Проблема соответствий Поста над алфавитом Σ;
6) Проблема останова в машине Тьюринга;
7) Проблема эквивалентности и тотальности;

4. ВВЕДЕНИЕ В АНАЛИЗ АЛГОРИТМОВ

4.1. Сравнительные оценки алгоритмов

При использовании алгоритмов для решения практических задач мы сталкиваемся с проблемой рационального выбора алгоритма решения задачи. Решение проблемы выбора связано с построением системы сравнительных оценок, которая в свою очередь существенно опирается на формальную модель алгоритма.

Будем рассматривать в дальнейшем, придерживаясь определений Поста, применимые к общей проблеме, правильные и финитные алгоритмы, т.е. алгоритмы, дающие *1-решение* общей проблемы. В качестве формальной системы будем рассматривать абстрактную машину, включающую процессор с фон-Неймановской архитектурой, поддерживающий адресную память и набор «элементарных» операций соотнесенных с языком высокого уровня.

В целях дальнейшего анализа примем следующие допущения:

- каждая команда выполняется не более чем за фиксированное время;
- исходные данные алгоритма представляются машинными словами по β битов каждое.

Конкретная проблема задается N словами памяти, таким образом, на входе алгоритма – $N_\beta = N*\beta$ бит информации. Отметим, что в ряде случаев, особенно при рассмотрении матричных задач N является мерой длины входа алгоритма, отражающей линейную размерность.

Программа, реализующая алгоритм для решения общей проблемы состоит из M машинных инструкций по $\beta_м$ битов – $M_\beta = M*\beta_м$ бит информации.

Кроме того, алгоритм может требовать следующих дополнительных ресурсов абстрактной машины:

- S_d – память для хранения промежуточных результатов;
- S_r – память для организации вычислительного процесса (память, необходимая для реализации рекурсивных вызовов и возвратов).

При решении конкретной проблемы, заданной N словами памяти алгоритм выполняет не более, чем конечное количество «элементарных» операций абстрактной машины в силу условия рассмотрения только финитных алгоритмов. В связи с этим введем следующее определение:

Определение 4.1. *Трудоёмкость алгоритма.*

Под трудоёмкостью алгоритма для данного конкретного входа – $F_a(N)$, будем понимать количество «элементарных» операций совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе.

Комплексный анализ алгоритма может быть выполнен на основе комплексной оценки ресурсов формальной системы, требуемых алгоритмом для решения конкретных проблем. Очевидно, что для различных областей применения веса ресурсов будут различны, что приводит к следующей комплексной оценке алгоритма:

$\psi_A = c_1 * F_a(N) + c_2 * M + c_3 * S_d + c_4 * S_r,$ где c_i – веса ресурсов**.**

4.2 Система обозначений в анализе алгоритмов

При более детальном анализе трудоемкости алгоритма оказывается, что не всегда количество элементарных операций, выполняемых алгоритмом на одном входе длины N, совпадает с количеством операций на другом входе такой же длины. Это приводит к необходимости введения специальных обозначений, отражающих поведение функции трудоемкости данного алгоритма на входных данных фиксированной длины.

Пусть D_A – множество конкретных проблем данной задачи, заданное в формальной системе. Пусть $D \in D_A$ – задание конкретной проблемы и $|D| = N$.

В общем случае существует собственное подмножество множества D_A, включающее все конкретные проблемы, имеющие мощность N:

обозначим это подмножество через D_N: $D_N = \{D \in D_N,: |D| = N\}$;

обозначим мощность множества D_N через $M_{DN} \rightarrow M_{DN} = |D_N|$.

Тогда содержательно данный алгоритм, решая различные задачи размерности *N*, будет выполнять в каком-то случае наибольшее количество операций, а в каком-то случае наименьшее количество операций. Ведем следующие обозначения:

1. $\boldsymbol{F_a^{\wedge}(N)}$ – *худший случай* – наибольшее количество операций, совершаемых алгоритмом *A* для решения конкретных проблем размерностью *N*:

$$F_a^{\wedge}(N) = \max_{D \in D_N} \{F_a(D)\} \text{ – худший случай на } D_N$$

2. $\boldsymbol{F_a^{\vee}(N)}$ – *лучший случай* – наименьшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью *N*:

$$F_a^{\vee}(N) = \min_{D \in D_N} \{F_a(D)\} \text{ – лучший случай на } D_N$$

3. $\boldsymbol{\bar{F}_a(N)}$ – *средний случай* – среднее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерностью *N:*

$$\bar{F}_a(N) = (1 / M_{DN}) * \sum_{D \in D_N} \{F_a(D)\} \text{ – средний случай на } D_N$$

4.3 Классификация алгоритмов по виду функции трудоёмкости

В зависимости от влияния исходных данных на функцию трудоемкости алгоритма может быть предложена следующая классификация, имеющая практическое значение для анализа алгоритмов:

1.Количественно-зависимые по трудоемкости алгоритмы

Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа, и не зависит от конкретных значений:

$$F_a(D) = F_a(|D|) = F_a(N)$$

Примерами алгоритмов с количественно-зависимой функцией трудоемкости могут служить алгоритмы для стандартных операций с массивами и матрицами – умножение матриц, умножение матрицы на вектор и т.д.

2.Параметрически-зависимые по трудоемкости алгоритмы

Это алгоритмы, трудоемкость которых определяется не размерностью входа (как правило, для этой группы размерность входа обычно фиксирована), а конкретными значениями обрабатываемых слов памяти:

$$F_a\,(D) = F_a\,(d_1,\ldots,d_n) = F_a\,(P_1,\ldots,P_m),\ m \le n$$

Примерами алгоритмов с параметрически-зависимой трудоемкостью являются алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов. Очевидно, что такие алгоритмы, имея на входе два числовых значения – аргумент функции и точность выполняют существенно зависящее от значений количество операций.

а) Вычисление x^k последовательным умножением $\Rightarrow F_a(x,\ k) = F_a\,(k)$.

б) Вычисление $e^x = \sum(x^n/n!)$, с точностью до $\xi \Rightarrow F_a = F_a\,(x,\ \xi)$

3. Количественно-параметрические по трудоемкости алгоритмы

Однако в большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных, в этом случае:

$$F_a\,(D) = F_a\,(||D||,\ P_1,\ldots,P_m) = F_a\,(N,\ P_1,\ldots,P_m)$$

В качестве примера можно привести алгоритмы численных методов, в которых параметрически-зависимый внешний цикл по точности включает в себя количественно-зависимый фрагмент по размерности.

3.1 Порядково-зависимые по трудоемкости алгоритмы

Среди разнообразия параметрически-зависимых алгоритмов выделим еще оду группу, для которой количество операций зависит от порядка расположения исходных объектов.

Пусть множество D состоит из элементов $(d_1,\ldots,d_n)$, *и* $||D||=N$,

Определим $D_p = \{(d_1,\ldots,d_n)\}$-множество всех упорядоченных N-*ок* из $d_1,\ldots,d_n$, отметим, что $|D_p|=n!$.

Если $F_a\ ({}_iD_p) \neq F_a\ ({}_jD_p)$, где ${}_iD_p,\ {}_jD_p \in D_p$, то алгоритм будем называть порядково-зависимым по трудоемкости.

Примерами таких алгоритмов могут служить ряд алгоритмов сортировки, алгоритмы поиска минимума и максимума в массиве. Рассмотрим более подробно алгоритм поиска максимума в массиве *S*, содержащим *n* элементов:

MaxS (S,n; Max)

Max $\leftarrow S_1$

For i$\leftarrow$*2 to n*

if Max $< S_i$

then Max $\leftarrow S_i$ (количество выполненных операций присваивания зависит от порядка следования элементов массива)

4.4 Асимптотический анализ функций

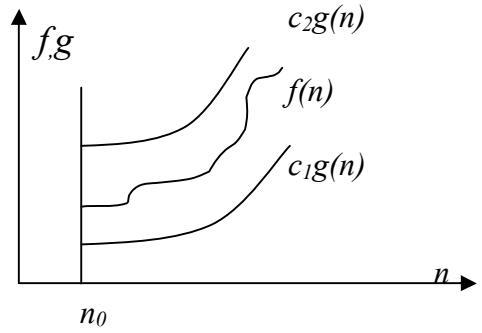
При анализе поведения функции трудоемкости алгоритма часто используют принятые в математике асимптотические обозначения, позволяющие показать скорость роста функции, маскируя при этом конкретные коэффициенты.

Такая оценка функции трудоемкости алгоритма называется *сложностью алгоритма* и позволяет определить предпочтения в использовании того или иного алгоритма для больших значений размерности исходных данных.

В асимптотическом анализе приняты следующие обозначения [6]:

1. Оценка Θ (тетта)

Пусть *f(n)* и *g(n)* – положительные функции положительного аргумента, $n \geq 1$ (количество объектов на входе и количество операций – положительные числа), тогда:



$f(n) = \Theta\ (g(n))$, если существуют положительные c_1, c_2, n_0, такие, что:

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$, при $n > n_0$

Обычно говорят, что при этом функция *g(n)* является асимптотически точной оценкой функции *f(n),* т.к. по определению функция *f(n)* не отличается от функции *g(n)* с точностью до постоянного множителя.

Отметим, что из $f(n) = \Theta(g(n))$ следует, что $g(n) = \Theta(f(n))$.

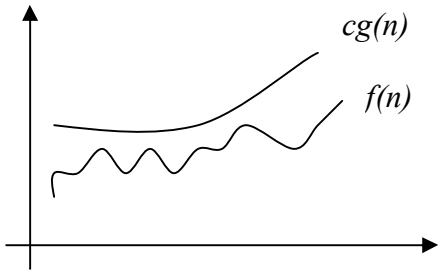
Примеры:

1) $f(n)=4n^2+nlnN+174 – f(n)= \Theta(n^2);$

2) $f(n)=\Theta(1)$ – запись означает, что *f(n)* или равна константе, не равной нулю, или *f(n)* ограничена константой на $\infty : f(n) = 7+1/n = \Theta(1)$.

2. Оценка *O* (*O* большое)

В отличие от оценки *Θ,* оценка *O* требует только, что бы функция *f(n)* не превышала *g(n)* начиная с $n > n0$, с точностью до постоянного множителя:



$$\exists\, c > 0,\ n0 > 0 :$$
$$0 \le f(n) \le c * g(n),\ \forall n > n0$$

Вообще, запись *O(g(n))* обозначает класс функций, таких, что все они растут не быстрее, чем функция *g(n)* с точностью до постоянного множителя, поэтому иногда говорят, что *g(n)* мажорирует функцию *f(n)*.

Например, для всех функций:

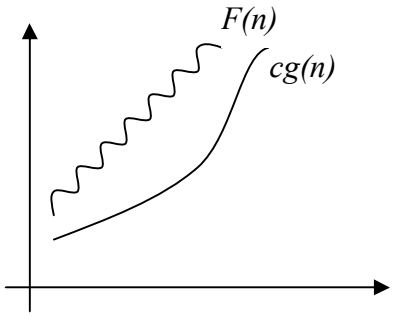
$f(n)=1/n,\ f(n)= 12,\ f(n)=3*n+17,\ f(n)=n*Ln(n),\ f(n)=6*n^2 +24*n+77$

будет справедлива оценка $O(n^2)$

Указывая оценку *O* есть смысл указывать наиболее «близкую» мажорирующую функцию, поскольку например для $f(n)= n^2$ справедлива оценка $O(2^n)$, однако она не имеет практического смысла.

3. Оценка Ω (Омега)

В отличие от оценки *O*, оценка Ω является оценкой снизу – т.е. определяет класс функций, которые растут не медленнее, чем *g(n)* с точностью до постоянного множителя:



$$\exists\, c > 0,\ n0 > 0 :$$

$$0 \le c * g(n) \le f(n)$$

Например, запись $\Omega(n*Ln(n))$ обозначает класс функций, которые растут не медленнее, чем $g(n) = n*Ln(n)$, в этот класс попадают все полиномы со степенью большей единицы, равно как и все степенные функции с основанием большим единицы.

Асимптотическое обозначение *O* восходит к учебнику Бахмана по теории простых чисел (Bachman, 1892), обозначения Θ, Ω введены Д. Кнутом- (Donald Knuth) [6].

Отметим, что не всегда для пары функций справедливо одно из асимптотических соотношений, например для $f(n)=n^{1+sin(n)}$ и $g(n)=n$ не выполняется ни одно из асимптотических соотношений.

В асимптотическом анализе алгоритмов разработаны специальные методы получения асимптотических оценок, особенно для класса рекурсивных алгоритмов. Очевидно, что Θ оценка является более прдпочтительной, чем оценка *O*. Знание асимптотики поведения функции трудоемкости алгоритма - его сложности, дает возможность делать прогнозы по выбору более рационального с точки зрения трудоемкости алгоритма для больших размерностей исходных данных.

4.5 Вопросы для самоконтроля

1) Формальная система языка высокого уровня;
2) Понятие трудоемкости алгоритма в формальном базисе;
3) Обобщенный критерий оценки качества алгоритма,
4) Система обозначений в анализе алгоритмов - худший, лучший и средний случаи;
5) Классификация алгоритмов по виду функции трудоемкости;
6) Примеры количественных и параметрически–зависимых алгоритмов;
7) Обозначения в асимптотическом анализе функций;
8) Примеры функций, не связанных асимптотическими обозначениями;

5. ТРУДОЕМКОСТЬ АЛГОРИТМОВ И ВРЕМЕННЫЕ ОЦЕНКИ

5.1. Элементарные операции в языке записи алгоритмов

Для получения функции трудоемкости алгоритма, представленного в формальной системе введенной абстрактной машины необходимо уточнить понятия «элементарных» операций, соотнесенных с языком высокого уровня.

В качестве таких «элементарных» операций предлагается использовать следующие:

1) Простое присваивание: $a \leftarrow b$;

2) Одномерная индексация *a[i]* : (адрес *(a)*+*i**длинна элемента);

3) Арифметические операции: *(*, /, -, +);*

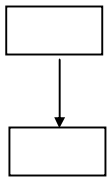
4) Операции сравнения: $a < b$;

5) Логические операции *(l1) {or, and, not} (l2);*

Опираясь на идеи структурного программирования, исключим команду перехода по адресу, считая ее связанной с операцией сравнения в конструкции ветвления.

После введения элементарных операций анализ трудоемкости основных алгоритмических конструкций в общем виде сводится к следующим положениям:

А) Конструкция «Следование»

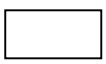


Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом.

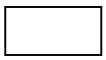
$F_{«следование»} = f_1 + \ldots + f_k$, *где* k – количество блоков.

В) Конструкция «Ветвление»

if (l) then

 f_{then} с вероятностью p

else

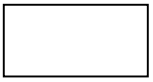
 f_{else} с вероятностью $(1-p)$

Общая трудоемкость конструкции «Ветвление» требует анализа вероятности выполнения переходов на блоки «Then» и «Else» и определяется как:

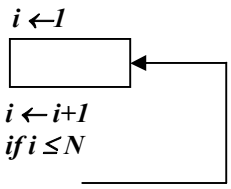
$$F_{«ветвление»} = f_{then} * p + f_{else} * (1-p).$$

C) Конструкция «Цикл»

for i ← 1 to N



end



После сведения конструкции к элементарным операциям ее трудоемкость определяется как:

$$F_{«цикл»} = 1+3*N+N*f_{«тела\ цикла»}$$

5.2 Примеры анализа простых алгоритмов

Пример 1 Задача суммирования элементов квадратной матрицы

```
SumM (A, n; Sum)
Sum ← 0
For i ← 1 to n
        For j ← 1 to n
                Sum ← Sum + A[i,j]
end for
Return (Sum)
End
```

Алгоритм выполняет одинаковое количество операций при фиксированном значении n, и следовательно является количественно-зависимым. Применение методики анализа конструкции «Цикл » дает:

$F_A(n)=1+1+ n *(3+1+ n *(3+4))=7 n^2+4* n +2 = \Theta(n^2)$, заметим, что под n понимается линейная размерность матрицы, в то время как на вход алгоритма подается n^2 значений.

Пример 2 Задача поиска максимума в массиве

```
MaxS (S,n; Max)
Max ← S[1]
For i ← 2 to n
        if Max < S[i]
                then   Max ← S[i]
end for
return Max
End
```

Данный алгоритм является количественно-параметрическим, поэтому для фиксированной размерности исходных данных необходимо проводить анализ для худшего, лучшего и среднего случая.

А). Худший случай

Максимальное количество переприсваиваний максимума (на каждом проходе цикла) будет в том случае, если элементы массива отсортированы по возрастанию. Трудоемкость алгоритма в этом случае равна:

$$F_A^{\wedge}(n)=1+1+1+ (n-1)\ (3+2+2)=7\ n - 4 = \Theta(n).$$

Б) Лучший случай

Минимальное количество переприсваивания максимума (ни одного на каждом проходе цикла) будет в том случае, если максимальный элемент расположен на первом месте в массиве. Трудоемкость алгоритма в этом случае равна:

$$F_A^{\vee}(n)=1+1+1+ (n-1)\ (3+2)=5\ n - 2 = \Theta(n).$$

В) Средний случай

Алгоритм поиска максимума последовательно перебирает элементы массива, сравнивая текущий элемент массива с текущим значением максимума. На очередном шаге, когда просматривается *к-ый* элемент массива, переприсваивание максимума произойдет, если в подмассиве из первых *к* элементов максимальным элементом является последний. Очевидно, что в случае равно-

мерного распределения исходных данных, вероятность того, что максимальный из *к* элементов расположен в определенной (последней) позиции равна *1/к*. Тогда в массиве из *n* элементов общее количество операций переприсваивания максимума определяется как:

$$\sum_{i=1}^{N} 1/i = Hn \approx Ln(N) + \gamma, \quad \gamma \approx 0{,}57$$

Величина *Hn* называется *n-ым* гармоническим числом. Таким образом, точное значение (математическое ожидание) среднего количества операций присваивания в алгоритме поиска максимума в массиве из *n* элементов определяется величиной *Hn* (на бесконечности количества испытаний), тогда:

$$\bar{F}_A(n)=1 + (n-1)\ (3+2) + 2\ (Ln(n) + \gamma)=5\ n +2\ Ln(n) - 4 +2\ \gamma = \Theta(n).$$

5.3. Переход к временным оценкам

Сравнение двух алгоритмов по их функции трудоемкости вносит некоторую ошибку в получаемые результаты. Основной причиной этой ошибки является различная частотная встречаемость элементарных операций, порождаемая разными алгоритмами и различие во временах выполнения элементарных операций на реальном процессоре. Таким образом, возникает задача перехода от функции трудоемкости к оценке времени работы алгоритма на конкретном процессоре:

Дано: $F_A(D_A)$ - трудоёмкость алгоритма требуется определить время работы программной реализации алгоритма – $T_A(D_A)$.

На пути построения временных оценок мы сталкиваемся с целым набором различных проблем, пофакторный учет которых вызывает существенные трудности. Укажем основные из этих проблем:

- неадекватность формальной системы записи алгоритма и реальной системы команд процессора;
- наличие архитектурных особенностей существенно влияющих на наблюдаемое время выполнения программы, таких как конвейер, кеширование памяти, предвыборка команд и данных, и т.д.;

- различные времена выполнения реальных машинных команд;
- различие во времени выполнения одной команды, в зависимости от значений операндов
- различные времена реального выполнения однородных команд в зависимости от типов данных;
- неоднозначности компиляции исходного текста, обусловленные как самим компилятором, так и его настройками.

Попытки различного подхода к учету этих факторов привели к появлению разнообразных методик перехода к временным оценкам.

1) Пооперационный анализ

Идея пооперационного анализа состоит в получении пооперационной функции трудоемкости для каждой из используемых алгоритмом элементарных операций с учетом типов данных. Следующим шагом является экспериментальное определение среднего времени выполнения данной элементарной операции на конкретной вычислительной машине. Ожидаемое время выполнения рассчитывается как сумма произведений пооперационной трудоемкости на средние времена операций:

$$T_A(N) = \sum F_{aoni}(N) * \bar{t}_{oni}$$

2) Метод Гиббсона

Метод предполагает проведение совокупного анализа по трудоемкости и переход к временным оценкам на основе принадлежности решаемой задачи к одному из следующих типов:

- задачи научно-технического характера с преобладанием операций с операндами действительного типа;
- задачи дискретной математики с преобладанием операций с операндами целого типа
- задачи баз данных с преобладанием операций с операндами строкового типа

Далее на основе анализа множества реальных программ для решения соответствующих типов задач определяется частотная встречаемость операций (рис 5.1), создаются соответствующие тестовые программы, и определяется среднее время на операцию в данном типе задач – $\bar{t}_{тип\ задачи}$.

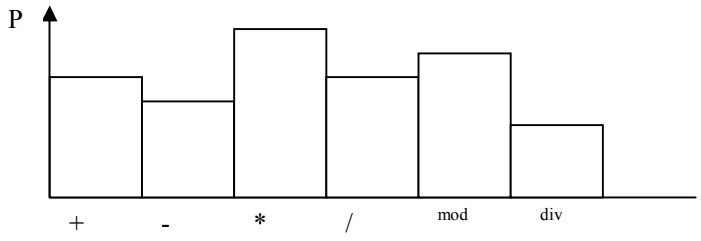


Рис 5.1 Возможный вид частотной встречаемости операций

На основе полученной информации оценивается общее время работы алгоритма в виде:

$$T_A(N) = F_A(N) * \bar{t}_{тип\ задачи}$$

3) Метод прямого определения среднего времени

В этом методе так же проводится совокупный анализ по трудоемкости – определяется $F_A(N)$, после чего на основе прямого эксперимента для различных значений $N_э$ определяется среднее время работы данной программы $T_э$ и на основе известной функции трудоемкости рассчитывается среднее время на обобщенную элементарную операцию, порождаемое данным алгоритмом, компилятором и компьютером – $\bar{t}_a$. Эти данные могут быть (в предположении об устойчивости среднего времени по N) интерполированы или экстраполированы на другие значения размерности задачи следующим образом:

$$\bar{t}_a = T_э(N_э) / F_A(N_э),\ T(N) = \bar{t}_a * F_A(N).$$

5.4 Пример пооперационного временного анализа

В ряде случаев именно пооперационный анализ позволяет выявить тонкие аспекты рационального применения того или иного алгоритма решения задачи. В качестве примера рассмотрим задачу умножения двух комплексных чисел:

$$(a+bi)*(c+di)=(ac - bd) + i(ad + bc)=e + if$$

1. Алгоритм A1 (прямое вычисление e, f – четыре умножения)

MultComplex1 (a, b, c, d; e, f)

*e←a*c - b*d*	**f_{A1} =8 операций**
*f←a*d + b*c*	**f_{*} =4 операций**
Return (e, f)	**$f_{\pm}$ =2 операций**
End.	**$f_{\leftarrow}$ =2 операций**

2. Алгоритм A2 (вычисление e, f за три умножения)

MultComplex2 (a, b, c, d; e, f)

z1←c(a + b)*	
z2←b(d + c)*	**f_{A2} =13 операций**
z3←a(d - c)*	**f_{*} =3 операций**
e←z1 - z2	**$f_{\pm}$ =5 операций**
f←z1 + z3	**$f_{\leftarrow}$ =5 операций**
Return (e, f)	
End.	

Пооперационный анализ этих двух алгоритмов не представляет труда, и его результаты приведены справа от записи соответствующих алгоритмов.

По совокупному количеству элементарных операций алгоритм *A2* уступает алгоритму *A1*, однако в реальных компьютерах операция умножения требует большего времени, чем операция сложения, и можно путем пооперационного анализа ответить на вопрос: при каких условиях алгоритм *A2* предпочтительнее алгоритма *A1*?

Введем параметры q и r, устанавливающие соотношения между временами выполнения операции умножения, сложения и присваивания для операндов действительного типа.

Тогда мы можем привести временные оценки двух алгоритмов к времени выполнения операции сложения/вычитания – t_+:

$t_* = q*t_+,\ q>1;$

$t_{\leftarrow}=r*t_+,\ r<1$, тогда приведенные к t_+ временные оценки имеют вид:

$T_{A1} = 4*q*t_+ + 2*t_+ + 2*r*t_+ = t_+*(4*q+2+2*r);$

$T_{A2} = 3*q*t_+ + 5*t_+ + 5*r*t_+ = t_+*(3*q+5+5*r).$

Равенство времен будет достигнуто при условии:

$4*q+2+2*r = 3*q+5+5*r$, откуда:

$q = 3 + 3r$ и следовательно при $q > 3 + 3r$ алгоритм *A2* будет работать более эффективно.

Таким образом, если среда реализации алгоритмов *A1* и *A2* – язык программирования, обслуживающий его компилятор и компьютер на котором реализуется задача – такова, что время выполнения операции умножения двух действительных чисел более чем втрое превышает время сложения двух действительных чисел, в предположении, что $r << 1$, а это реальное соотношение, то для реализации более предпочтителен алгоритм *A2*.

Конечно, выигрыш во времени пренебрежимо мал, если мы перемножаем только два комплексных числа, однако, если этот алгоритм является частью сложной вычислительной задачи с комплексными числами, требующей существенно значимого по времени количества умножений, то выигрыш во времени может быть ощутим. Оценка такого выигрыша на одно умножение комплексных чисел следует из только что проведенного анализа:

$$\Delta T = (q - 3 - 3*r)* t_{+}$$

5.5 Вопросы для самоконтроля

1) Элементарные операции в псевдоязыке высокого уровня;
2) Анализ трудоемкости основных алгоритмических конструкций;
3) Построение функции трудоемкости для суммирования матрицы;
4) Построение функции трудоемкости для задачи поиска максимума ;
5) Проблемы при переходе от трудоемкости к временным оценкам;
6) Методики перехода от функции трудоемкости к временным оценкам;
7) Возможности пооперационного анализа алгоритмов на примере задачи умножения комплексных чисел;

6. ТЕОРИЯ СЛОЖНОСТИ ВЫЧИСЛЕНИЙ И СЛОЖНОСТНЫЕ КЛАССЫ ЗАДАЧ

6.1 Теоретический предел трудоемкости задачи

Рассматривая некоторую алгоритмически разрешимую задачу, и анализируя один из алгоритмов ее решения, мы можем получить оценку трудоемкости этого алгоритма в худшем случае – $f\hat{}_A(D_A)=O(g(D_A))$. Такие же оценки мы можем получить и для других известных алгоритмов решения данной задачи. Рассматривая задачу с этой точки зрения, возникает резонный вопрос – а существует ли функциональный нижний предел для $g(D_A)$ и если «да», то существует ли алгоритм, решающий задачу с такой трудоемкостью в худшем случае.

Другая, более точная формулировка, имеет следующий вид: какова оценка сложности самого «быстрого» алгоритма решения данной задачи в худшем случае? Очевидно, что это оценка самой задачи, а не какого либо алгоритма ее решения. Таким образом, мы приходим к определению понятия функционального теоретического нижнего предела трудоемкости задачи в худшем случае:

$$F_{thlim} = min \{ \Theta (F_a\hat{} (D)) \}$$

Если мы можем на основе теоретических рассуждений доказать существование и получить оценивающую функцию, то мы можем утверждать, что любой алгоритм, решающий данную задачу работает не быстрее, чем с оценкой F_{thlim} в худшем случае:

$$F_a\hat{} (D) = \Omega (F_{thlim})$$

Приведем ряд примеров:

1) Задача поиска максимума в массиве $A=(a_1,\ldots,a_n)$ – для этой задачи, очевидно должны быть просмотрены все элементы, и $F_{thlim}=\Theta (n)$.

2) Задача умножения матриц - для этой задачи можно сделать предположение, что необходимо выполнить некоторые арифметические операции со всеми исходными данными, теоретическое обоснование какой–либо другой

оценки на сегодня не известно [6], что приводит нас к оценке $F_{thlim}=\Theta(n^2)$. Отметим, что лучший алгоритм умножения матриц имеет оценку $\Theta(n^{2,34})$ [6]. Расхождение между теоретическим пределом и оценкой лучшего известного алгоритма позволяет предположить, что либо существует, но еще не найден более быстрый алгоритм умножения матриц, либо оценка $\Theta(n^{2,34})$ должна быть доказана, как теоретический предел.

6.2 Сложностные классы задач

В начале 1960-х годов, в связи с началом широкого использования вычислительной техники для решения практических задач, возник вопрос о границах практической применимости данного алгоритма решения задачи в смысле ограничений на ее размерность. Какие задачи могут быть решены на ЭВМ за реальное время?

Ответ на этот вопрос был дан в работах Кобмена (Alan Cobham, 1964), и Эдмнодса (Jack Edmonds, 1965), где были введены сложностные классы задач.

1) Класс P (задачи с полиномиальной сложностью)

Задача называется полиномиальной, т.е. относится к классу P, если существует константа k и алгоритм, решающий задачу с $F_a(n)=O(n^k)$, где n - длина входа алгоритма в битах $n = |D|$ [6].

Задачи класса P – это интуитивно, задачи, решаемые за реальное время. Отметим следующие преимущества алгоритмов из этого класса:

- для большинства задач из класса P константа k меньше 6;
- класс P инвариантен по модели вычислений (для широкого класса моделей);
- класс P обладает свойством естественной замкнутости (сумма или произведение полиномов есть полином).

Таким образом, задачи класса P есть уточнение определения «практически разрешимой» задачи.

2) Класс NP (полиномиально проверяемые задачи)

Представим себе, что некоторый алгоритм получает решение некоторой задачи – соответствует ли полученный ответ поставленной задаче, и насколько быстро мы можем проверить его правильность?

Рассмотрим, например задачу о сумме:

Давно N чисел – $A = (a_1, \ldots a_n)$ и число V.

Задача: Найти вектор (массив) $X=(x_1, \ldots, x_n)$, $x_i \in \{0,1\}$, такой, что $\sum a_i x_i = V$.

Содержательно: может ли быть представлено число V в виде суммы каких либо элементов массива A.

Если какой-то алгоритм выдает результат – массив X, то проверка правильности этого результата может быть выполнена с полиномиальной сложностью: проверка $\sum a_i x_i = V$ требует не более $\Theta(N)$ операций.

Формально: $\forall D \in D_A$, $|D|=n$ поставим в соответствие сертификат $S \in S_A$, такой что $|S|=O(n^l)$ и алгоритм $A_s = A_s(D,S)$, такой, что он выдает «1», если решение правильно, и «0», если решение неверно. Тогда задача принадлежит классу NP, если $F(A_s)=O(n^m)$ [6].

Содержательно задача относится к классу NP, если ее решение некоторым алгоритмом может быть быстро (полиномиально) проверено.

6.3 Проблема $P = NP$

После введения в теорию алгоритмов понятий сложностных классов Эдмондсом (Edmonds, 1965) была поставлена основная проблема теории сложности – $P = NP$? Словесная формулировка проблемы имеет вид: можно ли все задачи, решение которых проверяется с полиномиальной сложностью, решить за полиномиальное время ? [6]

Очевидно, что любая задача, принадлежащая классу P, принадлежит и классу NP, т.к. она может быть полиномиально проверена – задача проверки решения может состоять просто в повторном решении задачи.

На сегодня отсутствуют теоретические доказательства как совпадения этих классов (*P=NP*), так и их несовпадения. Предположение состоит в том, что класс P является собственным подмножеством класса *NP*, т.е. *NP \ P* не пусто – рис 6.1

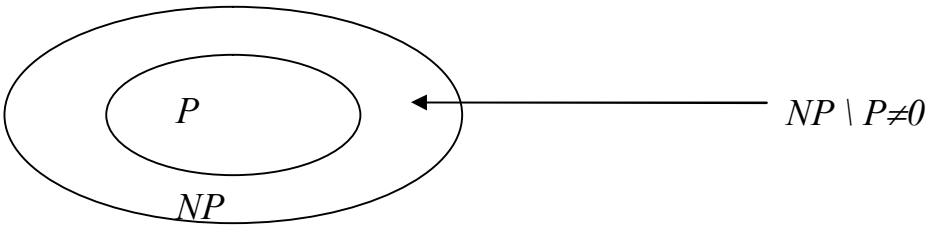


Рис 6.1 Соотношение классов P и NP

6.4 Класс *NPC* (*NP* – полные задачи)

Понятие *NP* – полноты было введено независимо Куком (Stephen Cook, 1971) и Левиным (журнал «Проблемы передачи информации», 1973,т.9, вып. 3) и основывается на понятии сводимости одной задачи к другой [6].

Сводимость может быть представлена следующим образом: если мы имеем задачу 1 и решающий эту задачу алгоритм, выдающий правильный ответ для всех конкретных проблем, составляющих задачу, а для задачи 2 алгоритм решения неизвестен, то если мы можем переформулировать (свести) задачу 2 в терминах задачи 1, то мы решаем задачу 2.

Таким образом, если задача 1 задана множеством конкретных проблем D_{A1}, а задача 2 – множеством, и существует функция f_s (алгоритм), сводящая конкретную постановку задачи 2 (d_{A2}) к конкретной постановке задачи 1(d_{A1}): $f_s(d_{(2)} \in D_{A2}) = d_{(1)} \in D_{A1}$, то задача 2 сводима к задаче 1.

Если при этом $F_A(f_s) = O(n^k)$, т.е. алгоритм сведения принадлежит классу P, то говорят, что *задача 1 полиномиально сводится к задаче 2* [6].

Принято говорить, что задача задается некоторым языком, тогда если задача 1 задана языком *L1*, а задача 2 – языком *L2*, то полиномиальная сводимость обозначается следующим образом: $L2 \leq_p L1$.

Определение класса *NPC* (*NP*-complete) или класса *NP*-полных задач требует выполнения следующих двух условий: во-первых, задача должна принадлежать классу *NP (L ∈ NP),* и, во-вторых, к ней полиномиально должны сводиться все задачи из класса *NP* ($L_x \le_P L$, для каждого $L_x \in NP$), что схематично представлено на рис 6.2.

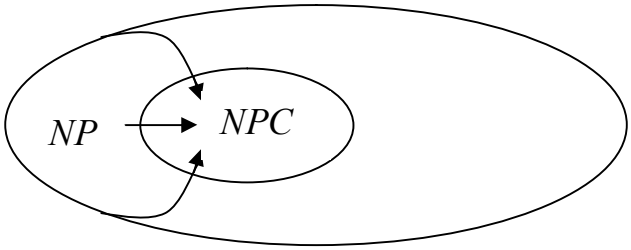


Рис 6.2 Сводимость и класс и NPC

Для класса *NPC* доказана следующая теорема: Если существует задача, принадлежащая классу *NPC*, для которой существует полиномиальный алгоритм решения ($F = O(n^k)$), то класс *P* совпадает с классом *NP*, т.е. *P=NP* [6].

Схема доказательства состоит в сведении любой задачи из NP к данной задаче из класса *NPC* с полиномиальной трудоемкостью и решении этой задачи за полиномиальное время (по условию теоремы).

В настоящее время доказано существование сотен *NP*– полных задач [6,7], но ни для одной из них пока не удалось найти полиномиального алгоритма решения. В настоящее время исследователи предполагают следующее соотношение классов, показанное на рис 6.3 – $P \neq NP$, то есть $NP \setminus P \neq 0$, и задачи из класса *NPC* не могут быть решены (сегодня) с полиномиальной трудоемкостью.

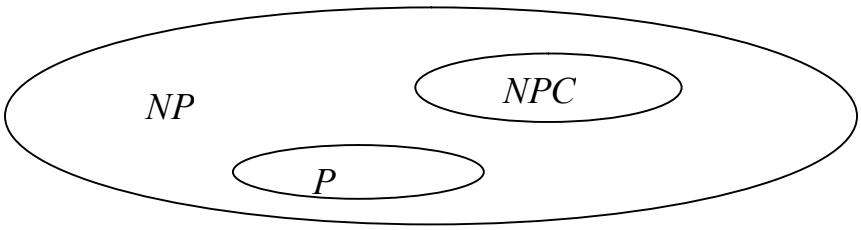


Рис 6.3 Соотношение классов P, NP, NPC

6.5 Примеры *NP* – полных задач

6.5.1 Задача о выполнимости схемы

Рассмотрим схему из функциональных элементов «и», «или», «не» с n битовыми входами и одним выходом, состоящую не более, чем из $O(n^k)$ элементов – рис 6.4

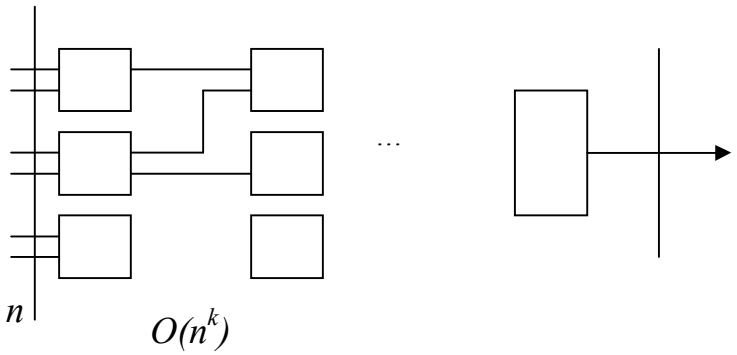


Рис 6.4 Абстрактная функциональная схема

Будем понимать под выполняющим набором значений из множества *{0,1}* на входе схемы, такой набор входов – значения $x_1,\ldots,x_n$, при котором на выходе схемы будет значение «1».

Формулировка задачи – существует ли для данной схемы выполняющий набор значений входа. Очевидно, что задача принадлежит классу *NP* – проверка предъявленного выполняющего набора не сложнее количества функциональных элементов, и следовательно не больше чем $O(n^k)$.

Это была одна из первых задач, для которой была доказана ее *NP* полнота, т.е. любая задача из класса *NP* полиномиально сводима к задаче о выполнимости схемы [6].

Решение этой задачи может быть получено перебором всех 2^n возможных значений входа с последующей проверкой на соответствие условию выполняющего набора. В худшем случае придется проверить все возможные значения входа, что приводит к оценке $F\hat{}(n) = O(n^k * 2^n)$. Для этой, как и для всех других *NP*–полных задач не известен полиномиальный алгоритм решения.

6.5.2 Задача о сумме

Уже рассмотренная задача о сумме также является NP–полной, отметим, что если количество слагаемых фиксировано, то сложность задачи является полиномиальной, так как:

- для 2-х слагаемых $\Rightarrow C_N^2=(N*(N-1))/(1*2) = O(N^2);$
- для 3-х слагаемых $\Rightarrow C_N^3=(N*(N-1)*(N-2))/(1*2*3) = O(N^3).$

Однако в общем случае придется перебирать 2^N различных вариантов, так как по биномиальной теореме $(a+b)^N = \sum c_N^k * a^{N-k} * b^k$, а при $a=b=1,$ имеем:

$$(1+1)^N = \sum C_N^k = 2^N, \text{ следовательно, } F_A(N, V) = O(N * 2^N).$$

6.5.3 Задача о клике

Пусть дан граф $G = G(V,E)$, где V – множество из n вершин, а E – множество ребер. Будем понимать под кликой максимальный по количеству вершин полный подграф в графе в G.

Задача состоит в определении клики в заданном графе G

Поскольку в полном графе на m вершинах имеется $m(m-1)/2$ ребер, то проверка, является ли данный граф полным, имеет сложность $O(m^2)$. Очевидно, что если мы рассматриваем подграф с m вершинами в графе G с вершинами $(m < n),$ то всего существует C_n^m различных подграфов. Если в задаче о клике количество вершин клики фиксировано, то перебирающий алгоритм имеет полиномиальную сложность:

$$F(m, n) = O(m^2 * C_n^m) = O(m^2 * n^m).$$

Однако в общем случае придется проверять все подграфы с количеством вершин $m = (2, n)$ на их полноту и определить максимальное значения m для которого в данном графе G существует полный подграф, что приводит к оценке в худшем случае:

$$\hat{F}(n) = \sum_k O(k^2 * C_n^k) \Rightarrow O(n^2 * 2^n)$$

6.6 Вопросы для самоконтроля

1) Теоретический предел трудоемкости задачи;

2) Основные задачи теории сложности вычислений, понятие реально разрешимых задач;
3) Понятие сложностных классов задач, класс P;
4) Сложностной класс NP, понятие сертификата;
5) Проблема P=NP, и ее современное состояние;
6) Сводимость языков и определение класса NPC;
7) Примеры NP – полных задач;
8) Задача о клике и ее особенности;

7. ПРИМЕР ПОЛНОГО АНАЛИЗА АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ О СУММЕ

7.1 Формулировка задачи и асимптотическая оценка

Словесно задача о сумме формулируется как задача нахождения таких чисел из данной совокупности, которые в сумме дают заданное число, классически задача формулируется в терминах целых чисел [6].

В терминах структур данных языка высокого уровня задача формулируется, как задача определения таких элементов исходного массива *S* из *N* чисел, которые в сумме дают число *V* (отметим, что задача относится к классу *NPC*).

Детальная формулировка:

Дано: *Массив S[i], i={1, N} и число V.*

Требуется: определить такие S_j, *что* $\sum S_j = V$

Тривиальное решение определяется равенством $V=Sum$, где $Sum=\sum S_i$, условия существования решения имеют вид:

$$Min\ \{S[i],\ i=1,N\} \le V \le Sum.$$

Получим асимптотическую оценку сложности решения данной задачи для алгоритма, использующего прямой перебор всех возможных вариантов. Поскольку исходный массив содержит *N* чисел, то проверке на равенство *V* подлежат следующие варианты решений:

- *V* содержит 1 слагаемое $\Rightarrow C_N^1 = N$ вариантов;
- *V* содержит 2 слагаемых $\Rightarrow C_N^2 = (N*(N-1))/(1*2)$ вариантов;
- *V* содержит 3 слагаемых $\Rightarrow C_N^3=(N*(N-1)*(N-2))/(1*2*3)$ вариантов;
- и т.д. до проверки одного варианта с N слагаемыми.

Поскольку сумма биномиальных коэффициентов для степени *N* равна - $(1+1)^N = \sum C_N^k = 2^N$ и для каждого варианта необходимо выполнить суммирование (с оценкой *O(N)*) для проверки на *V*, то оценка сложности алгоритма в худшем случае имеет вид:

$$\hat{F}_A(N, V) = O(N*2^N) \qquad (7.1)$$

7.2 Алгоритм точного решения задачи о сумме (метод перебора)

Определим вспомогательный массив, хранящий текущее сочетание исходных чисел в массиве *S*, подлежащих проверке на *V* – массив *Cnt[j]*, элемент массива равен «0», если число *S[j]* не входит в *V* и равен «1», если число *S[j]* входит в *V*

Решение получено, если $V = \sum S[j]*Cnt[j]$.

Могут быть предложены следующие две реализации механизма полного перебора вариантов:

- перебор по всевозможным сочетаниям из *k* элементов по *N,* т.е. сначала алгоритм пытается представить *V* как один из элементов массива *S*, затем перебираются все возможные пары, затем все возможные тройки и т.д.;
- перебор по двоичному счётчику, реализованному в массиве *Cnt*:

Вторая идея алгоритмически более проста и сводится к решению задаче об увеличении двоичного счётчика в массиве *Cnt* на «1»:

- при 00...0111 увеличение на «1» приводит к сбросу всех правых «1» и установке в «1» следующего самого правого «0»;
- при 00...1000, когда последний элемент счетчика равен «0» увеличение на «1» приводит к переустановке последнего элемента в массиве *Cnt* с «0» в «1».

Рассматривая массив *Cnt* как указатель на элементы массива *S*, подлежащие суммированию в данный момент, мы производим суммирование и проверку на *V*, до тех пор, пока решение не будет найдено или же безрезультатно будут просмотрены все возможные варианты.

Таким образом, алгоритм точного решения задаче о сумме методом прямого перебора имеет в формальной системе языка высокого уровня следующий вид:

```
TASKSUM(S,N,V; CNT,FL)
FL ← false
i ← 1
repeat
        Cnt[i] ← 0
        i ← i+1
Until i > N
Cnt[N] ← 1
Repeat
 Sum ← 0
 i ← 1
 Repeat
  Sum ← Sum + S[i] * Cnt[i]
  i ← i+1
 Until i > N
if Sum = V
        FL ← true
Return (Cnt,FL)
j ← N
While Cnt[j] = 1
                Cnt[j] = 0
                j ← j-1
 Cnt[j] ← 1
Until Cnt[0] = 1
Return(Cnt,FL)
```

7.3 Анализ алгоритма точного решения задачи о сумме

Рассмотрим лучший и худший случай для данного алгоритма:

а) В лучшем случае, когда последний элемент массива совпадает со значением V: $V=S[N]$, необходимо выполнить только одно суммирование, что приводит к оценке: $F_a\check{}(N)=\Theta(N)$;

б) В худшем случае, если решения вообще нет, то придется проверить все варианты, и $F_a\hat{}(N) = \Theta (N*2^N)$.

Получим детальную оценку для худшего случая, используя принятую методику подсчета элементарных операций:

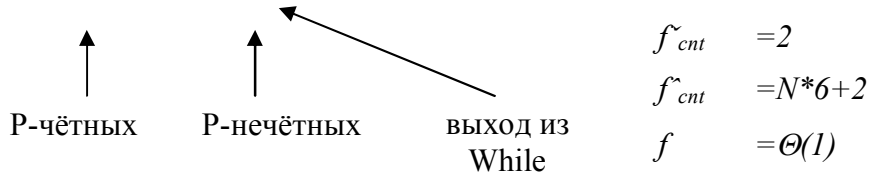
$$F_a\hat{}(N) = 2+N*(3+2)+2+(2^N-1)*\{2+N*(3+5)+1+1+f_{cnt}+2+2\} \quad (7.2)$$

Для получения значения f_{cnt} - количества операций, необходимых для увеличения счетчика на «1» рассмотрим по шагам проходы цикла *While*, в котором выполняется эта операция:

CNT	Количество проходов в While	Операций
001	1	6+2
010	0	2
011	2	2*6+2
100	0	2
101	1	6+2
110	0	2
111	3	3*6+2

Таким образом:

$f_{cnt} = (1/2)*(2)+(1/2)*(2)+(1/2)*((1/2)*1*6+(1/4)*2*6+(1/8)*3*6+...) =$



$=2 + 1/2 * 6 * (1/2^1+2/2^2+3/2^3+...) = 2+ 3 * (\sum_{k=1}^{\infty} k/2^k);$

Так как $\sum k*x^k = x/(1-x)^2$, [6]

то $\sum k*(1/2)^k = (1/2)/(1-(1/2))^2 = 2$, и, следовательно:

$f_{Cnt} = 8$ (! и не зависит от длины счетчика)

Подстановка f_{Cnt} в *(7.2)* дает:

$F^{\wedge}_A(N) = 4+5*N+(2^N-1)*(8*N+16),$ и окончательно:

$F^{\wedge}_A(N) =8*N*2^N+16*2^N-3*N-12,$

что согласуется с асимптотической оценкой – формула (1).

7.4 Вопросы для самоконтроля

1) Формулировка задачи о сумме;
2) Асимптотическая оценка сложности алгоритма для прямого перебора;
3) Алгоритм решения задачи о сумме;
4) Подалгоритм увеличения на единицу двоичного счетчика;
5) Оценки трудоемкости для лучшего и худшего случая;
6) Функция трудоемкости алгоритма для решения задачи о сумме в худшем случае;

8. РЕКУРСИВНЫЕ ФУНКЦИИ И АЛГОРИТМЫ

8.1 Рекурсивные функции

а) Терминологическое введение

По сути один и тот же метод, применительно к различным областям носит различные названия – это индукция, рекурсия и рекуррентные соотношения – различия касаются особенностей использования.

Под *индукцией* понимается метод доказательства утверждений, который строится на базе индукции при $n=0,1$, затем утверждение полагается правильным при $n=n$ b и проводится доказательство для $n+1$.

Под *рекурсией* понимается метод определения функции через её предыдущие и ранее определенные значения, а так же способ организации вычислений, при котором функция вызывает сама себя с другим аргументом.

Термин *рекуррентные соотношения* связан с американским научным стилем и определяет математическое задание функции с помощью рекурсии.

Основной задачей исследования рекурсивно заданных функций является получение $f(n)$ в явной или как еще говорят «замкнутой» форме, т.е. в виде аналитически заданной функции. В связи с этим рассмотрим ряд примеров:

б) Примеры рекурсивного задания функций

1. $$\begin{cases} f(0)=0 \\ f(n)= f(n-1)+1 \end{cases}$$

Здесь нетрудно сообразить, что $f(n)=n$.

2. $$\begin{cases} f(0)=1 \\ f(n)= n*f(n-1) \end{cases}$$

Последовательная подстановка дает – $f(n)=1*2*3*...*n = n!$

Для полноты сведений приведем формулу Стирлинга для приближенного вычисления факториала для больших n:

$$n! \approx (2\pi n)^{1/2} *(n^n)/(e^n)$$

3. $$\begin{cases} f(0)=1 \\ \\ \\ \end{cases}$$

$f(1)=1$

$f(n)= f(n-1)+ f(n-2), \quad n\geq 2$

Эта рекурсивная функция определяет числа Фибоначчи: 1 1 2 3 5 8 13, которые достаточно часто возникают при анализе различных задач, в том числе и при анализе алгоритмов. Отметим, что асимптотически $f(n)\approx [1,618^n]$ [9].

4. $$\begin{cases} f(0)=1 \\ f(n)= f(n-1)+ f(n-2)+\ldots+1 = \sum f(i)+1 \end{cases}$$

Для получения функции в явном виде рассмотрим ее последовательные значения: $f(0)=1,\ f(1)=2,\ f(2)=4,\ f(3)=8$, что позволяет предположить, что $f(n)=2^n$, точное доказательство выполняется по индукции.

5. $$\begin{cases} f(0)=1 \\ f(n)= 2*f(n-1) \end{cases}$$

Мы имеем дело с примером того, что одна и та же функция может иметь различные рекурсивные определения – $f(n)=2^n$, как и в примере 4, что проверяется элементарной подстановкой.

6. $$\begin{cases} f(0)=1 \\ f(1)=2 \\ f(n)= f(n-1)*f(n-2) \end{cases}$$

В этом случае мы можем получить решение в замкнутой форме, сопоставив значениям функции соответствующие степени двойки:

$$f(2) = 2 = 2^1$$

$$f(3) = 4 = 2^2$$

$$f(4) = 8 = 2^3$$

$$f(5) = 32 = 2^5$$

$$f(6) = 256 = 2^8$$

Обозначив через F_n - *n -ое* число Фибоначчи, имеем: $f(n)=2^{Fn}$.

8.2 Рекурсивная реализация алгоритмов

Большинство современных языков высокого уровня поддерживают механизм рекурсивного вызова, когда функция, как элемент структуры языка программирования, возвращающая вычисленное значение по своему имени, может вызывать сама себя с другим аргументом. Эта возможность позволяет напрямую реализовывать вычисление рекурсивно определенных функций. Отметим, что в силу тезиса Черча–Тьюринга аппарат рекурсивных функций Черча равномощен машине Тьюринга, и, следовательно, любой рекурсивный алгоритм может быть реализован итерационно.

Рассмотрим пример рекурсивной функции, вычисляющий факториал:

F(n);

If n=0 or n=1 (проверка возможности прямого вычисления)

Then

F ← 1

Else

*F ← n*F(n-1);* (рекурсивный вызов функции)

Return (F);

End;

Анализ трудоемкости рекурсивных реализаций алгоритмов, очевидно, связан как с количеством операций, выполняемых при одном вызове функции, так и с количеством таких вызовов. Графическое представление порождаемой данным алгоритмом цепочки рекурсивных вызовов называется деревом рекурсивных вызовов. Более детальное рассмотрение приводит к необходимости учета затрат как на организацию вызова функции и передачи параметров, так и на возврат вычисленных значений и передачу управления в точку вызова.

Можно заметить, что некоторая ветвь дерева рекурсивных вызовов обрывается при достижении такого значения передаваемого параметра, при котором функция может быть вычислена непосредственно. Таким образом, ре-

курсия эквивалентна конструкции цикла, в котором каждый проход есть выполнение рекурсивной функции с заданным параметром.

Рассмотрим пример для функции вычисления факториала (рис 8.1):

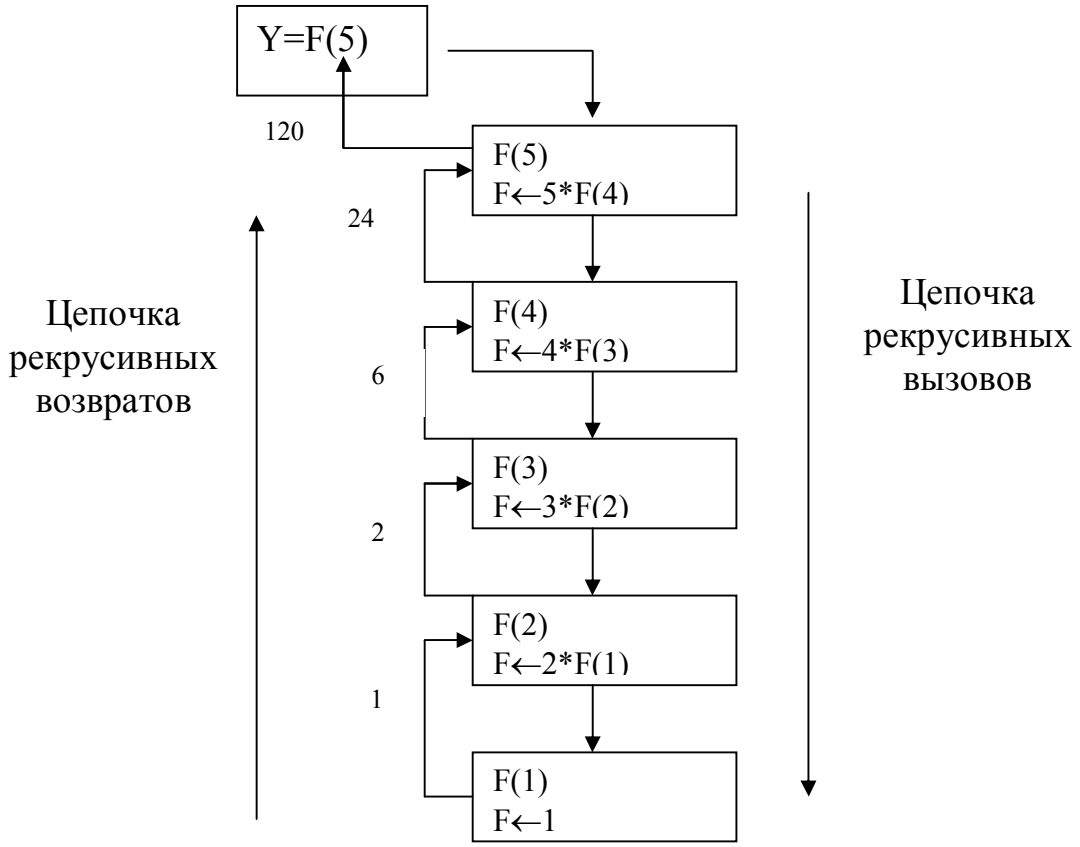


Рис 8.1 Дерево рекурсии при вычислении факториала – F(5)

Дерево рекурсивных вызовов может иметь и более сложную структуру, если на каждом вызове порождается несколько обращений – фрагмент дерева рекурсий для чисел Фибоначчи представлен на рис 8.2:

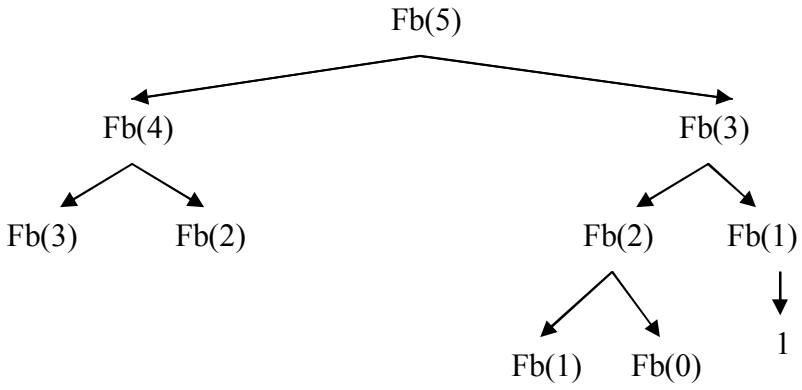


Рис 8.2 Фрагмент дерева рекурсии при вычислении чисел Фибоначчи – F(5)

8.3 Анализ трудоемкости механизма вызова процедуры

Механизм вызова функции или процедуры в языке высокого уровня существенно зависит от архитектуры компьютера и операционной системы. В рамках IBM PC совместимых компьютеров этот механизм реализован через программный стек. Как передаваемые в процедуру или функцию фактические параметры, так и возвращаемые из них значения помещаются в программный стек специальными командами процессора. Дополнительно сохраняются значения необходимых регистров и адрес возврата в вызывающую процедуру. Схематично этот механизм иллюстрирован на рис 8.3:

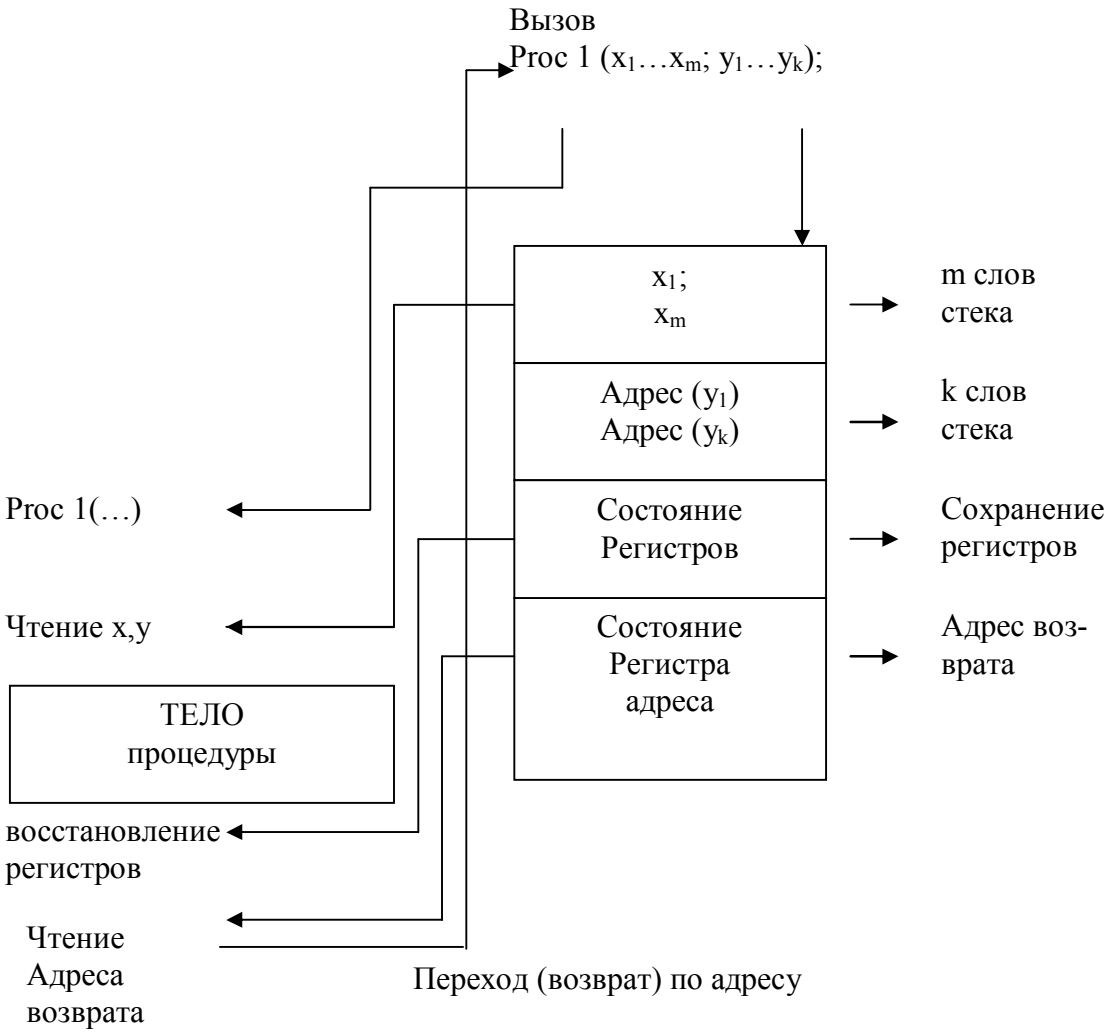


Рис 8.2 Механизм вызова процедуры с использованием программного стека

Для подсчета трудоемкости вызова будем считать операции помещения слова в стек и выталкивания из стека элементарными операциями в формальной системе. Тогда при вызове процедуры или функции в стек помещается адрес возврата, состояние необходимых регистров процессора, адреса возвращаемых значений и передаваемые параметры. После этого выполняется переход по адресу на вызываемую процедуру, которая извлекает переданные фактические параметры, выполняет вычисления, помещает их по указанным в стеке адресам, и при завершении работы восстанавливает регистры, выталкивает из стека адрес возврата и осуществляет переход по этому адресу.

Обозначив через:

m - количество передаваемых фактических параметров,

k - количество возвращаемых процедурой значений,

r - количество сохраняемых в стеке регистров, имеем:

$f_{вызова} = m+k+r+1+m+k+r+1 = 2*(m+k+r+1)$ элементарных операций на один вызов и возврат.

Анализ трудоемкости рекурсивных алгоритмов в части трудоемкости самого рекурсивного вызова можно выполнять разными способами в зависимости от того, как формируется итоговая сумма элементарных операций – рассмотрением в отдельности цепочки рекурсивных вызовов и возвратов, или совокупно по вершинам дерева рекурсивных вызовов.

8.4 Анализ трудоемкости алгоритма вычисления факториала

Для рассмотренного выше рекурсивного алгоритма вычисления факториала количество вершин рекурсивного дерева равно, очевидно, *n*, при этом передается и возвращается по одному значению *(m=1, k=1)*, в предположении о сохранении четырех регистров – *r=4*, а на последнем рекурсивном вызове значение функции вычисляется непосредственно – в итоге:

$$f_A(n)=n*2*(1+1+4+1)+(n-1)*(1+3)+1*2=18*n - 2$$

Отметим, что *n* – параметр алгоритма, а не количество слов на входе.

8.5 Вопросы для самоконтроля

1) Понятие индукции и рекурсии;
2) Примеры рекурсивного задания функций;
3) Рекурсивная реализация алгоритмов
4) Трудоемкость механизма вызова функции в языке высокого уровня;
5) Рекурсивное дерево, рекурсивные вызовы и возвраты;
6) Анализ трудоемкости рекурсивного алгоритма вычисления факториала;

9. РЕКУРСИВНЫЕ АЛГОРИТМЫ И МЕТОДЫ ИХ АНАЛИЗА

9.1 Логарифмические тождества

При анализе рекурсивных алгоритмов достаточно часто используются логарифмические тождества, далее предполагается, что, a > 0, b > 0, c > 0, основание логарифма не равно единице:

$b^{\log_b a} = a;$ $e^{\ln x} = x;$ $\log_b a^c = c * \log_b a;$ $\log_b a = 1/\log_a b$

$\log_b a = \log_c a / \log_c b \Rightarrow$ в записи $\Theta(\ln(x))$ основание логарифма не существенно, если он больше единицы, т.к. константа скрывается обозначением Θ.

$a^{\log_b c} = c^{\log_b a}$

Можно показать, что для любого $\xi > 0$ $\ln(n) = o(n^{\xi})$, при $n \to \infty$

9.2 Методы решения рекурсивных соотношений

В математике разработан ряд методов, с помощью которых можно получить явный вид рекурсивно заданной функции[2, 6] – метод индукции, формальные степенные ряды, метод итераций и т.д. Рассмотрим некоторые из них:

а) Метод индукции

Метод состоит в том, что бы сначала угадать решение, а затем доказать его правильность по индукции. Пример:

$$\begin{cases} f(0)=1 \\ f(n+1)=2*f(n) \end{cases}$$

Предположение: $f(n)=2^n$

Базис: если $f(n)=2^n$, *то* $f(0)=1$, что выполнено по определению.

Индукция: Пусть $f(n)=2^n$, тогда для $n+1 \Rightarrow f(n+1)= 2 * 2^n = 2^{n+1}$

Заметим, что базис существенно влияет на решение, так, например:

Если $f(0)=0$, то $f(n)=0$;

если $f(0)=1/7$, то $f(n)=(1/7)*2^n$; *если* $f(0)=1/64$, то $f(n)=(2)^{n-6}$

б) Метод итерации (подстановки)

Суть метода состоит в последовательной подстановке – итерации рекурсивного определения, с последующим выявлением общих закономерностей:

Пусть $f(n)=3*f(n/4)+n$, тогда:

$f(n)=n+3*f(n/4)=n+3*[\ n/4+3*f(n/16)\ ]=n+3*\ [n/4+3\{\ n/16+3*f(n/64)\ \}\ ]$, и раскрывая скобки, получаем:

$f(n)=n+3*n/4+9*n/16+27*n/64+\ldots+3^i*n/4^i$

Остановка рекурсии произойдет при $(n / 4^{\,i}) \le 1 \Rightarrow i \ge \log_4 n$, в этом случае последнее слагаемое не больше, чем $3^{\log_4 n}*\Theta(1) = n^{\log_4 3}*\Theta(1)$.

$f(n) = n*\sum (3/4)^{k} + n^{\log_4 3}*\Theta(1)$, т.к. $\sum(3/4)^{k} = 4*n$, то окончательно:

$$f(n) = 4 * n + n^{\log_4 3}*\Theta(1) = \Theta(n)$$

9.3 Рекурсивные алгоритмы.

Основной метод построения рекурсивных алгоритмов – это метод декомпозиции. Идея метода состоит в разделении задачи на части меньшей размерности, получение решение для полученных частей и объединение решений.

В общем виде, если происходит разделение задачи на b подзадач, которое приводит к необходимости решения a подзадач размерностью n/b, то общий вид функции трудоемкости имеет вид [6]:

$$f_A(n)= a *f_A(\ n/b\ )+d(n)+U(n) \quad (9.1)$$, где:

$d(n)$ – трудоемкость алгоритма деления задачи на подзадачи,

$U(n)$ – трудоемкость алгоритма объединения полученных решений.

Рассмотрим, например, известный алгоритм сортировки слиянием, принадлежащий Дж. Фон Нейману [6]:

На каждом рекурсивном вызове переданный массив делится пополам, что дает оценку для $d(n) =\Theta(1)$, далее рекурсивно вызываем сортировку полученных массивов половинной длины (до тех пор, пока длина массива не станет равной единице), и сливаем возвращенные отсортированные массивы за $\Theta(n)$.

Тогда ожидаемая трудоемкость на сортировку составит:

$$f_A(n)= 2 *f_A(\ n/2\ )+ \Theta(1)+ \Theta(n)$$

Тем самым возникает задача о получении оценки сложности функции трудоемкости, заданной в виде (9.1), для произвольных значений *a* и *b*.

9.4 Основная теорема о рекуррентных соотношениях

Следующая теорема принадлежит Дж. Бентли, Д. Хакен и Дж. Саксу (1980 г.), достаточно полное доказательство этой теоремы приведено в [6].

Теорема. Пусть $a \geq 1,\ b > 1$ - константы, $g(n)$ - функция, пусть далее:

$f(n)=a*f(n/b)+g(n)$, где $n/b = \lfloor (n/b) \rfloor$ или $\lceil (n/b) \rceil$

Тогда:

1) $Если\ g(n) = O(n^{\log_b a-\xi}),\ \xi>0,\ то\ f(n)=\Theta(n^{\log_b a})$

Пример: $f(n) = 8f(n/2)+n^2$, тогда $f(n) = \Theta(n^3)$

2) $Если\ g(n)=\Theta(n^{\log_б a}),\ то\ f(n)=\Theta(n^{\log_b a}*\log n)$

Пример: $f_A(n)= 2 * f_A(n/2)+ \Theta (n)$, тогда $f(n) = \Theta(n*\log n)$

3) $Если\ g(n) = \Omega(n^{\log_b a+e}),\ e > 0$, то $f(n) = \Theta(g(n))$

Пример: $f(n)=2*f(n/2)+n^2$, имеем:

$$n^{\log_b a} = n^1, \text{ и следовательно: } f(n) = \Theta(n^2)$$

Данная теорема является мощным средством анализа асимптотической сложности рекурсивных алгоритмов, к сожалению, она не дает возможности получить полный вид функции трудоемкости.

9.5 Вопросы для самоконтроля

1) Анализ рекурсивных соотношений методом итераций;
2) Анализ рекурсивных соотношений методом подстановки;
3) Общий вид функции трудоемкости для метода декомпозиции;
4) Основная теорема о рекуррентных соотношениях;
5) Примеры решения рекуррентных соотношений на основе теоремы Бентли, Хакен, Сакса;

10. ПРЯМОЙ АНАЛИЗ РЕКУРСИВНОГО ДЕРЕВА ВЫЗОВОВ

10.1 Алгоритм сортировки слиянием

Рассмотрим подход к получению функции трудоемкости рекурсивного алгоритма, основанный на непосредственном подсчете вершин дерева рекурсивных вызовов, на примере алгоритма сортировки слиянием.

Рекурсивная процедура *Merge Sort – MS* получает на вход массив *A* и два индекса *p* и *q*, указывающие на ту часть массива, которая будет сортироваться при данном вызове. Вспомогательные массивы *Bp* и *Bq* используются для слияния отсортированных частей массива.

MS(A, p ,q, Bp, Bq)

If $p \neq q$ (проверка останов рекурсии при $p=q$)

then

$r \leftarrow (p+q)$ *div 2*

MS(A, p, r, Bp,Bq) (рекурсивный вызов для первой части)

MS(A, r+1, q, Bp,Bq) (рекурсивный вызов для второй части)

Merge(A, p, r, q, Bp, Bq) (слияние отсортированных частей)

Return (A)

End

10.2 Слияние отсортированных частей (Merge)

Рассмотрим процедуру слияния отсортированных частей массива *A*, использующую дополнительные массивы *Bp* и *Bq*, в конец которых с целью остановки движения индекса помещается максимальное значение. Поскольку сам алгоритм рекурсивной сортировки устроен так, что объединяемые части массива *A* находятся рядом друг с другом, то алгоритм слияния вначале копирует отсортированные части в промежуточные массивы, а затем формирует объединенный массив непосредственно в массиве *A*.

Merge (A,p,r,q,Bp,Bq) *Количество операций в данной строке*

```
Max ← A[r]                                              2
      If Max <A[q] Then                                 2
            Max ← A[q]                                  ½*2
kp ← r - p + 1                                          3
p1 ← p – 1                                              2
For i ← 1 to kp     (копирование первой части)          1+ kp*3
      Bp [ i ] ← A[p1 + i ]                             kp*(4)
Bp[ kp+1] ← Max                                         3

kq ← q - r                                              2
For i ← 1 to kq     (копирование второй части)          1+ kq*3
      Bq [ i ] ←A[ r + i ]                              kq*(4)
Bq [ kq+ 1] ← Max                                       3
```

(заметим, что m=kp + kq = q – p + 1 – длина объединенного массива)

```
pp ← p                                                  1
pq ← r+1                                                2
For i ← p to q  (слияние частей)                        1+m*3
      If Bp [ pp ] < Bq [ pq ]                          m*3
        Then
            A[ i ] ← Bp[ pp ]                           ½*m*3
            pp ← pp +1                                  ½*m*2
        Else
            A [ i ] ← Bq [ pq ]                         ½*m*3
            pq ← pq +1                                  ½*m*2
Return(A)
End
```

На основании указанного количества операций можно получить трудоемкость процедуры слияния отсортированных массивов в среднем:

$$F_{merge}(m) = 2+2+1+3+2+1+kp*7+3+2+1+kq*7+3+1+2+1+m*(3+3+3+2)$$
$$= 11*m + 7*(kp+kq) + 23 = 18*m+23. \quad (10.1)$$

10.3 Подсчет вершин в дереве рекурсивных вызовов

Алгоритм, получая на входе массив из n элементов, делит его пополам при первом вызове, поэтому рассмотрим случай, когда $n=2^k$, $k = log_2 n$.

В этом случае мы имеем полное дерево рекурсивных вызовов глубиной k, содержащее n листьев, фрагмент дерева показан на рис 10.1.

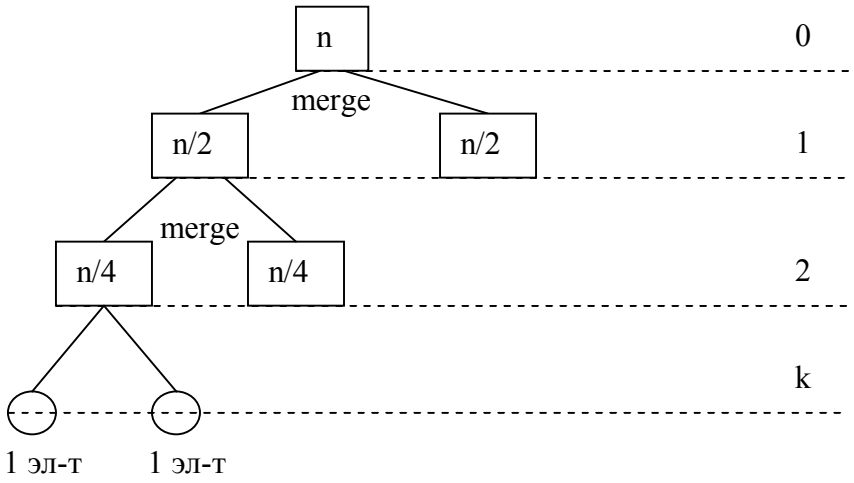


Рис 10.1 Фрагмент рекурсивного дерева при сортировке слиянием

Обозначим количество вершин дерева через V:

$$V = n + n/2+n/4+n/8+...+1 = n*(1+1/2+1/4+1/8+...+1)=2n - 1=2^{k+1} - 1$$

Из них все внутренние вершины порождают рекурсию, количество таких вершин – $V_r = n-1$, остальные n вершин – это вершины в которых рассматривается только один элемент массива, что приводит к останову рекурсии.

10.4 Анализ трудоемкости алгоритма сортировка слиянием

Таким образом, для n листьев дерева выполняется вызов процедуры *MS* с вычислением $r+1$, проверка условия $p=q$ и возврат в вызывающую процедуру для слияния, что в сумме с учетом трудоемкости вызова даёт:

$$F1(n) = n*2*(5+4+1) + n*2(If\ p=q\ u\ r+1) = 22*n;$$

Для n-1 рекурсивных вершин выполняется проверка длины переданного массива, вычисление середины массива, рекурсивный вызов процедур MS, и возврат, поскольку трудоемкость вызова считается при входе в процедуру, то мы получаем:

$$Fr(n) = (n-1)*2*(5+4+1) + (n-1)*(1+3+1) = 24*n - 24;$$

Процедура слияния отсортированных массивов будет вызвана *n-1* раз, при этом трудоемкость складывается из трудоемкости вызова и собственной трудоемкости процедуры *Merge*:

Трудоемкость вызова составит (для 6 параметров и 4-х регистров):

$$Fm_{вызов}(n) = (n-1)*2*(6+4+1) = 22*n - 22;$$

Поскольку трудоемкость процедуры слияния для массива длиной *m* составляет *18*m + 23 (10.1),* и процедура вызывается *n-1* раз с длинами массива равными *n, n/2, n/4, ...,* причем *2* раза с длиной *n/2, 4* раза с длиной *n/4*, то совокупно имеем:

$$Fm_{слияние}(n) = (n-1)*23 + 18*n + 2*18*(n/2) + 4*18*(n/4) + ... + =$$

$$= \{\text{учитывая, что таким образом обрабатывается } k\text{-}1 \text{ уровней}\}$$

$$= 18*n *(\log_2 n - 1) + 23*(n-1) = 18*n* \log_2 n + 5*n - 23;$$

Учитывая все компоненты функции трудоемкости, получаем окончательную оценку:

$$Fa(n) = F1(n) + Fr(n) + Fm_{вызов}(n) + Fm_{слияние}(n) =$$

$$= 22*n + 24*n - 24 + 22*n - 22 + 18*n* \log_2 n + 5*n - 23 =$$

$$= 18*n* \log_2 n + 73*n - 69 \quad (10.2)$$

Если количество чисел на входе алгоритма не равно степени двойки, то необходимо проводить более глубокий анализ, основанный на изучении поведения рекурсивного дерева, однако при любых ситуациях с данными оценка главного порядка $\Theta(n* \log_2 n)$ не измениться [6].

10.5 Вопросы для самоконтроля

1) Рекурсивный алгоритм сортировки слиянием

2) Процедура слияния двух отсортированных массивов

3) Оценка трудоемкости процедуры слияния;

4) Подсчет вершин в дереве рекурсивных вызовов для алгоритма сортировки слиянием;

5) Анализ алгоритма рекурсивной сортировки методом прямого подсчета вершин рекурсивного дерева;

11. ТЕОРИЯ И АЛГОРИТМЫ МОДУЛЯРНОЙ АРИФМЕТИКИ

11.1 Алгоритм возведения числа в целую степень

Задача о быстром возведении числа в целую степень, т.е. вычисление значения $y = x^n$ для целого n лежит в основе алгоритмического обеспечения многих криптосистем [11], отметим, что в этом аспекте применения вычисления производятся по mod_k. Представляет интерес детальный анализ известного быстрого алгоритма возведения в степень методом последовательного возведения в квадрат [6]. В целях этого анализа представляется целесообразным введение трех следующих специальных функций:

1. Функция $\beta(n)$

Функция определена для целого положительного n, и $\beta(n)$ есть количество битов в двоичном представлении числа n. Отметим, что функция $\beta(n)$ может быть представлена в виде: $\beta(n) = [\log_2(n)]+1=[\log_2(n+1)]$, где $[x]$ – целая часть x, $n > 0$.

2. Функция $\beta_1(n)$

Функция определена для целого положительного n, и $\beta_1(n)$ есть количество «1» в двоичном представлении числа n. Отметим, что функция $\beta_1(n)$ не является монотонно возрастающей функцией, например, для всех $n=2^k$ $\beta_1(n)=1$. График функции для начальных значений n представлен на рис 11.1.

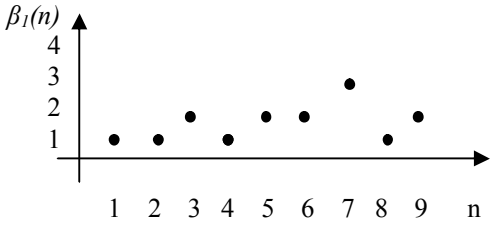


Рис 11.1 Значения функции для $n=1,2,...9$.

В силу определения $\beta_1(n)$ справедливо неравенство:

$$1 \le \beta_1(n) \le \beta(n) = [\log_2(n)]+1, \text{ т.е.} \beta_1(n) = O(\log_2(n))$$

Отметим, что функция $\beta_1(n)$ может быть рекурсивно задана следующим образом [5]:

$$\begin{cases} \beta_1(0)=0; \beta_1(1) = 1; \end{cases}$$

$\beta_1(2n) = \beta_1(n);$

$\beta_1(2n+1) = \beta_1(n) + 1;$

3. Функция $\beta_0(n)$

Функция определена для целого положительного *n,* и $\beta_0(n)$ есть количество «0» в двоичном представлении числа *n.* Отметим, что функция $\beta_0(n)$ не является монотонно возрастающей функцией, так для всех $n = 2^k - 1$ $\beta_0(n) = 0$

Для любого *n* справедливо соотношение $\beta(n) = \beta_0(n) + \beta_1(n)$.

Для дальнейшего анализа представляет так же интерес определение среднего значения функции $\beta_1(n)$ *для* $n = \{0,1,\ldots,N\}$, *где* $N = 2^k - 1$ (т.е. двоичное представление числа *N* занимает *k* разрядов), обозначим его через $\beta_s(N)$.

*Т*огда: $\beta_s(N) = \frac{1}{N+1}\sum_{m=0}^{N}\beta_s(m)$, поскольку количество чисел, имеющих *L* единиц в *K* разрядах равно количеству сочетаний из *L* по *K,* то, тогда:

$\sum_{m=0}^{N}\beta_s(m) = \sum_{L=1}^{k} L * C_K^L = \sum_{L=1}^{k} L * \frac{K}{L} C_{K-1}^{L-1} = K * \sum_{L=0}^{k-1} C_K^L = K * 2^{K-1}$, поскольку $N = 2^k - 1$, то:

$$\beta_s(N) = \frac{1}{N+1}\sum_{m=0}^{N}\beta_s(m) = \frac{K * 2^{k-1}}{2^K - 1 + 1} = \frac{K}{2} = \frac{\log_2(N+1)}{2} = \frac{\beta(N)}{2} \quad (11.1).$$

Идея быстрого алгоритма решения задачи о возведении в степень состоит в использовании двоичного разложения числа n и вычисления соответствующих степеней путем повторного возведения в квадрат [6]. Пусть, например, $n = 11$, тогда $x^{11} = x^8 * x^2 * x^1$, $x^4 = x^2 * x^2$ и $x^8 = x^4 * x^4$.

Алгоритмическая реализация идеи требует последовательного выделения битов, возведения *x* в квадрат и домножения *y* на те степени *x*, для которых в двоичном разложении *n* присутствует единица.

```
XstK (x,n;y);
z ← x;
y ← 1;
Repeat
      If (n mod 2) = 1
            then
                  y ← y*z;
      z ← z*z;
      n ← n div 2;
Until n = 0
Return (y)
End
```

Получим функцию трудоемкости данного алгоритма, используя введенные ранее обозначения и принятую методику счета элементарных операций в формальной системе процедурно-ориентированного языка высокого уровня:

$Fa(n) = 2 + \beta(n)*(2+2+2+1) + \beta_1(n)*(2) = 7*\beta(n) + 2*\beta1(n)+2$ *(11.2)*

Количество проходов цикла определяется количеством битов в двоичном представлении n – $\beta(n)$, а количество повторений операции $y \leftarrow y*z$ – количеством единиц в этом представлении – $\beta_1(n)$, что и отражает формула 11.2.

Определим трудоемкость алгоритма для особенных значений n, такими особенными значениями являются случаи, когда $n=2^k$ или $n=2^k - 1$:

- в случае если $n=2^k$, *то* $\beta_1(n)=1$ *и* $Fa(n)= 7*\beta(n) + 4;$

- в случае если $n=2^k -1$, *то* $\beta_1(n)= \beta(n)$ *и* $Fa(n)= 9*\beta(n) + 2$.

Если показатель степени заранее неизвестен, то можно получить среднюю оценку, в предположении, что представление числа n занимает не более k двоичных разрядов, т.е. $n < 2^k$ или $log_2 n < k$. Тогда по формуле *(11.1)*

$\beta_s(N) = \beta(N)/2$, *где* $N=2^k-1$, откуда:

$Fa(n) \leq 7*\beta(N) + 2*\beta_s(N)+2 = 8*\beta(N) + 2 = 8*([log_2(n)]+1)+2 = 8*k +2.$

Таким образом, количество операций, выполняемых быстрым алгоритмом возведения в степень, линейно зависит от количества битов в двоичном представлении показателя степени. Введение специальных функций $\beta_1(n)$ и $\beta(n)$ позволило получить точное значение функции трудоемкости анализируемого алгоритма.

11.2 Сведения из теории групп

Пусть A – не пустое множество и o - отображение (операция) определенное на множестве $A.$, пусть так же $e \in A$ – выделенный элемент множества A, называемый единицей, тогда, если выполнены следующие условия в теории групп определяют [8]:

1. Замкнутость

Отображение $o: A \times A \rightarrow A$, o называют групповой операцией.

Если $(a , b) \rightarrow c$, то обычно записывают $c = a \; o \; b;$

2. Ассоциативность

$\forall x,y,z, \in A$ *выполнено* $x \circ (y \circ x) = (x \circ y) \circ z;$

Если выполнены условия замкнутости и ассоциативности, то пара состоящая из множества A и операции $\circ$ – $\{ A , \circ \}$ называется *полугруппой*.

3. Существование единицы

$\forall a \in A$ *и* $e \in A$ *выполнено* $a \circ e = e \circ a = a;$

Если выполнены условия замкнутости, ассоциативности и существования единицы, то тройка состоящая из множества A, операции $\circ$, и элемента e – $\{ A , \circ , e \}$ называется *моноидом*.

4. Существование обратного элемента

$\forall x \in A , \exists y \in A : x \circ y = y \circ x = e;$

Если выполнены условия замкнутости, ассоциативности, существования единицы и существования обратного элемента, то тройка состоящая из множества A, операции $\circ$, и элемента e – $\{ A , \circ , e \}$ называется *группой*.

Приведем некоторые примеры:

а) Множество целых положительных чисел N_1 с обычной операцией сложения $\{N_1 , +\}$ образуют полугруппу, но не моноид, т.к. отсутствует единица группы.

б) Множество целых неотрицательных чисел N_0 с обычной операцией сложения и нулем $\{N_0 , +, 0\}$ образуют моноид, но не группу, т.к. в множестве N_0 отсутствуют обратные элементы.

в) Множество всех целых чисел (включая отрицательные) Z с обычной операцией сложения и нулем $\{Z , +, 0\}$ образуют группу, обратным элементом для данного является элемент, равный данному по модулю и имеющий противоположный знак.

11.3 Сведения из теории простых чисел

а) Сравнения

Говорят, что два числа a и b *сравнимы по модулю* c, если они дают при делении на c равные остатки.

Операция получения остатка от деления a на c записывается в виде:

$a \ mod_c = d$, что эквивалентно представлению: $a = k * c + d;$

Сравнимость двух чисел по модулю означает, что:

$a \bmod_c = b \bmod_c$ и записывается как $(a \equiv b) \bmod_c$

Примеры: $(13 \equiv 6) \bmod_7$, $(17 \equiv 22) \bmod_5$

Если, $a \bmod_c = 0$, то, число a делится на c без остатка: $(a \equiv 0) \bmod_c$

б) Простые числа

Число p называется простым, если оно не имеет других делителей, кроме единицы и самого себя. Очевидно, что в качестве возможных делителей есть смысл проверять только простые числа, меньшие или равные квадратному корню из проверяемого числа

Множество простых счетно, доказательство принадлежит Евклиду:

Пусть $p_1, \ldots, p_k$, - все простые числа, но тогда число

$a = (p_1 * p_2 * \ldots * p_k + 1)$ в остатке отделения на любое из них дает единицу

$a \bmod_{pi} = 1$ и следовательно является простым

в) Функция $\pi(n)$

Функция $\pi(n)$ в теории простых чисел обозначает количество простых чисел не превосходящих n.

Например $\pi(12)=5$, т.к. существует 5 простых чисел не превосходящих 12, а именно: 2,3,5,7,11.

Асимптотическое поведение функции $\pi(n)$ было получено в конце XIX века [6] и связано с функций интегрального логарифма:

Для больших n – $\pi(n) \approx li(n) \approx n / \ln n$

Полученный результат означает, что простые числа не так уж «редки», вероятность того, что среди взятых случайно $\ln n$ чисел, не превосходящих n, одно из них простое, достаточно велика. Отметим, что это используется при поиске больших простых чисел в вероятностном тесте Миллера–Рабина [6].

11.4 Вопросы для самоконтроля

1) Функции подсчета количества битов и количества единиц в двоичном представлении числа и их свойства;
2) Алгоритм быстрого возведения в степень
3) Анализ трудоемкости алгоритма быстрого возведения в степень;
4) Понятие полугруппы, моноида и группы, примеры групп;
5) Сравнения и сведения из теории простых чисел;

12. КРИПТОСИСТЕМА RSA И ТЕОРИЯ АЛГОРИТМОВ

12.1 Мультипликативная группа вычетов по модулю *n*

Рассмотрим некоторые группы, образованные на множестве вычетов по модулю *n*:

Пусть *n* – целое положительное число, тогда множество остатков от деления любого целого положительного числа на *n* называется множеством вычетов по модулю *n* и обозначается как Z_n:

$$Z_n = \{ 0, 1, 2, \ldots, n-1\}$$

Если в качестве групповой операции рассмотреть операцию сложения по модулю *n*: $(+mod_n)$, то множество Z_n образует с этой операцией и нулем в качестве «единицы» группу $\{ Z_n, +mod_n, 0 \}$, которую называют аддитивной группой вычетов по модулю *n*.

Обратным элементом для $a \in Z_n$ будет элемент $a^{-1} = (n - a)\ mod_n$

Если в качестве групповой операции рассмотреть операцию умножения по модулю *n*: $(*mod_n)$, то множество Z'_n образует с этой операцией и единицей группу $\{ Z'_n, *mod_n, 1 \}$, которую называют мультипликативной группой вычетов по модулю *n,* обозначаемую обычно как Z_n^*.

Обратный элемент в группе Z_n^* существует, только если *НОД (z, n) = 1*

Количество чисел, взаимопростых с *n*, и, следовательно, количество элементов в группе Z_n^* может быть получено по формуле Эйлера [6, 11]:

$| Z_n^* | = \varphi(n) = n*\Pi\ (1-1/p_i)$, где p_i – простые делители числа *n*.

Например: $\varphi(15) = 15*(1-1/3)(1-1/5) = 15*2/3*4/5=8$

Если число *n* – простое, т.е. $n = p$, то $\varphi(p) = p(1-1/p) = (p-1)$

Нахождение обратного элемента для некоторого элемента мультипликативной группы по умножению обычно выполняется с помощью расширенного алгоритма Евклида [6].

Заметим, что доказана теорема о единственности обратного элемента в группе Z_n^* [6], а *1* и *n - 1* являются обратными сами себе, т.к.:

$$1*1 = 1mod_n \text{ и } (n-1)*(n-1) = (n^2 - 2n + 1)mod_n = 1$$

Эти числа называются тривиальными корнями из единицы по модулю *n.*

Рассмотрим, например $Z_7^* = \{ 1, 2, 3, 4, 5, 6 \}$:

Обратным элементом к *2* будет *4*, т.е. $2^{-1} = 4 \ mod_7$, т.к. $2*4 = 8 \ mod_7 = 1$

Обратным элементом к 3 будет 5, т.е. $3^{-1} = 5 \ mod_7$, т.к. $3*5 = 15 \ mod_7 = 1$

12.2 Степени элементов в Z_n^* и поиск больших простых чисел

Поскольку групповая операция умножения по модулю ($*mod_n$) применима к любой паре чисел из Z_n^*, то мы можем определить степени элементов:

$$(a*a) \ mod_n = a^2; \qquad (a^2 * a) \ mod_n = a^3$$

Для степеней элементов в группе Z_p^*, справедлива малая теорема Ферма:

Если *p* – простое число, то для каждого элемента справедливо сравнение:

$$\forall a \in Z_p^* : a^{p-1} \equiv 1 \ mod_p$$

Например, для Z_7^* справедливо: $5^6 \equiv 4^6 \equiv 3^6 \equiv 2^6 \equiv 1 mod_7$

Обобщением малой теоремы Ферма для любого (не обязательно простого) *n* является теорема Эйлера (Ферма–Эйлера):

$$\forall a \in Z_n^* : a^{\varphi(n)} \equiv 1 \ mod_n$$

На теореме Ферма–Эйлера основан специальный алгоритм поиска больших простых чисел – вероятностный тест Миллера-Рабина [6].

Напомним, что количество простых чисел, не превосходящих *x* – функция $\pi(x)$ имеет следующую асимптотическую оценку: $\pi(x) \approx x/\ln x$. Это приводит к оценке $1/\ln n$ для вероятности того, что наугад (случайно) взятое число *n* является простым.

Идея вероятностного теста Миллера-Рабина состоит в следующем:

Генерируем случайное число *n* и выбираем некоторое $a \in \{2,....n-2\}$, тогда по теореме Ферма–Эйлера:

Если $(a^{n-1}) \ mod_n \neq 1$, то, очевидно, что число n – составное;

если $(a^{n-1}) \ mod_n = 1$, то, возможно необходимо проверить другое a;

Вероятность ошибки теста экспоненциально падает с ростом успешных проверок с различными значениями $a \in \{2,....n-2\}$, реально выполняется порядка нескольких десятков проверок [6].

12.3 Криптосистема RSA

Предложенная в 1977 году Риверстом, Шамилем и Адлеманом (R. Rivest, A. Shamir, L. Adleman) криптосистема с открытым ключом – RSA может быть кратко описана следующим образом [11]:

а) Находим два больших простых числа p и q (тест Миллера - Рабина)

б) Вычисляем $n = p*q;\ n \approx 2^{512} - 2^{768}$

в) По построению $\varphi(n) = p*q*(1-1/q)*(1-1/p) = (p-1)*(q-1)$

г) Выбираем число e, такое, что: $НОД\ (e, \varphi(n)) = 1$;

д) Находим число f обратное к e по модулю $\varphi(n)$ с помощью расширенного алгоритма Евклида:

$$f = e^{-1} \bmod_{\varphi(n)},\ m.e.(e*f \equiv 1) \bmod_{\varphi(n)},\text{или } e*f = k*\varphi(n)+1;$$

Шифрование:

Разбиваем сообщение на блоки M_i: $\beta(M_i) = \beta(n)-1$ и вычисляем:

$$C_i = M_i^e \bmod_n$$

Дешифрование:

По принятому сообщению C_i вычисляем (все операции по mod_n):

$$C_i^f \bmod_n = (M^e)^f = M^{e*f} = M^{k*\varphi(n)+1} = M*(M^{\varphi(n)})^k = M_i$$

12.4 Криптостойкость RSA и сложность алгоритмов факторизации

Поскольку значения e и n известны, то задача вскрытия криптосистемы сводится к вычислению f, такого, что $(e*f \equiv 1) \bmod_{\varphi(n)}$

На сегодня теоретически не доказано, что для определения f необходимо разложить n на множители, однако, если такой алгоритм будет найден, то на его основе можно построить быстрый алгоритм разложения числа на простые множители [11].

Поэтому криптостойкость RSA определяется сегодня алгоритмической сложностью задачи факторизации – задачи разложения числа на простые множители. Отметим, что за последние 20 лет алгоритмический прогресс в этой области значительно превышает рост производительности процессоров.

На сегодня в области трудно решаемых задач приятa следующая единица измерения временной сложности задачи – *1 MY*.

1 MY – это задача, для решения которой необходима работа компьютера, выполняющего 1 млн. операций в секунду в течение одного года.

В 1977 году Р. Риверст прогнозировал на основе лучшего в то время алгоритма решения задачи факторизации методом эллиптических кривых временную сложность факторизации составного числа длиной в 129 десятичных цифр (*129D*) – $n \approx 10^{129}$ в $4 * 10^{16}$ лет [11]. Однако этот модуль был разложен на множители за *5000 MY* (с использованием сети Интернет) в 1994 году алгоритмом, использующим метод квадратичного решета. Модуль RSA *140D* был факторизирован в 1999 году алгоритмом, использующим метод обобщенного числового решета за *2000 MY.* Более подробные сведения о временной сложности задачи факторизации и рад проектов по факторизации модулей RSA приведены в [11].

Наилучший сегодня алгоритм факторизации, использующий метод обобщенного числового решета имеет следующую временную оценку:

$$T(n) = O\left(e^{(\ln n)^{1/3} * (\ln \ln n)^{2/3}} \right);$$

В заключение отметим, что именно успехи асмптотического и экспериментального анализа алгоритмов позволяют не только прогнозировать временную сложность раскрытия криптосистемы RSA, обеспечивая тем самым ее криптостойкость, но и рассчитывать длину модуля (количество битов в двоичном представлении числа *n*) необходимую для эффективного шифрования с прогнозируемым временем раскрытия.

12.5 Вопросы для самоконтроля

1) Мультипликативная группа вычетов по модулю N и ее свойства;
2) Степени элементов и теорема Ферма-Эйлера;
3) Идея вероятностного теста Миллера-Рабина для поиска больших простых чисел;
4) Криптосистема RSA;
5) Применение теории алгоритмов к анализу криптостойкости RSA.

Э К З А М Е Н А Ц И О Н Н Ы Е В О П Р О С Ы

1) Исторические аспекты разработки теории алгоритмов;
2) Цели и задачи классической теории алгоритмов;
3) Цели и задачи теории асимптотического анализа алгоритмов;
4) Цели и задачи практического анализа алгоритмов;
5) Применение результатов теории алгоритмов;
6) Формализации алгоритма, определения Колмогорова и Маркова;
7) Требования к алгоритму, связанные с формальными определениями;
8) Понятие общей и конкретной проблемы по Посту;
9) Пространство символов и примитивные операции в машине Поста;
10) Понятие финитного 1-процесса в машине Поста;
11) Способы задания проблем и формулировка 1;
12) Гипотеза Поста;
13) Формальное описание машины Тьюринга;
14) Функция переходов в машине Тьюринга;
15) Понятие об алгоритмически неразрешимых проблемах
16) Проблема позиционирования в машине Поста;
17) Проблема соответствий Поста над алфавитом Σ;
18) Проблема останова в машине Тьюринга;
19) Проблема эквивалентности и тотальности;
20) Формальная система языка высокого уровня;
21) Понятие трудоемкости алгоритма в формальном базисе;
22) Обобщенный критерий оценки качества алгоритма,
23) Обозначения в анализе алгоритмов: худший, лучший и средний случаи;
24) Классификация алгоритмов по виду функции трудоемкости;
25) Примеры количественных и параметрически–зависимых алгоритмов;
26) Обозначения в асимптотическом анализе функций;

27) Примеры функций, не связанных асимптотическими обозначениями;
28) Элементарные операции в псевдоязыке высокого уровня;
29) Анализ трудоемкости основных алгоритмических конструкций;
30) Построение функции трудоемкости для задачи суммирования матрицы;
31) Построение функции трудоемкости для поиска максимума в массиве;
32) Проблемы при переходе от трудоемкости к временным оценкам;
33) Методики перехода от функции трудоемкости к временным оценкам;
34) Возможности пооперационного анализа алгоритмов на примере задачи умножения комплексных чисел;
35) Теоретический предел трудоемкости задачи;
36) Основные задачи теории сложности вычислений
37) Понятие сложностных классов задач, класс P;
38) Сложностной класс NP, понятие сертификата;
39) Проблема P=NP, и ее современное состояние;
40) Сводимость языков и определение класса NPC;
41) Примеры NP – полных задач;
42) Задача о клике и ее особенности;
43) Формулировка задачи о сумме;
44) Асимптотическая оценка сложности алгоритма для прямого перебора;
45) Алгоритм решения задачи о сумме;
46) Подалгоритм увеличения на единицу двоичного счетчика;
47) Оценки трудоемкости для лучшего и худшего случая;
48) Функция трудоемкости алгоритма для решения задачи о сумме;
49) Понятие индукции и рекурсии;
50) Примеры рекурсивного задания функций;
51) Рекурсивная реализация алгоритмов
52) Трудоемкость механизма вызова функции в языке высокого уровня;
53) Рекурсивное дерево, рекурсивные вызовы и возвраты;
54) Трудоемкость рекурсивного алгоритма вычисления факториала;

55) Анализ рекурсивных соотношений методом итераций;
56) Анализ рекурсивных соотношений методом подстановки;
57) Общий вид функции трудоемкости при решении задач методом декомпозиции;
58) Основная теорема о рекуррентных соотношениях;
59) Примеры решения рекуррентных соотношений на основе теоремы Бентли – Хакен – Сакса;
60) Рекурсивный алгоритм сортировки слиянием
61) Процедура слияния двух отсортированных массивов
62) Оценка трудоемкости процедуры слияния;
63) Подсчет вершин в дереве рекурсивных вызовов для алгоритма сортировки слиянием;
64) Анализ алгоритма рекурсивной сортировки методом прямого подсчета вершин рекурсивного дерева;
65) Функции подсчета количества битов и количества единиц в двоичном представлении числа и их свойства;
66) Алгоритм быстрого возведения в степень
67) Анализ трудоемкости алгоритма быстрого возведения в степень;
68) Понятие полугруппы, моноида и группы, примеры групп;
69) Сравнения и сведения из теории простых чисел;
70) Мультипликативная группа вычетов по модулю N и ее свойства;
71) Степени элементов группы и теорема Ферма-Эйлера;
72) Вероятностный тест Миллера-Рабина для поиска простых чисел;
73) Криптосистема RSA;
74) Применение теории алгоритмов к анализу криптостойкости RSA.

Л И Т Е Р А Т У Р А

1. Ахо А. Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Том 1 – Синтаксический анализ. – М.: Мир, 1978 г. – 612 с., ил.

2. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: – М.: Издательский дом «Вильямс», 2001 г. –384 с., ил.

3. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. – 2-ое изд., испр. – СПб.: Невский диалект, 2001 г. – 352 с., ил.

4. Карпов Ю.Г. Теория автоматов – СПб.: Питер, 2002 г. – 224с., ил.

5. Кнут Д. Искусство программирования. Тома 1, 2, 3. 3-е изд. Пер. с англ. : Уч. пос. – М.: Изд. дом "Вильямс", 2001 г.

6. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 2001 г. – 960 с., 263 ил.

7. Макконнел Дж. Анализ алгоритмов. Вводный курс. – М.: Техносфера, 2002 г. –304 с.

8. Новиков Ф. А. Дискретная математика для программистов. – СПб.: Питер, 2001 г. – 304 с., ил.

9. Романовский И.В. Дискретный анализ. Учебное пособие для студентов, специализирующихся по прикладной математике. – Издание 2-ое, исправленное. – СПб.; Невский диалект, 2000 г. – 240 с., ил.

10. Успенский В.А. Машина Поста. – М.: Наука, 1979 г. – 96 с. – (Популярные лекции по математике).

11. Чмора А.Л. Современная прикладная криптография. – М.: Гелиос АРВ, 2001 г. – 256 с., ил.

Учебное издание

Ульянов Михаил Васильевич, Шептунов Максим Валерьевич

Математическая логика и теория алгоритмов
Часть II
Теория алгоритмов
Учебное пособие

Подписано в печать 20.04.2003
Формат 60 x 80 1/16
Объем 5,5 п.л. Тираж 100 экз. Заказ № 128

Отпечатано в типографии Московской государственной академии
приборостроения и информатики
107846, Москва, ул. Стромынка, 20