МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.И. Фролов

ОСНОВЫ АНАЛИЗА АЛГОРИТМОВ

Рекомендовано учебно-методическим советом ОрелГТУ в качестве учебного пособия

Орел 2008

УДК 004.421(075):510.52(075)]001.8(075)
ББК 32.973.2я7
Ф91

Рецензенты:
Заведующий кафедрой «Информационные системы»
Орловского государственного технического университета
доктор технических наук, профессор
И.С. Константинов

Профессор кафедры систем автоматизированного проектирования и управления
Санкт-Петербургского государственного технологического института,
доктор технических наук, профессор
В.И. Халимон

Ф91 Фролов, А.И. **Основы анализа алгоритмов:** учебное пособие / А.И. Фролов. – Орел: ОрелГТУ, 2008. – 110 с.

В учебном пособии рассмотрены вопросы анализа трудоемкости и сложности итеративных и рекурсивных алгоритмов, основы теории сложности алгоритмов. Приводятся основные понятия и определения анализа, теоретические основы и практические приемы анализа алгоритмов по времени, основанные на выведении функции трудоемкости путем подсчета количества элементарных операций в записи алгоритма на императивном языке программирования или псевдокоде.

Предназначено студентам специальностей 230105 «Программное обеспечение вычислительной техники и автоматизированных систем», 230201 «Информационные системы и технологии» и направления подготовки бакалавров, 230100.62 «Информатика и вычислительная техника», изучающим дисциплину «Структуры и алгоритмы обработки данных», а также другие дисциплины, связанные с построением и анализом алгоритмов. Может быть использовано аспирантами и преподавателями технических вузов.

УДК 004.421(075):510.52(075)]001.8(075)
ББК 32.973.2я7

Содержание

Введение

В настоящее время компьютерная техника и соответствующее программное обеспечение применяются для решения все более и более требовательных по времени и памяти задач. Несмотря на то, что современный этап развития вычислительной техники характеризуется высокими темпами роста вычислительной мощности и объемов памяти компьютеров, существуют и появляются новые алгоритмические задачи, требующие тщательной проработки вопросов эффективности разрабатываемых алгоритмов. Такие задачи появляются при разработке серверных приложений, обрабатывающих запросы множества «клиентов», при организации сложных вычислений на персональных компьютерах, в процессах первичной обработки данных на промышленных микроконтроллерах АСУТП и т.д.

При подготовке специалистов и бакалавров по специальностям 230105 «Программное обеспечение вычислительной техники и автоматизированных систем», 230201 «Информационные системы и технологии» и направлению 230100.62 «Информатика и вычислительная техника», основной сферой деятельности которых является разработка программного обеспечения, значительное внимание уделяется их умению анализировать и обоснованно выбирать методы решения задач организации вычислений и обработки данных. Данные задачи обычно сводятся к анализу и выбору структур данных и алгоритмов. Поэтому рассмотрение вопросов анализа алгоритмов по времени и памяти является актуальным.

Наибольший вес в комплексной оценке эффективности алгоритма в большей части случаев имеет его трудоемкость. На этапе анализа возможных методов решения задачи имеет смысл оценка сложности алгоритма, то есть асимптотическая оценка его трудоемкости. При реализации алгоритма на конкретном языке программирования для оценки эффективности, а также в целях

выявления возможностей ее повышения, необходима уже более точная оценка в виде функции трудоемкости, или временная оценка.

Оценка сложности по памяти обычно является тривиальной задачей. Исключение составляют рекурсивные алгоритмы, где необходимо учитывать память, выделяемую для организации рекурсивных вызовов и, следовательно, определять их количество. Однако эта функция количества вызовов получается в ходе анализа сложности алгоритма по времени, и получение оценки по памяти заключается лишь в умножении правой части найденной зависимости на объем памяти, необходимой для реализации одного вызова. Поэтому в данном учебном пособии основное внимание уделено рассмотрению вопросов анализа алгоритмов по времени.

Практически во всех источниках, приведенных в списке литературы, в той или иной мере рассматриваются вопросы анализа, однако данный материал не систематизирован, не обобщен или не рассчитан по сложности на студентов указанных выше специальностей и направлений. Как правило, приводятся лишь примеры анализа трудоемкости или сложности для конкретных рассматриваемых алгоритмов.

В данном учебном пособии рассматриваются основные понятия и определения анализа алгоритмов, основы теории сложности алгоритмов, подробно рассматриваются теоретические основы и практические приемы анализа алгоритмов по времени, основанные на выведении функции трудоемкости путем подсчета количества элементарных операций в записи алгоритма на императивном языке программирования или псевдокоде. В разделах 2-4 приведены задачи, которые ориентированы в основном на проведение практических занятий и самостоятельную работу студентов по дисциплине «Структуры и алгоритмы обработки данных». Учебное пособие также может быть использовано при изучении других дисциплин, связанных с основами алгоритмизации, построением и анализом алгоритмов.

1 Основные понятия и определения анализа

При использовании алгоритмов для решения практических задач возникает проблема рационального выбора алгоритма решения задачи. То есть предполагается, что одну и ту же задачу можно решить, используя различные алгоритмы, отличающиеся количеством ресурсов, необходимых для их реализации. В качестве ресурсов рассматриваются машинное время и память, необходимые для решения задачи определенным алгоритмом.

В целях однозначной трактовки используемых ниже терминов введем следующие определения, относящиеся к теории алгоритмов.

Алгоритм – это заданное на некотором языке (в терминах некоторой формальной системы) конечное предписание, задающее конечную последовательность выполнимых элементарных операций для решения задачи.

Общая проблема – это постановка решаемой алгоритмом задачи в общем виде. Например: упорядочить по возрастанию массив целых чисел A[1..N].Конкретная проблема – это конкретная постановка задачи, которую будет решать алгоритм в процессе работы программы. Например: упорядочить по возрастанию массив целых чисел A={4, 2, 9, 14, 7}. Тогда можно говорить, что общая проблема состоит из множества (или является множеством) конкретных проблем (Э. Пост, 1936 г.).

Формальная система – конечное множество принятых по соглашению символов и конечное число точных правил оперирования этими символами, которые дают возможность образовать из символов некоторые комбинации. Примеры формальных систем: машина Тьюринга, машина Поста. Такие машины называются абстрактными. Абстрактная машина – это математическая формализация, которая моделирует правила выполнения программы (или алгоритма) на реальной вычислительной машине.

1.1 Сравнительные оценки алгоритмов

Решение проблемы выбора алгоритма связано с построением системы сравнительных оценок, которая опирается на его представление в некоторой формальной системе. При этом рассматриваются алгоритмы, удовлетворяющие следующим требованиям:

– алгоритм применим к общей проблеме, то есть может давать решения любой конкретной проблемы данного множества;

– алгоритм корректно решает поставленную задачу;

– алгоритм является финитным, то есть решает проблему за конечное время.

Примем следующие допущения:

– каждая команда выполняется не более чем за фиксированное время;

– исходные данные алгоритма представляются машинными словами по b битов каждое.

Конкретная проблема задается N словами памяти, таким образом, на входе алгоритма – $N_b = N \cdot b$ бит информации. Программа, реализующая алгоритм для решения общей проблемы, состоит из M машинных инструкций по b_M битов – $M_b = M \cdot b_M$ бит информации. Кроме того, алгоритм может требовать следующих дополнительных ресурсов абстрактной машины:

– S_d – объем памяти, необходимый для хранения промежуточных результатов;

– S_r – объем памяти, необходимый для организации вычислительного процесса (например, 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. Тогда трудоемкость алгоритма можно определить следующим образом.

Под трудоемкостью алгоритма $F_A(N)$ для данного конкретного входа N понимается количество «элементарных» операций, совершаемых алгоритмом для решения конкретной проблемы в данной формальной системе.

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

$$y_A = c_1 \cdot F_A(N) + c_2 \cdot N + c_3 \cdot M + c_4 \cdot S_d + c_5 \cdot S_r,$$

где c_i – веса ресурсов.

Во-первых, необходимо отметить, что для большинства алгоритмов в приведенной зависимости доминирующим параметром будет являться трудоемкость, а ее анализ зачастую является сложным и трудоемким процессом. Во-вторых, оценки потребности алгоритма в остальных ресурсах могут быть получены достаточно просто. Исключение составляет только определение объема памяти, необходимой для организации рекурсии. Однако рассмотренный в разделе анализа рекурсивных алгоритмов метод анализа дерева рекурсивных вызовов и возвратов позволяет определить количество таких вызовов, и задача также становится элементарной. Поэтому дальнейшее рассмотрение вопросов анализа алгоритмов будет посвящено теоретическим и практическим аспектам анализа трудоемкости алгоритмов.

Однако сразу отметим, что эффективность алгоритма по времени не единственный и не всегда главный критерий его оценки. Можно выделить несколько причин, в силу которых справедливо это утверждение:

1. Созданная программа будет использоваться только несколько раз. Тогда стоимость написания и отладки программы будет доминировать в общей стоимости программы, а фактическое время выполнения не окажет на нее существенного влияния. В этом

случае следует предпочесть алгоритм, наиболее простой для реализации.

2. Программа будет работать только с малыми входными наборами данных. В этом случае порядок скорости роста трудоемкости (точные определения используемым здесь терминам будут даны ниже) будет иметь меньшее значение, чем константа, присутствующая в функции трудоемкости. В этом случае также предпочтителен менее эффективный по времени, но более простой алгоритм.

3. Эффективные, но сложные алгоритмы могут быть нежелательными при сопровождении программ не той организацией или не теми лицами, которые их разработали.

4. Эффективный по времени алгоритм требует большого объема памяти.

5. В численных алгоритмах не менее важны точность и устойчивость.

1.2 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.

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

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

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

Тогда 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. Худший случай – соответствует конкретным проблемам размерности N, для решения которых алгоритму A потребуется наибольшее количество операций – $F_A^{\wedge}(N)$.

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

2. Лучший случай – соответствует конкретным проблемам размерности N, для решения которых алгоритму A потребуется наименьшее количество операций – $F_A^{\vee}(N)$.

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

3. Средний случай определяет` $\overline{F}_A(N)$ – среднее количество операций, необходимых алгоритму A для решения конкретных проблем размерностью N.

$$\overline{F}_A(N) = \frac{1}{M_{DN}} \cdot \sum_{D \in D_N} F_A(D) \text{ – средний случай на } D_N.$$

Данное выражение получено в предположении, что вероятности появления конкретных проблем из подмножества D_N равны, то есть все конкретные проблемы вносят одинаковый вклад в трудоемкость среднего случая. На практике такое предположение вводится достаточно часто вследствие невозможности или сложности точного определения вероятностей на этапе анализа.

1.3 Классификация алгоритмов по виду функции трудоемкости

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 входного набора данных 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 выполнения стандартных операций с массивами и матрицами – умножение матриц, обращение матриц и т.д.

2. Параметрически-зависимые по трудоемкости алгоритмы

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

$$F_A(D) = F_A(d_1,...,d_N) = F_A(P_1,...,P_m),\ m \le N$$

где d_i – значения входных параметров;

P_i – значения входных параметров, влияющих на функцию трудоемкости.

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.

Например:

– вычисление x^k последовательным умножением: $F_A = F_A(x,k) = F_A(k)$;

– вычисление $e^x = \sum\left(x^n/n!\right)$ с точностью до x: $F_A = F_A(x,x)$.

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 параметров. В этом случае:

$$F_A(D) = F_A(|D|, P_1,...,P_m) = F_A(N, P_1,...,P_m).$$

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 количественно-параметрических по трудоемкости алгоритмов выделим еще оду группу, для которой количество операций зависит от порядка расположения исходных объектов.

Пусть множество D состоит из элементов (d_1,…,d_N), и $|D|=N$,

Определим $D_p = \{(d_1,...,d_N)\}$ – множество всех упорядоченных N-ок из $d_1,...,d_N$, отметим, что $|D_p| = N!$.

Если для некоторых $F_A(D_{pi}) \neq F_A(D_{pj})$, где $D_{pi}, D_{pi} \in D_p$ и $i \neq j$, то алгоритм будем называть порядково-зависимым по трудоемкости.

Примерами таких алгоритмов могут служить ряд алгоритмов сортировки, алгоритмы поиска максимального или минимального элементов в массиве. Например, в алгоритме поиска максимума в массиве количество операций присваивания нового значения переменной, хранящей текущее максимальное значение, будет зависеть от порядка следования элементов.

1.4 Вопросы для самоконтроля

1. Охарактеризуйте область практического применения анализа алгоритмов.
2. На соответствие каким требованиям должен быть проверен алгоритм перед началом анализа?
3. Какие ресурсы формальной системы входят в комплексную оценку алгоритма?
4. Назовите основные причины, вследствие которых можно пренебречь анализом трудоемкости алгоритма.
5. Дайте определения худшего, лучшего и среднего случая при анализе алгоритмов. Какие из этих оценок, на ваш взгляд, имеют большее практическое значение при анализе алгоритмов?
6. Назовите основные классы алгоритмов по виду функции трудоемкости, охарактеризуйте их.
7. Приведите примеры алгоритмов, относящихся по виду функции трудоемкости к различным классам.

2 Анализ трудоемкости алгоритмов

Можно выделить следующие основные методы анализа трудоемкости алгоритмов:

1. Определение вычислимости алгоритма в машине Тьюринга (или в другой формальной системе, например, машине Поста). В этом случае работа алгоритма описывается в виде функций переходов машины Тьюринга. При анализе подсчитывается число переходов, необходимое для решения задачи. Анализ потребностей в памяти подразумевает подсчет числа ячеек в ленте машины Тьюринга, необходимых для решения задачи. Такой анализ позволяет точно определить относительную скорость двух алгоритмов, однако его практическое осуществление чрезвычайно сложно (затруднения возникают при представлении решения задачи в виде функций переходов) и занимает много времени.

2. Экспериментальный анализ заключается в определении времени выполнения программы на конкретной вычислительной машине. При анализе алгоритмов этот метод представляет меньший интерес, так как оценивается трудоемкость не самого алгоритма, а реализующей его программы (зависит от набора команд процессора и времени их выполнения, объема встроенной кеш-памяти, наличия и эффективности потоковой обработки, деталей реализации программы, используемого компилятора и т.д.). Во-вторых, для проведения экспериментального анализа необходимо готовое приложение, реализующее алгоритм. То есть выбор одного из нескольких алгоритмов по результатам анализа возможен только после их реализации, что приводит к необоснованным затратам труда программиста в случае сложных алгоритмов. В-третьих, экспериментальный анализ не позволяет выявить «слабые места» алгоритма и оценить возможности снижения его трудоемкости.

3. Третий способ анализа алгоритмов предполагает, что алгоритм в общем виде записан на каком-либо императивном языке высокого уровня, таком как Pascal, C, C++ или достаточно общем

псевдокоде. На основе такого описания подсчитывается общее количество операций и выписывается функция трудоемкости. Этот метод представляет наибольший практический интерес вследствие относительной логической простоты, возможности проведения анализа до реализации алгоритма на языке программирования и возможности повышения эффективности алгоритма на основе интерпретации полученных оценок. В качестве недостатков метода можно отметить сложность учета операций при анализе некоторых классов алгоритмов и сложность перехода к временным оценкам при необходимости определения времени выполнения программы.

Дальнейшее рассмотрение практических инструментов анализа алгоритмов будет базироваться на последнем методе. Описание алгоритмов будет осуществляться на достаточно общем псевдокоде, который включает основные необходимые для представления алгоритмов элементарные операции и алгоритмические конструкции, и может быть легко интерпретирован при необходимости реализации алгоритма на императивном языке программирования.

2.1 Элементарные операции и конструкции в языке записи алгоритмов

Для получения функции трудоемкости алгоритма необходимо уточнить понятия «элементарных» операций, соотнесенных с языком высокого уровня.

В качестве таких «элементарных» операций предлагается использовать следующие:

– простое присваивание: «$a \neg b$»;

– одномерная индексация «*a[i]*»: (адрес (a)+i · длина элемента);

– арифметические операции: «*», «/», «–», «+», «*mod*», «*div*»;

– операции сравнения: «$a < b$», «$a > b$», «$a <= b$», «$a >= b$», «$a <> b$»;

– логические операции: «*or*», «*and*», «*not*».

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.

1. Конструкция «Следование»

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

$$F_{следование} = f_1 + f_2 + ... + f_k,$$

где k – количество блоков.

2. Конструкция «Ветвление»

if (l) then

группа операторов трудоемкостью f_{then}

(с вероятностью p)

else

группа операторов трудоемкостью f_{else}

(с вероятностью (1-p))

end if

Общая трудоемкость выполнения условного оператора складывается из трудоемкости выполнения условно исполняемых операторов f_{then} и f_{else} и трудоемкости вычисления самого логического выражения f_l. Причем учет трудоемкости условно исполняемых операторов требует анализа вероятности выполнения перехода p. Таким образом, трудоемкость конструкции «Ветвление» определяется как:

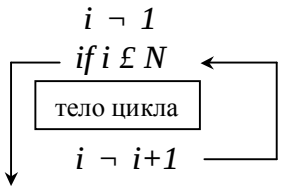
$$F_{ветвление} = f_l + p \cdot f_{then} + (1 - p) \cdot f_{else}.$$

3. Конструкция «Цикл»

Реализацию цикла «for» можно представить в следующем виде:

for i ¬ 1 to N do
тело цикла
end for

⇔



Очевидно, что цикл выполнится N раз, что дает «N трудоемкостей тела цикла» и $N \cdot 3$ вспомогательных операций (увеличение переменной цикла, присваивание и проверка условия). Помимо этого один раз выполняется начальное присваивание переменной цикла и одна проверка условия, после которой происходит выход из цикла. Таким образом, после сведения конструкции «Цикл» к элементарным операциям ее трудоемкость определяется как:

$$F_{цикл} = 1+1+3 \cdot N + N \cdot f_{тело\ цикла}$$

Таким же образом могут быть получены функции трудоемкости циклов с пред- и постусловием.

Например, рассмотрим следующую абстрактную процедуру:

for i ¬ 1 to n do	$1+1+3 \cdot n$
if (i mod 2 = 0) then	$n \cdot (1+1)$
for j ¬ 1 to n do	$n \cdot p \cdot (1+1+3 \cdot n)$
оператор трудоемкостью 2	$n \cdot p \cdot n \cdot 2$
end for	
else	
for j ¬ 1 to i do	$n \cdot (1-p) \cdot f_{цикл\_else}$
оператор трудоемкостью 1	
end for	
end if	
end for	

Требуется найти трудоемкость процедуры при четном n.

Процедура является параметрически-зависимой. Внешний цикл выполнится n раз, тогда общая трудоемкость процедуры определяется как:

$$F_A(n) = 1+1+3\cdot n + n\cdot f_{внеш},$$

где $f_{внеш}$ – трудоемкость внешнего цикла:

$$f_{внеш} = 1+1+ p\cdot f_{then} + (1-p)\cdot f_{else} =$$
$$= 2 + p\cdot f_{цикл\_then} + (1-p)\cdot f_{цикл\_else}.$$

Оператор проверки условия выполнится всего n раз для значений i от 1 до n. Так как n – четное, секция then выполнится $n/2$ раз, откуда получим p=1/2. Количество итераций цикла в секции else зависит от параметра i, поэтому имеет смысл рассматривать вклад этого цикла отдельно (во избежание усложнения анализа). Тогда получим:

$$f_{внеш} = 2 + \frac{1}{2}(1+1+3\cdot n + 2\cdot n) + (1-p)\cdot f_{цикл\_else} =$$
$$= \frac{5}{2}n + 3 + (1-p)\cdot f_{цикл\_else}.$$

Подставив полученную зависимость в выражение для общей трудоемкости, получим:

$$F_A(n) = 2 + 3n + n\left(\frac{5}{2}n + 3 + (1-p)\cdot f_{цикл\_else}\right) =$$
$$= 2 + 6n + \frac{5n^2}{2} + n\cdot(1-p)\cdot f_{цикл\_else}.$$

Заметим, что аналогичный результат можно получить, суммируя приведенные справа от алгоритма вклады каждого оператора.

Правый член выражения представляет собой вклад трудоемкости цикла секции else. Цикл выполняется $n/2$ раз, причем количество его итераций зависит от параметра, который принимает только нечетные значения от 1 до n. Таким образом, вклад этого цикла в общую трудоемкость можно представить следующей суммой:

$$\sum_{k=1}^{n/2}\left(1+1+3\cdot(2k-1)+1\cdot(2k-1)\right) = \sum_{k=1}^{n/2} 2 + \sum_{k=1}^{n/2} 4(2k-1) =$$

$$= 2 \cdot \frac{n}{2} + 8 \sum_{k=1}^{n/2} k - 4 \sum_{k=1}^{n/2} 1 = n + 8 \cdot \frac{n/2(n/2+1)}{2} - 4 \cdot \frac{n}{2} =$$

$$= n^2 + n .$$

Отсюда окончательно получим:

$$F_A(n) = 2 + 6n + \frac{5n^2}{2} + n^2 + n = 3\frac{1}{2} n^2 + 7n + 2 .$$

2.2 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. По заданной трудоемкости алгоритма $F_A(D)$ требуется определить время работы программной реализации алгоритма – $T_A(D)$.

Построение временных оценок приводит к ряду проблем, решение которых вызывает существенные трудности:

– несоответствие 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 типа операндов;

– неоднозначности компиляции исходного текста, обусловленные как самим компилятором, так и его настройками.

Различные попытки учета этих факторов привели к появлению разнообразных методик перехода к временным оценкам.

1. Пооперационный анализ

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

$$T_A(D) = \sum_i F_{aoni}(D) \cdot \bar{t}_{oni} .$$

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, создаются соответствующие тестовые программы, и определяется среднее время на операцию в данном типе задач $\bar{t}_{тип\ задачи}$.

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

$$T_A(D) = F_A(D) \cdot \bar{t}_{тип\ задачи}$$

3. Метод прямого определения среднего времени

В этом методе также 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:

$$\bar{t}_a = \frac{T_э(N_э)}{F_A(N_э)},$$

$$T_A(N) = \bar{t}_a \cdot F_A(N).$$

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:

$$(a + ib) \cdot (c + id) = (ac - bd) + i(ad + bc) = e + if\ .$$

1. Алгоритм А1 (прямое вычисление e, f – четыре умножения)

```
MultComplex1 (a, b, c, d; e, f)
        e ¬ a*c - b*d
        ¦ ¬ a*d + b*c
        Return (e, ¦)
end MultComplex1
```

Количество операций (всего и по группам)

f_{A1} =8 (всего);

f_* =4 (умножения);

$f_\pm$ =2 (сложения и вычитания);

$f_\leftarrow$ =2 (присваивания).

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

MultComplex2 (a, b, c, d; e, f)
z1 ¬ c(a + b)*
z2 ¬ b(d + c)*
z3 ¬ a(d - c)*
e ¬ z1 - z2
¦ ¬ z1 + z3
Return (e, ¦)
end MultComplex2

Количество операций: f_{A2}=13, f_{*}=3, $f_{\pm}$=5, $f_{\leftarrow}$=5 операций.

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 определить, 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:

$$t_{*} = q \cdot t_{+}, q > 1;$$
$$t_{\leftarrow} = r \cdot t_{+}, r < 1.$$

Тогда можно привести временные оценки двух алгоритмов к времени выполнения операции сложения/вычитания – t_{+}:

$$T_{A1} = 4 \cdot q \cdot t_{+} + 2 \cdot t_{+} + 2 \cdot r \cdot t_{+} = t_{+}(4q + 2 + 2r);$$
$$T_{A2} = 3 \cdot q \cdot t_{+} + 5 \cdot t_{+} + 5 \cdot r \cdot t_{+} = t_{+}(3q + 5 + 5r).$$

Равенство времен будет достигнуто при условии:

$$4q + 2 + 2r = 3q + 5 + 5r,$$

откуда:

$$q = 3 + 3r.$$

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 процедура перемножения используется только несколько раз, однако, если этот алгоритм является частью сложной вычислительной задачи с комплексными числами, требующей значительного количества умножений, то выигрыш во времени может быть ощутим. Оценка такого выигрыша на одно умножение комплексных чисел следует из только что проведенного анализа:

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

2.3 Особенности анализа среднего случая порядково-зависимых алгоритмов. Классы входных данных

Порядково-зависимые алгоритмы достаточно широко распространены. Особенность анализа среднего случая заключается в необходимости учитывать все возможные конкретные проблемы размерности N, то есть все расстановки множества входных значений. Множество различных комбинаций одного и того же входного набора данных может быть огромным. Рассмотрение каждой из них не представляется возможным. Например, число различных расстановок N различных чисел в списке есть $N!$.

С целью уменьшения количества рассматриваемых возможностей необходимо разбивать различные входные множества (конкретные проблемы) на классы в зависимости от поведения алгоритма на каждом множестве.

Например, в списке из N элементов осуществляется линейный поиск, причем допускается, что поиск всегда является удачным. Имеется $N!/N$ входных множеств, в которых первый элемент является

искомым. На каждом из этих множеств алгоритму для решения задачи потребуется одинаковое количество операций. Имеется также $N!/N$ входных множеств, в которых второй элемент является искомым и т.д. Для данной задачи получим N классов и сможем легко определить объем работы на каждом из них. Для получения оценки трудоемкости в среднем случае необходимо только усреднить результаты по количеству классов с учетом их веса.

Таким образом, анализ среднего случая можно разбить на несколько этапов:

1. Определяются классы наборов входных данных, на которых алгоритм дает одинаковые оценки трудоемкости.
2. Определяется вероятность, с которой входной набор данных принадлежит каждому классу.
3. Определяются функции трудоемкости алгоритма на данных каждого класса.
4. Функция трудоемкости в среднем случае определяется по формуле:

$$F_A(N) = \sum_{i=1}^{m} p_i F_{Ai}(N),$$

где N – размер набора входных данных;

m – число классов;

p_i – вероятность того, что набор входных данных принадлежат классу с номером i;

$F_{Ai}(N)$ – функция трудоемкости алгоритма на наборе данных из класса i.

Достаточно часто, в случае невозможности или сложности точного определения весов классов, предполагают, что вероятности попадания входных данных в каждый из r классов одинаковы. В этом случае среднее время работы можно оценить по эквивалентной упрощенной формуле:

$$F_A(N) = \frac{1}{m} \sum_{i=1}^{m} F_{Ai}(N).$$

Например, рассмотрим алгоритм быстрого линейного поиска:

```
Search (r, n, x)
        r[n+1] ¬ x                          3
        i ¬ 1                               1
        while x<>r[i] do                    2+2·k
            i ¬ i+1                         2·k
    end while
    if i<=n then                            1
            return (i)
    else
            return (0)
    end if
end Search
```

Примечание: трудоемкость операций возврата значений не приведена и не рассматривается, так как обычно включается в трудоемкость механизма вызова процедуры. Принципы анализа трудоемкости механизма вызова будут обсуждаться при рассмотрении вопросов анализа рекурсивных алгоритмов, где эта составляющая играет более важную роль вследствие неоднократности вызовов.

Необходимо определить трудоемкость алгоритма в среднем случае, если вероятность того, что аргумент поиска присутствует в списке p=0,5.

На основе приведенного выше количества операций можно определить трудоемкость алгоритма в зависимости от количества итераций цикла while:

$$f_A(k) = 3+1+2+2k+2k+1 = 4k+7.$$

Вначале необходимо выделить два класса входных наборов данных: набор данных, содержащий аргумент поиска и не содержащий его. Тогда функцию трудоемкости можно определить как

$$\overline{F}_A(n) = p \cdot f_{удач} + (1-p) f_{неудач},$$

где $f_{удач}$ – трудоемкость алгоритма в случае, когда аргумент поиска присутствует в исходном наборе данных (имеется в виду трудоемкость в среднем случае при равных вероятностях обнаружения искомого элемента в каждой позиции);

$f_{неудач}$ – трудоемкость алгоритма в худшем случае, то есть при отсутствии аргумента поиска в исходном наборе данных.

В случае неудачного поиска должны быть просмотрены все элементы списка, и выход из цикла происходит при $i=n+1$. То есть осуществляется $k=n$ итераций цикла while. Отсюда трудоемкость алгоритма в случае неудачного поиска

$$f_{неудач} = 4n + 7.$$

При определении трудоемкости в случае удачного поиска также можно говорить о различных классах входных данных: искомый элемент находится в первой позиции (цикл while не выполнится ни разу), искомый элемент находится во второй позиции (цикл while выполнится 1 раз), …, искомый элемент находится в последней позиции (цикл while выполнится n-1 раз) – всего n возможных классов. Предполагая, что появление искомого элемента в любой позиции равновероятно, получим:

$$f_{удач} = \frac{1}{n}\sum_{k=0}^{n-1}(4k+7) = \frac{4}{n}\sum_{k=0}^{n-1}k + \frac{1}{n}\sum_{i=0}^{n-1}7 =$$

$$= \frac{4}{n}\cdot\frac{n(n-1)}{2} + 7 = 2n + 5.$$

Сводя воедино результаты, получим:

$$\overline{F}_A(n) = \frac{1}{2}(2n+5) + \frac{1}{2}(4n+7) = 3n + 6.$$

2.4 Вопросы для самоконтроля

1. Перечислите и охарактеризуйте основные методы анализа алгоритмов.

2. Приведите виды функций трудоемкости для основных алгоритмических конструкций.

3. Самостоятельно выпишите функции трудоемкости для циклов «while» и «repeat».

4. Какими особенностями анализа трудоемкости может быть обусловлена необходимость перехода к временным оценкам?

5. Перечислите и охарактеризуйте методики перехода к временным оценкам.

6. Чем обусловливается сложность анализа среднего случая порядково-зависимых алгоритмов? В чем заключается подход, позволяющий упростить процедуру анализа?

7. Приведите и опишите этапы анализа среднего случая порядково-зависимых алгоритмов.

2.5 Задачи

Примечание: в задачах на анализ трудоемкости итеративных алгоритмов допускается не учитывать трудоемкость механизма вызова процедуры.

Задача 2.1.

Дан алгоритм определения суммы элементов квадратной матрицы:

```
MatrS (A, n)
    S ¬ 0
    for i ¬ 1 to n do
        for j ¬ 1 to n do
            S ¬ S+A[i,j]
        end for
    end for
    return (S)
end MatrS
```

Определите трудоемкость алгоритма как функции от размерности матрицы n (двумерную индексацию необходимо рассматривать как две операции).

Ответ: $F_A(n) = 7n^2 + 5n + 3$.

Задача 2.2.

Дан алгоритм определения суммы элементов квадратной матрицы, расположенных не на главной диагонали:

```
MatrSD (A, n)
    S ¬ 0
    for i ¬ 1 to n do
        for j ¬ 1 to n do
            if i<>j then
                S ¬ S+A[i,j]
            enf if
        end for
    end for
    return (S)
end MatrSD
```

Определите трудоемкость алгоритма как функции от размерности матрицы n (двумерную индексацию необходимо рассматривать как две операции).

Ответ: $F_A(n) = 8n^2 + n + 3$.

Задача 2.3.

Имеется алгоритм перемножения квадратных матриц:

```
MatrM (A, B, n; C)
    for i ¬ 1 to n do
        for j ¬ 1 to n do
```

```
                C[i,j] ¬ 0
                for k ¬ 1 to n do
                        C[i,j] ¬ C[i,j]+A[i,k]*B [k,j]
                end for
            end for
        end for
    end MatrM
```

Определите трудоемкость алгоритма как функции от размерности матриц n (двумерную индексацию необходимо рассматривать как две операции).

Ответ: $F_A(n) = 14n^3 + 8n^2 + 5n + 2$.

Задача 2.4.

Имеется абстрактная процедура:

```
for i ¬ 1 to n do
    if (i mod 3 = 1) then
        for j ¬ 1 to n do
            оператор трудоемкостью 2
        end for
    else
        оператор трудоемкостью 5
    end if
end for
```

Определите трудоемкость процедуры как функции от n кратного 3.

Ответ: $F_A(n) = 2\frac{1}{3}n^2 + 9n + 2$.

Задача 2.5.

Имеется абстрактная процедура:

```
for i ¬ 1 to n do
    if (i mod 3 = 1) then
        for j ¬ 1 to n do
            оператор трудоемкостью 2
        end for
    else
        if (i mod 3 = 0) then
            оператор трудоемкостью 5
        end if
    end if
end for
```

Определите трудоемкость процедуры как функции от *n* кратного 3.

Ответ: $F_A(n) = 2\frac{1}{3}n^2 + 8\frac{2}{3}n + 2$.

Задача 2.6.

Имеется абстрактная процедура:

```
for i ¬ 1 to n do
    if (i mod 3 = 0) then
        for j ¬ 1 to n do
            оператор трудоемкостью 1
        end for
    else
        for j ¬ 1 to i do
            оператор трудоемкостью 2
        end for
```

```
    end if
end for
```

Определите трудоемкость процедуры как функции от *n* кратного 3.

Ответ: $F_A(n) = 1\frac{2}{3}n^2 + 2n + 3.$

Задача 2.7.

Имеется итеративный алгоритм вычисления *n*-го числа Фибоначчи (F_0=0, F_1=1, $Fn = F_{n-1} + F_{n-2}$):

```
Fib (n)
    if n = 0 then
        return (0)
    else
        b ¬ 0
        a ¬ 1
        for i ¬ 1 to n do
            c ¬ a+b
            b ¬ a
            a ¬ c
        end for
        return (c)
    end if
end Fib
```

Определите трудоемкость алгоритма как функции от *n*>0.

Ответ: $F_A(n) = 7n + 5.$

Задача 2.8.

Пусть имеется число *n*, являющееся положительной целой степенью двойки: *n* =2, 4, 8, … . Для вычисления логарифма *n* используется следующий алгоритм:

```
log (n)
    l ¬ 1
    x ¬ 2
    while x<n do
        x ¬ 2*x
        l ¬ l+1
    end while
    return (l)
end log
```

Определите трудоемкость алгоритма как функции от *n*.

Ответ: $F_A(n) = 5\log_2 n - 2$.

Задача 2.9.

Дан алгоритм линейного поиска в списке, содержащем *n* значений:

```
Search (r, n, x)
    i ¬ 1;
    while (i<=n) and (r[i]<>x) do
        i ¬ i+1
    end while
    if i>n then
        return (0)
    else
        return (i)
    end if
end Search
```

Определите трудоемкость алгоритма в: а) худшем, б) лучшем и в) среднем случаях, если вероятность того, что элемент не будет найден в списке p=1/3.

Ответ:

а) $F_A^{\wedge}(n) = 6n + 6$;

б) $F_A^{\vee}(n) = 6$;

в) $\overline{F}_A(n) = 4n + 4$.

Задача 2.10.

Для нахождения минимального элемента массива используется следующий алгоритм:

```
Min (a, n)
    min ¬ a[1]
    for i ¬ 2 to n do
        if a[i]<min then
            min ¬ a[i]
        end if
    end for
    return (min)
end Min
```

Определите трудоемкость алгоритма в: а) худшем, б) лучшем и в) среднем случаях. При анализе среднего случая предполагается, что вероятность истинности логического выражения в операторе *if* равна 0,5.

Ответ:

а) $F_A^{\wedge}(n) = 7n - 3$;

б) $F_A^{\vee}(n) = 5n - 1$;

в) $\overline{F}_A(n) = 6n - 2$.

Задача 2.11.

Поиск делением пополам в упорядоченном массиве реализуется следующим алгоритмом (с целью упрощения процедуры анализа используется алгоритм, отличающийся от классического):

```
SearchB (a, n, x)
    L ¬ 1
    R ¬ n
    i ¬ (L+R) div 2
    while (L<=R) and (x<>a[i]) do
        if x<a[i] then
            R ¬ i-1
        else
            L ¬ i+1
        end if
        i ¬ (L+R) div 2
    end while
    if x=a[i] then
        return (i)
    else
        return (0)
    end if
end SearchB
```

Определите трудоемкость алгоритма в: а) худшем, б) лучшем и в) среднем случаях, предполагая, что:

– количество элементов в списке $n=2^k-1$ для некоторого $k>0$;

– искомое значение всегда присутствует в списке;

– вероятность истинности логического выражения в операторе *if* цикла *while* равна 0,5.

Ответ:

а) $F_A^{\wedge}(n) = 11\log_2(n+1)$;

б) $F_A^{\vee}(n) = 11$;

в) $\overline{F}_A(n) = 11 \cdot \frac{n+1}{n}\log_2(n+1) - 11.$

Задача 2.12.

Решите предыдущую задачу при следующих условиях:

– количество элементов в списке $n=2^k - 1$ для некоторого $k>0$;

– вероятность того, что элемент не будет найден в списке $p=0{,}4$;

– вероятность истинности логического выражения в операторе *if* цикла *while* равна 0,5.

Ответ:

а) $F_A^{\wedge}(n) = 11\log_2(n+1) + 11$;

б) $F_A^{\vee}(n) = 11$;

в) $\overline{F}_A(n) = 11 \cdot \frac{5n+3}{5n}\log_2(n+1) - 2\frac{1}{5}.$

3 Асимптотический анализ алгоритмов

Очевидно, что получение точной функции трудоемкости для многих алгоритмов является сложной задачей. При этом необходимо иметь достаточно точное описание алгоритма на псевдокоде или его реализацию на каком-либо языке высокого уровня. Такой подход неэффективен на начальном этапе выбора одного из нескольких алгоритмов, решающих данную задачу.

В такой ситуации применяют асимптотический анализ алгоритмов, являющийся менее сложным и емким по времени. Суть данного анализа заключается в нахождении скорости роста (или порядка) функции трудоемкости для больших значений размера набора входных данных.

Получаемая в процессе асимптотического анализа оценка называется сложностью алгоритма. Ее соотношение с функцией трудоемкости можно выразить следующими допущениями при больших размерностях входного набора данных:

1. Постоянные коэффициенты при членах функции трудоемкости считаются незначимыми.

2. Быстрорастущие члены функции доминируют над членами с более медленным ростом. То есть членами меньшего порядка можно пренебречь.

Например, при анализе алгоритма установлено, что он делает $N^3 - 30N$ сравнений. Скорость роста функции трудоемкости или сложность алгоритма определяется, как N^3. При этом можно отметить, что уже при объеме входных данных N=100 разница между N^3 и $N^3 - 30N$ составляет лишь 0,3%.

При анализе поведения функции трудоемкости алгоритма обычно используют принятые в математике асимптотические обозначения, позволяющие показать скорость роста функции.

3.1 Скорости роста функций: определения и классификация

Алгоритмы можно сгруппировать (выделить классы) по скорости роста их трудоемкости:

– алгоритмы, трудоемкость которых растет не медленнее, чем заданная функция;

– алгоритмы, трудоемкость которых растет не быстрее, чем заданная функция;

– алгоритмы, трудоемкость которых растет с той же скоростью, что и заданная функция.

При анализе алгоритма в качестве классообразующих принимают следующие функции от n: 1; $\log n$; $n \log n$; n^k, где k – положительное целое число (обычно не больше 6); 2^n; $n!$.

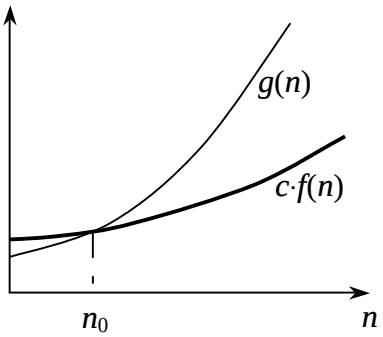
Пусть $f(n)$ и $g(n)$ – положительные функции аргумента $n \geq 1$ (количество параметров на входе и количество операций – положительные числа), тогда можно привести следующие обозначения асимптотического анализа:

1. Оценка W ***(Омега большое)***

$\Omega(f(n))$ определяет класс функций, растущих не медленнее, чем $f(n)$, с точностью до постоянного множителя. То есть класс $\Omega(f(n))$ задается указанием свой нижней границы, и такую оценку называют нижней оценкой сложности алгоритма. Функция $g(n)$ принадлежит этому классу, если при всех значениях аргумента n, больших некоторого порогового значения n_0, выполняется условие $g(n) \geq c \cdot f(n)$ для некоторого положительного числа c.

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

Поэтому, с точки зрения анализа сложности, класс Ω представляет меньший интерес, так как принадлежность функции трудоемкости g некоторого алгоритма к классу $\Omega(f(n))$ говорит только о том, что сложность алгоритма растет так же быстро или



быстрее, чем $f(n)$.

Рисунок 3.1 – Графическая иллюстрация выражения $g(n) = \Omega(f(n))$

Однако, как будет показано ниже, информация о принадлежности функции трудоемкости алгоритма к этому классу имеет практическое значение для доказательства принадлежности этой функции к другому классу, представляющему больший интерес.

2. Оценка* O *(О большое)

$O(f(n))$ определяет класс функций, растущих не быстрее $f(n)$ с точностью до постоянного коэффициента. Функция $f(n)$ образует верхнюю границу для класса $O(f(n))$. Функция $f(n)$ принадлежит классу $O(f(n))$, если $g(n) \le c \cdot f(n)$ для всех n, больших некоторого порога n_0, и некоторой положительной константы c.

Например, для всех функций: $g(n) = 10$, $g(n) = \frac{1}{n}$, $g(n) = 2n - 15$, $g(n) = n\log_2 n$, $g(n) = 10n^2 - 3n$, будет справедлива оценка $O(n^2)$.

Класс $O(f(n))$ имеет большее практическое значение, так как принадлежность функции трудоемкости алгоритма (например, в худшем случае) к нему говорит о том, что сложность алгоритма растет не быстрее чем $f(n)$.

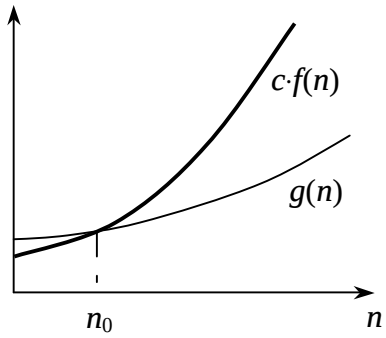


Рисунок 3.2 – Графическая иллюстрация выражения $g(n) = \mathrm{O}(f(n))$

3. Оценка* Q *(тетта)

$\Theta(f(n))$ определяет класс функций, растущих с той же скоростью, что и $f(n)$. Этот класс представляет собой пересечение двух предыдущих классов:

$$\Theta(f(n)) = \Omega(f(n)) \cap \mathrm{O}(f(n)).$$

Функция $g(n)$ принадлежит классу $\Theta(f(n))$, если найдутся такие константы $c_1>0$ и $c_2>0$, что при всех $n>n_0$ выполняется условие $c_1 \cdot f(n) \le g(n) \le c_2 \cdot f(n)$.

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

Примеры:

– для функции $g(n) = 2n^2 + n\log_2 n - 15$ справедлива следующая оценка: $g(n) = \Theta(n^2)$;

– $g(n) = \Theta(1)$ – запись означает, что $g(n)$ или равна константе, не равной нулю, или $g(n)$ ограничена константой на ∞, например: $g(n) = 4 + \frac{1}{n}$.

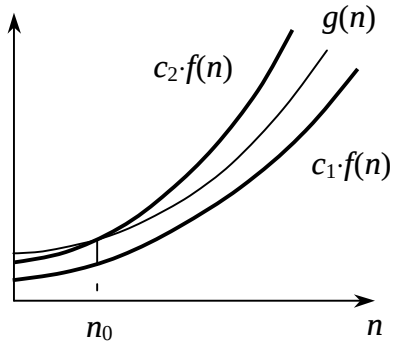


Рисунок 3.3 – Графическая иллюстрация выражения $g(n) = \Theta(f(n))$

3.2 Особенности использования асимптотических оценок

Доказательство принадлежности

Доказательство принадлежности функции $g(n)$ к какому-либо классу сложности, заданному функцией $f(n)$ можно осуществить двумя способами: либо с помощью данных выше определений оценок, либо найдя предел $\lim\limits_{n\to\infty}\dfrac{g(n)}{f(n)} = c$ и воспользовавшись следующими утверждениями:

– $g(n) \in \mathrm{O}(f(n))$, если $0 \le c < \infty$;

– $g(n) \in \Omega(f(n))$, если $0 < c \le \infty$;

– $g(n) \in \Theta(f(n))$, если $0 < c < \infty$.

Давая оценки O и Ω, есть смысл указывать наиболее «близкую» мажорирующую функцию (из приведенного выше ряда классообразующих функций), поскольку, например, для $g(n) = 10n^2 - 3n$ справедливы оценки $\mathrm{O}(2^n)$ и $\mathrm{O}(n!)$, однако они не имеют практического смысла. Поэтому при получении c=0 для оценки O или $c = \infty$ для оценки Ω необходимо попытаться доказать оценку для функции $f(n)$ с более низкой или более быстрой высотой роста соответственно для получения постоянной c.

Особенности обозначений

В асимптотическом анализе принято обозначение принадлежности скорости роста функции к определенному классу в виде $g(n) = O(f(n))$, что эквивалентно записи $g(n) \in O(f(n))$.

Свойства оценок

Введенные определения обладают следующими свойствами:

Транзитивность:

$$g(n) = \Theta(f(n)) \text{ и } f(n) = \Theta(h(n)) \Rightarrow g(n) = \Theta(h(n));$$

$$g(n) = O(f(n)) \text{ и } f(n) = O(h(n)) \Rightarrow g(n) = O(h(n));$$

$$g(n) = \Omega(f(n)) \text{ и } f(n) = \Omega(h(n)) \Rightarrow g(n) = \Omega(h(n)).$$

Рефлексивность:

$$g(n) = \Theta(g(n));$$

$$g(n) = O(g(n));$$

$$g(n) = \Omega(g(n)).$$

Симметричность:

$$g(n) = \Theta(f(n)) \text{ если и только если } f(n) = \Theta(g(n)).$$

Обратимость:

$$g(n) = O(f(n)) \text{ если и только если } f(n) = \Omega(g(n)).$$

3.3 Учет операций при анализе сложности алгоритмов

Тот факт, что асимптотические оценки трудоемкости алгоритмов даются с точностью до постоянного множителя перед членом высшего порядка, позволяет значительно упростить учет элементарных операций при анализе сложности. Например, нет необходимости учитывать операции, выполняемые в теле цикла от 1 до N, если их число невелико и постоянно, так как этот коэффициент при N не повлияет на полученную асимптотическую оценку. Учтенная операция начальной установки переменной цикла также будет отброшена как член функции трудоемкости с меньшей скоростью роста.

Поэтому при анализе сложности алгоритмов имеет смысл учитывать только «значимые» операции. Их выбор осуществляется в два этапа:

– выбирается значимая операция или группа операций, скорость роста количества которых соответствует скорости роста алгоритма (например, скорость роста количества операций в цикле линейна – c_1n+c_2, тогда в качестве значащей может быть выбрана операция проверки условия на выход из цикла, так как она выполняется $n+1$ раз);

– из этих операций отбираются те, которые содержатся в теле алгоритма и определяют его сложность (операции, составляющие накладные расходы, не рассматриваются).

В качестве значимых обычно выступают операции двух типов: операции проверки условия и арифметические операции. Все операторы проверки условия считаются эквивалентными. Их обычно учитывают в алгоритмах поиска и сортировки.

Арифметические операции могут разбиваться на две группы: аддитивные и мультипликативные. Аддитивные операторы (называемые для краткости «сложениями») включают сложение, вычитание, увеличение и уменьшение счетчика. Мультипликативные операторы (или «умножения») включают умножение, деление и взятие остатка по модулю. Разбиение на эти две группы связано с тем, что умножения работают дольше, чем сложения. На практике некоторые алгоритмы используют логарифмы и тригонометрические функции, которые образуют еще одну, более времяемкую, чем умножения, группу операций (обычно их значения вычисляются с помощью разложений в ряд).

Целочисленное умножение или деление на степень двойки могут образовывать специальный случай. Эти операции сводится к сдвигу, а последний по скорости эквивалентен сложению. Однако случаев, где эта разница существенна, немного, поскольку умножение и деление на 2 встречаются в первую очередь в алгоритмах рекурсивного типа, где значимую роль зачастую играют операторы проверки условия.

Особо выделяются алгоритмы, работающие с внешней памятью. Базовой операцией, выполняемой по отношению к файлам, является перенос одного блока в буфер, находящийся в основной памяти, или запись содержимого буфера в файл. Природа устройств внешней памяти такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико в сравнении со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке. Поэтому при анализе алгоритмов, в которых используются данные, хранящиеся во внешней памяти, чаще всего значащей операцией является обращение к блоку внешней памяти (чтение или запись).

Проиллюстрируем процедуру выбора значимых операций на примере приведенного выше алгоритма быстрого линейного поиска. Дадим асимптотические оценки трудоемкости алгоритма (определим сложность алгоритма) в лучшем, худшем и среднем случаях. Предположение о равной вероятности удачного и неудачного поиска сохраняется.

В качестве значимой операции при анализе алгоритмов поиска обычно выбирается операция проверки условия на равенство/неравенство аргумента поиска и текущего элемента списка. Действительно, количество операций проверки условия в данном алгоритме определяет число итераций цикла. При анализе сложности будем учитывать только операцию проверки условия в цикле while, так как при больших n трудоемкость выполнения оператора if дает незначительный вклад в общую трудоемкость и его можно отнести к накладным расходам на обработку результата работы значащей части алгоритма.

В лучшем случае искомый элемент будет располагаться в первой позиции списка. Тогда алгоритм выполнит одну операцию проверки условия (цикл while не выполнится ни разу). Тогда сложность алгоритма в лучшем случае (best) $B(n) = \Theta(1)$.

В худшем случае искомый элемент не будет найден, и выполнение цикла будет остановлено при достижении барьера в

позиции n+1. Цикл выполнится n раз, что дает в худшем случае (worst) $W(n) = n+1$операций проверки условия, то есть $W(n) = \Theta(n)$.

При анализе среднего случая также необходимо выделить два класса: удачный и неудачный поиск. Оценка неудачного поиска n+1 определена выше. В случае удачного поиска выделяются n равновероятных классов: элемент найден в первой позиции – 1 операция проверки условия, элемент найден во второй позиции – 2 операции проверки условия, …, элемент найден в n-ой позиции – n операция проверки условия. Отсюда для среднего случая (average) получим:

$$A(n) = \frac{1}{2}(n+1) + \frac{1}{2} \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} i .$$

Раскрывая сумму и приводя подобные, получим:

$$A(n) = \frac{n}{2} + \frac{1}{2} + \frac{1}{2n} \cdot \frac{n(n+1)}{2} = \frac{3}{4}n + \frac{3}{4}.$$

Следовательно $A(n) = \Theta(n)$.

3.4 Вопросы для самоконтроля

1. Какие допущения принимаются при определении скорости роста функции?
2. Перечислите и дайте определения обозначениям, применяемым при асимптотическом анализе.
3. Приведите примеры пар функций $f(n)$ и $g(n)$, удовлетворяющих условиям различных классов.
4. Приведите примеры пар функций, которые не удовлетворяют условиям ни одного из классов.
5. Перечислите и поясните основные свойства асимптотических оценок.
6. Что понимается под термином «значащая операция» при анализе сложности? Опишите и поясните на примере процедуру выбора значащей операции.
7. Операции каких видов обычно выбираются в качестве значимых при анализе сложности?

8. Какие рекомендации можно дать при выборе значащей операции для алгоритмов сортировки и поиска, численных алгоритмов, алгоритмов, работающих во внешней памяти.

3.5 Задачи

Задача 3.1.

Даны следующие функции от n:

$f_1(n) = n^2$;

$f_2(n) = n^2 + 100n$;

$f_3(n) = n^3 - 60$.

Для каждой пары функций укажите, когда $f_i(n)$ принадлежит классу $O(f_j(n))$, и когда $f_j(n) = \Omega(f_i(n))$. Докажите справедливость сделанных предположений.

Задача 3.2.

Даны следующие функции от n:

$f_1(n) = n \log_2 n$;

$f_2(n) = n^2$;

$f_3(n) = n\sqrt{n}$.

Для каждой пары функций укажите, когда $f_i(n)$ принадлежит классу $O(f_j(n))$, и когда $f_j(n) = \Omega(f_i(n))$. Докажите справедливость сделанных предположений.

Задача 3.3.

Определите и докажите верхнюю и нижнюю оценки скорости роста следующих функций:

$g(n) = 8n + 5$;

$g(n) = 5(\log_2 n)^2$.

Задача 3.4.

Определите и докажите верхнюю и нижнюю оценки скорости роста следующих функций:

$g(n) = \frac{n^2 - 3n}{3}$;

$g(n) = \log_2 \log_2 n$.

Задача 3.5.

Определите и докажите верхнюю и нижнюю оценки скорости роста следующих функций:

$g(n) = 4n\sqrt{n}$;

$g(n) = \sqrt{n} \log_2 n$.

Задача 3.6.

Расположите следующие функции в порядке возрастания скорости роста:

n;

$\sqrt{n}$;

$\log_2 n$;

$\sqrt{n} \log_2 n$;

$(\log_2 n)^2$;

$n / \log_2 n$.

Докажите справедливость сделанной расстановки.

Задача 3.7.

Оцените сложность алгоритма быстрого линейного поиска, приведенного в теоретической части, в среднем случае. При анализе принимаются следующие допущения:

– количество элементов в списке n – четное;

– вероятность того, что целевое значение будет найдено $p_{уд}$=0,25;

– вероятность того, что целевое значение будет найдено в первой половине списка, в случае, если оно вообще есть в списке, p_1=0,75.

Ответ: $A(n) = \frac{27}{32}n + \frac{7}{8}$, $A(n) = \Theta(n)$.

Задача 3.8.

Дан алгоритм поиска делением пополам в упорядоченном массиве длины n.

```
SearchB (a, n, x)
    L ¬ 1
    R ¬ n
    while (L<=R) do
        i ¬ (L+R) div 2
        if x<a[i] then
            R ¬ i-1
        else
            if x>a[i] then
                L ¬ i+1
            else
                break
            end if
        end if
    end while
    if x=a[i] then
        return (i)
    else
        return (0)
    end if
end SearchB
```

Найдите асимптотические оценки сложности алгоритма в а) худшем, б) лучшем и в) среднем случаях по количеству операций проверки условия (в качестве значимой принять одну операцию проверки условия в теле цикла while). При анализе принимаются следующие допущения:

– количество элементов в списке $n = 2^k - 1$ для некоторого целого $k>0$;

– искомое значение может отсутствовать в списке;

– вероятность удачного поиска $p=0,5$.

Ответ:

а) $W(n) = \log(n+1)$, $W(n) = \Theta(\log n)$;

б) $B(n) = 1$, $B(n) = \Theta(1)$;

в) $A(n) = \log(n+1) + \frac{\log(n+1)}{2n} - \frac{1}{2}$, $A(n) = \Theta(n)$.

Задача 3.9.

Дан алгоритм сортировки простыми вставками:

```
InsertionSort(list, n)
    for i ¬ 2 to N do
        newEl ¬ list[i]
        list[0] ¬ newEl
        j ¬ i
        while (newEl<list [j-1]) do
            list[j] ¬ list[j-1]
            j ¬ j-1
        end while
        list[j] ¬ newEl
    end for
end InsertionSort
```

Произведите анализ сложности алгоритма в: а) худшем, б) лучшем и в) среднем случаях. При анализе среднего случая принимается, что ситуации вставки элемента во все позиции отсортированной части равновероятны.

Примечание. Анализ среднего случая следует производить в два этапа: вначале определяется среднее число сравнений $C(i)$, необходимое для определения положения i-го вставляемого элемента (т.е. сложность прохода в зависимости от его номера i), затем среднее число всех необходимых для сортировки операций на основе первого шага (сумма по всем проходам).

Ответ:

а) $W(n) = \frac{n^2 + n}{2} - 1,\ W(n) = \Theta(n^2)$;

б) $B(n) = n - 1,\ B(n) = \Theta(n)$;

в) $A(n) = \frac{1}{4}n^2 + \frac{3}{4}n - 1,\ A(n) = \Theta(n^2)$.

Задача 3.10.

Дан алгоритм сортировки простым обменом (пузырьковая сортировка):

```
BubbleSort (list, n)
    j ¬ N
    sw ¬ true
    while sw do
        j ¬ j-1
        sw ¬ false
        for i ¬ 1 to j do
            if list[i] > list[i+1] then
                Swap(list[i], list[i+1]) //перестановка элементов
                sw ¬ true
            end if
```

end for

end while

end BubbleSort

Произведите анализ сложности алгоритма в: а) худшем, б) лучшем и в) среднем случаях по количеству операций проверки условия. При анализе среднего случая принимается, что завершение процесса сортировки вследствие отсутствия перестановок равновероятно после всех проходов.

Примечание. Анализ среднего случая следует производить в два этапа: вначале определяется число сравнений $C(i)$, выполненное на первых i проходах, затем сложность в среднем случае как среднее $C(i)$ по всем проходам, на которых может произойти остановка.

Ответ:

а) $W(n) = \frac{n^2 - n}{2}$, $W(n) = \Theta(n^2)$;

б) $B(n) = n - 1$, $B(n) = \Theta(n)$;

в) $A(n) = \frac{1}{3}n^2 - \frac{1}{6}n$, $A(n) = \Theta(n^2)$.

4 Анализ рекурсивных алгоритмов

Рекурсивные алгоритмы находят применение при решении широкого спектра задач. Их анализ более сложен по сравнению с анализом итеративных алгоритмов. Это обусловливается спецификой учета трудоемкости различных частей рекурсивного алгоритма, необходимостью анализа затрат на рекурсивные вызовы и сложностью интерпретации полученных в виде рекуррентных зависимостей результатов. Поэтому рассмотрению принципов анализа рекурсивных алгоритмов посвящен отдельный раздел.

4.1 Рекурсивная реализация алгоритмов

Большинство современных языков высокого уровня поддерживают механизм рекурсивного вызова, когда функция, как элемент структуры языка программирования, может вызывать сама себя с другим аргументом. Эта возможность позволяет напрямую реализовывать вычисление рекурсивно определенных функций. Однако необходимо отметить, что любой рекурсивный алгоритм может быть реализован итеративно.

Рассмотрим пример рекурсивной функции, вычисляющей факториал:

```
F(n)
    if n<=1 then //проверка возможности прямого вычисления
        F ¬ 1
    else
        F ¬ n*F(n-1)        //рекурсивный вызов функции
    end if
    return (F)
end if
```

Трудоемкость рекурсивных реализаций алгоритмов, очевидно, будет зависеть как от количества операций (как в базовом, так и в общем случаях), выполняемых при одном вызове функции, так и от количества таких вызовов. Графическое представление порождаемой

данным алгоритмом цепочки рекурсивных вызовов называется деревом рекурсивных вызовов. Оно может быть использовано для определения количества вызовов и их трудоемкости. 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, рекурсия эквивалентна конструкции цикла, в котором каждый проход есть выполнение рекурсивной функции с заданным параметром.

Рассмотрим в качестве примера функцию вычисления факториала. На рисунке 4.1 приведена схема организации процесса вычисления для n=5:

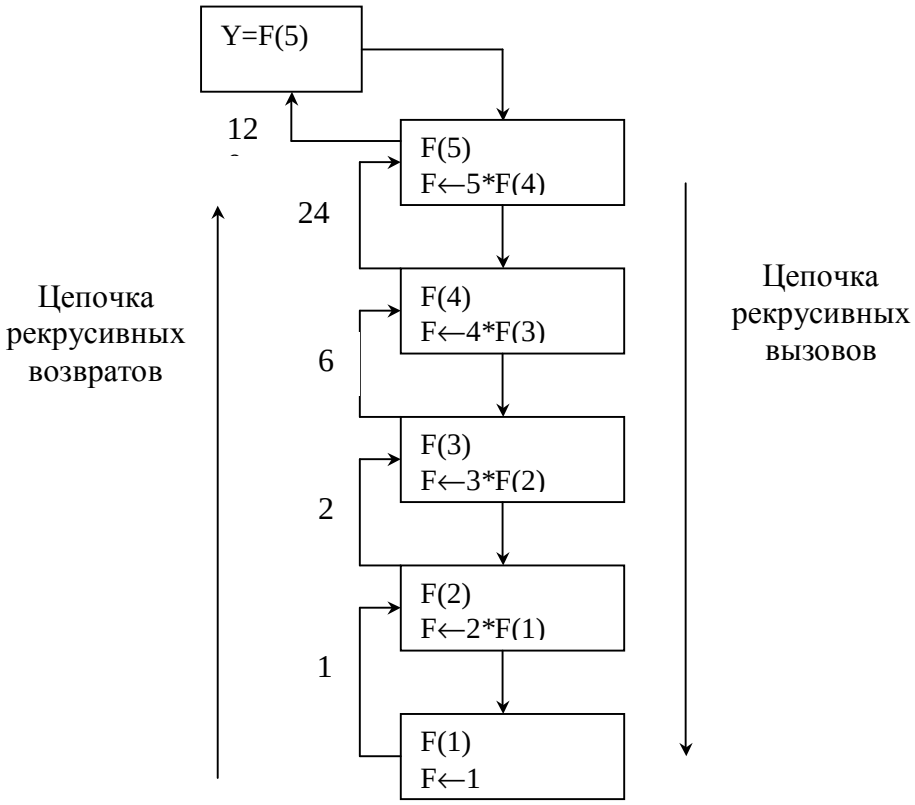


Рисунок 4.1 – Детальная схема вызовов функции вычисления факториала

Дерево рекурсивных вызовов может иметь и более сложную структуру, если в каждом вызове порождается несколько обращений. Например, на рисунке 4.2. приведено дерево рекурсивных вызовов функции вычисления чисел Фибоначчи для n=3.

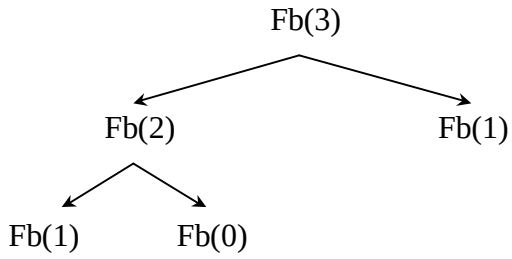


Рисунок 4.2 – Дерево рекурсивных вызовов функции вычисления числа Фибоначчи

4.2 Анализ трудоемкости рекурсивных алгоритмов

На основе сказанного выше можно выделить два основных этапа процедуры анализа трудоемкости алгоритма:

– анализ трудоемкости механизма вызова процедуры $f_{вызовов}(n)$;

– анализ трудоемкости самих рекурсивных вызовов $f_A(n)$.

Результирующая функция трудоемкости определяется следующим образом:

$$F_A(n) = f_{вызовов}(n) + f_A(n).$$

Определение трудоемкости самого рекурсивного вызова можно выполнять разными способами в зависимости от того, как формируется итоговая сумма элементарных операций:

– построение рекурсивной функции трудоемкости на основе анализа механизма декомпозиции задачи;

– прямой анализ рекурсивного дерева вызовов.

4.2.1 Анализ трудоемкости механизма вызова процедуры

Механизм вызова функции или процедуры, заданной на языке высокого уровня, существенно зависит от архитектуры компьютера и операционной системы, компилятора и используемой модели управления механизмом вызова. Обычно вызов реализуется через программный стек. Как передаваемые в процедуру или функцию фактические параметры, так и адреса возвращаемых значений помещаются в программный стек специальными командами процессора. Дополнительно сохраняются значения необходимых регистров и адрес возврата в вызывающую процедуру (адрес команды, следующей за вызовом подпрограммы). Схематично этот механизм представлен на рисунке 4.3.

Для подсчета трудоемкости вызова будем считать операции помещения слова в стек и выталкивания из стека элементарными операциями в формальной системе. Обозначим:

m – количество передаваемых фактических параметров,

k – количество возвращаемых процедурой значений,

r – количество сохраняемых в стеке регистров.

Тогда трудоемкость вызова процедуры или функции определяется как количество слов, помещаемых в стек:

– адрес возврата (1 операция);

– состояние необходимых регистров процессора (r операций);

– адреса возвращаемых значений (k операций);

– передаваемые параметры (m операций).

После этого выполняется переход по адресу на вызываемую процедуру, в которой:

– извлекаются переданные фактические параметры (m операций);

– выполняются вычисления (трудоемкость учитывается при анализе самой рекурсивной процедуры), результаты которых

сохраняются по адресам возвращаемых значений, извлеченных из стека (*k* операций);

– из стека извлекаются состояния регистров (*r* операций) и происходит восстановление регистров (операцию восстановления не будем учитывать в силу высокой скорости операций с регистрами);

– из стека извлекается адрес возврата (1 операция) и осуществляет переход по этому адресу.

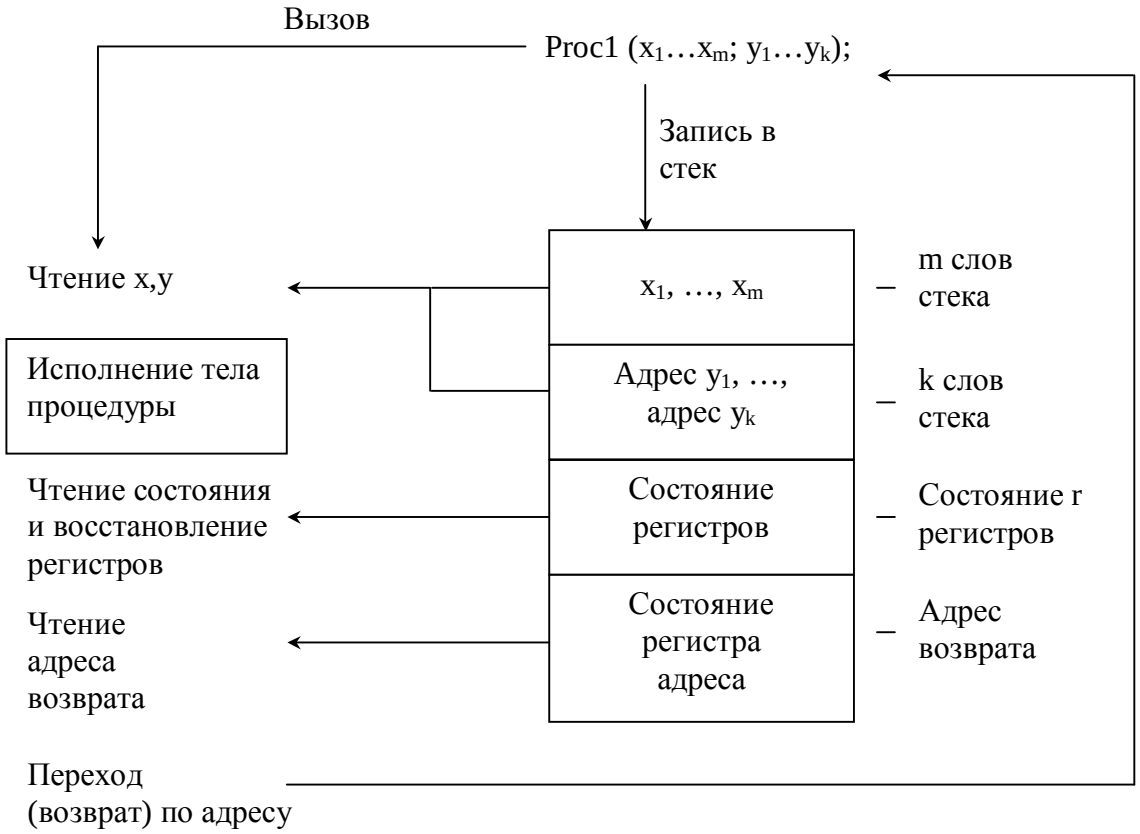


Рисунок 4.3 – Механизм организации вызова процедуры

Необходимо отметить, что в приведенной схеме вызова процедуры не учитываются различия в механизмах вызова процедур и функций. При вызове функции адрес возвращаемого значения в стек не помещается. В стеке резервируется место для возвращаемого функцией значения объявленного типа (будем считать одной элементарной операцией). По результатам выполнения вычислений в стек записывается результат (1 операция). При завершении работы функции вершина стека указывает на это значение, и оно

присваивается переменной, вызвавшей функцию (операция учитывается при анализе вызывающего алгоритма).

Следовательно, реализация вызова функции приводит к двум элементарным операциям на одно возвращаемое значение, что соответствует количеству операций на одно значение при передаче в процедуру адресов возвращаемых значений. Таким образом, учитывая уровень абстракции псевдокода (описанный алгоритм может быть легко реализован как в виде процедуры, так и в виде функции), можно не делать различий между способом возвращения значений.

На основании изложенного выше можно определить количество элементарных операций на один вызов и возврат:

$$f_{вызова} = 2(m + k + r + 1).$$

Для рассмотренного выше рекурсивного алгоритма вычисления $n!$ количество вершин рекурсивного дерева, то есть вызовов, равно n, при этом передается и возвращается по одному значению (m=1, k=1). Количество сохраняемых регистров зависит от вида вызова, количества регистров, изменяемых подпрограммой в процессе работы, и используемых языка программирования и компилятора. Во избежание усложнения процедуры анализа в части рассмотрения системных особенностей организации вызова будем в дальнейшем рассматривать простейший случай сохранения четырех регистров (BP, SP, SS, DS) r=4. Тогда получим:

$$f_{вызовов}(n) = n \cdot f_{вызова} = n \cdot 2 \cdot (1+1+4+1) = 14n,$$

где n – параметр алгоритма (а не количество слов на входе).

Также рассмотрим на этом примере простейший случай анализа дерева рекурсии для определения функции трудоемкости самих рекурсивных вызовов. В данном случае дерево рекурсии вырождается в линейный список. Всего делается n вызовов рекурсивной процедуры F (в приведенной выше схеме n=5). Из них n-1 вызовов приводят к новым рекурсивным вызовам (выполняются операторы секции else). Здесь можно выделить 1 операцию проверки условия и 3

элементарных операции, затраченных на подготовку к рекурсивному вызову и обработку возвращенных результатов (уменьшение значения n, умножение на n и присваивание полученного значения F). Один рекурсивный вызов приводит к возможности нерекурсивного решения, которое включает две элементарных операции (проверка условия и присваивание). Таким образом, получим:

$$f_A(n) = (n-1)\cdot(1+3)+1\cdot 2 = 4n-2.$$

Теперь можем определить полную функцию трудоемкости рекурсивного алгоритма вычисления факториала:

$$F_A(n) = 14n + 4n - 2 = 18n - 2.$$

4.2.2 Построение рекурсивной функции трудоемкости на основе анализа механизма декомпозиции задачи

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

В общем случае, если происходит разделение некоторой задачи с исходным набором данных размера n, которое приводит к необходимости решения a подзадач размерностью n/b, то общий вид функции трудоемкости можно представить следующим образом:

$$f_A(n) = a\cdot f_A(n/b) + d(n) + c(n),$$

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

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

Таким образом, трудоемкость рекурсивного алгоритма зависит:

– от количества рекурсивных вызовов и размерности входных наборов данных;

– трудоемкости подготовительных действий;

– трудоемкости завершающих действий.

Также необходимо учитывать трудоемкость нерекурсивного решения задачи. Очевидно, что приведенная выше зависимость

является рекурсивной и выражает трудоемкость общего случая. При достижении некоторого значения *n*, при котором возможно нерекурсивное решение, имеет место базовый случай функции. Он дает трудоемкость нерекурсивного решения задачи на соответствующем наборе входных данных.

Рекурсивный алгоритм *DAC* (Divide And Conquer – разделяй и властвуй) в общем виде можно представить следующим образом:

```
DAC (data, N; solution)
    if (N <= SizeLimit) then
        DirectSolution (data, N; solution)
    else
        DivideInput (data, N; smalSets[], smalSizes[], nSmal)
        for i ¬ 1 to nSmal do
            DAC (smalSets[i], smalSizes[i]; smalSol[i] )
        end for
        CombineSolutions (smalSol[], nSmal; solution)
    end if
end DAC
```

При описании алгоритма использованы следующие обозначения:

data – набор входных данных;

N – количество значений в наборе;

solution – решение задачи;

SizeLimit – размер задачи, при котором возможно прямое решение;

DirectSolution – нерекурсивный алгоритм прямого решения;

DivideInput – процедура, разбивающая набор входных данных на несколько меньших наборов;

smalSets – список меньших наборов данных;

smalSizes – список размеров меньших наборов данных;

nSmal – количество меньших наборов данных;

smalSol – список решений подзадач на меньших наборах данных;

CombineSolutions – процедура сведения решений подзадач в решение задачи.

Сначала осуществляется проверка, не мал ли размер задачи настолько, чтобы решение можно было найти с помощью простого нерекурсивного алгоритма *DirectSolution*, и если это так, то происходит его вызов. Если задача слишком велика, то сначала вызывается процедура *DivideInput*, которая разбивает входные данные на несколько меньших наборов. Затем происходит рекурсивный вызов алгоритма DAC на каждом из меньших входных множеств, а функция *CombineSolutions* сводит полученные результаты воедино.

Если известно, как шаги приведенного абстрактного и реального алгоритмов сочетаются друг с другом, и известна трудоемкость каждого шага, то для определения трудоемкости рекурсивного алгоритма можно воспользоваться следующей зависимостью:

$$f_A(N) = \begin{cases} f_{DIR}(N), & \text{при } N \le SizeLimit \\ \sum_{i=1}^{nSmal} f_A(smalSizes[i]) + f_{DIV}(N) + f_{COM}(N), & \text{при } N > SizeLimit \end{cases},$$

где f_{DIR}, f_{DIV}, f_{COM} – функции трудоемкости процедур *DirectSolution*, *DivideInput*, *CombineSolutions* соответственно.

В приведенной зависимости не учтена операция проверки условия достижения размерности задачи, позволяющей получить нерекурсивное решение. Также реальные рекурсивные алгоритмы могут содержать операции вне тела if (как до него, так и после). При необходимости точного вычисления функции трудоемкости учет этих затрат при каждом вызове может быть осуществлен путем их введения в функции трудоемкости алгоритма прямого решения f_{DIR} и алгоритма деления входных данных f_{DIV}.

Рассмотрим приведенный выше алгоритм вычисления факториала:

– прямое решение возможно при *SizeLimit*, равном 1;

– трудоемкость прямого решения f_{DIR} составляет 2 элементарных операции (проверка условия и непосредственно присваивание);

– трудоемкость деления входа на более мелкие части f_{DIV} составляет 2 элементарных операции (проверка условия и уменьшение параметра n на единицу);

– осуществляется один рекурсивный вызов $nSmal$=1 на наборе данных $smalSizes[1]=n-1$;

– трудоемкость объединения результатов f_{COM} – две операции (умножение и присваивание).

Отсюда получим рекурсивную функцию трудоемкости алгоритма вычисления факториала:

$$f_A(n) = \begin{cases} 2, & \text{при } n \le 1 \\ f_A(n-1) + 2 + 2, & \text{при } n > 1 \end{cases}.$$

Нетрудно убедиться, что полученная рекуррентная зависимость соответствует выражению для $f_A(n)$, полученному в результате анализа цепочки рекурсивных вызовов и возвратов.

При получении рекурсивной функции трудоемкости можно включить в нее трудоемкость механизма вызова. Для этого необходимо учитывать трудоемкость механизма вызова по одному разу для вызова, приводящего к нерекурсивному решению, и для вызова, приводящего к рекурсии:

$$F_A(N) = \begin{cases} f_{DIR}(N) + f_{вызова}, & \text{при } N \le SizeLimit \\ \sum_{i=1}^{nSmal} F_A(smalSizes[i]) + f_{DIV}(N) + f_{COM}(N) + f_{вызова}, & \text{при } N > SizeLimit \end{cases}$$

Для алгоритма вычисления факториала ранее было определено, что трудоемкость одного вызова и возврата составляет 14 элементарных операций. На основании приведенной выше зависимости получим рекуррентное выражение полной трудоемкости алгоритма:

$$F_A(n) = \begin{cases} 2+14, & \text{при } n \le 1 \\ F_A(n-1)+2+2+14, & \text{при } n > 1 \end{cases},$$

которое соответствует результату, полученному при анализе дерева рекурсии.

Получаемые в процессе анализа рекуррентные зависимости могут быть более сложными и их интерпретация затруднена. Тогда необходимо сводить рекуррентные зависимости к так называемому замкнутому виду. Для этого разработан ряд математических методов, которые будут рассмотрены ниже, так как они представляют больший интерес при анализе сложности рекурсивных алгоритмов (только один из этих методов позволяет получить точный вид функции, остальные дают асимптотические оценки).

4.2.3 Анализ дерева рекурсивных вызовов

Анализ трудоемкости рекурсивного алгоритма основан на определении трудоемкости механизма вызова, трудоемкости самого вызова и количества таких вызовов. Анализ дерева рекурсии позволяет определить количество вызовов, причем как приводящих к рекурсии, так и приводящих к нерекурсивному решению.

Процедура анализа в общем случае выглядит следующим образом:

1. Строится дерево рекурсивных вызовов.

2. Определяется количество вершин (вызовов) на каждом из уровней.

3. Определяется трудоемкость вызовов на каждом из уровней.

4. Общая трудоемкость алгоритма определяется как сумма произведений количества вершин на трудоемкость вызовов по всем уровням:

$$F_A(N) = \sum_{i=1}^{h} f_{Ai} \cdot k_i,$$

где N – размер набора входных данных;

h – высота дерева;

f_{Ai} – трудоемкость вызова на i-ом уровне дерева;

k_i – количество узлов (вызовов) на i-ом уровне дерева.

Обычно функция трудоемкости вызова на уровне листьев, соответствующего прямому решению, имеет вид, отличный от функции трудоемкости в вызовах внутренних узлов. В этом случае вклад уровня листьев учитывается отдельно:

$$F_A(N) = \sum_{i=1}^{h-1} f_{Ai} \cdot k_i + f_{Аллис} \cdot k_{лист} \qquad (*)$$

где $f_{Алист}$ – трудоемкость прямого решения;

$k_{лист}$ – количество узлов на уровне листьев (нерекурсивных вызовов).

Нередко вклад всех внутренних узлов одинаков. Тогда можно воспользоваться упрощенной зависимостью:

$$F_A(N) = f_{Аввнут} \cdot k_{внутр} + f_{Аллис} \cdot k_{лист}, \qquad (**)$$

где $f_{Авнутр}$ – трудоемкость вызовов во внутренних узлах;

$k_{внутр}$ – количество внутренних узлов (рекурсивных вызовов).

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

Алгоритм сортировки прямым слиянием

Рекурсивная реализация алгоритма сортировки прямым слиянием подразумевает разбиение входного набора данных на две части (цепочка рекурсивных вызовов) до достижения наборов данных единичной длины, которые по умолчанию являются упорядоченными, и слияние их в упорядоченные двойки, четверки и т.д. (цепочка рекурсивных возвратов).

Рекурсивная процедура *MS* (Merge Sort) сортирует массив *list* и получает на вход два индекса *l* и *r*, указывающие на ту часть массива, которая будет обрабатываться в данном вызове:

```
MS (l, r)
  if l<r then
      m ¬ (l+r) div 2 //деление входа на 2 части
      MS(l, m)        //рекурсивный вызов для первой части
```

```
        MS(m+1, r)          //рекурсивный вызов для второй части
        Merge(l, m, r)      //слияние отсортированных частей
    end if
end MS
```

Анализ процедуры слияния отсортированных частей Merge

Данный этап является подготовительным и базируется исключительно на принципах анализа итеративных алгоритмов.

Классическая схема слияния двух отрезков выглядит следующим образом. Производят сравнение начальных элементов каждого из отрезков, меньший элемент выводят, а на его место становится следующий элемент из этой последовательности. Эту операцию повторяют до тех пор, пока один из отрезков не закончится. Оставшиеся элементы другого отрезка выводят, не изменяя порядка.

Трудоемкость вывода очередного элемента по результатм стравнения превышает трудоемкость вывода элемента из оставшейся части отрезка. При этом количество элементов, выведенных из оставшейся части, будет зависить от порядка следования элементов в исходной последовательности, то есть алгоритм слияния является порядково-зависимым. Учет всех возможных классов наборов входных данных не является сложной задачей, но вероятности появления различных наборов определить достаточно сложно. Поэтому остановимся на рассмотрении худшего случая. При этом предположим:

1. Сливаемые на каждом уровне рекурсии отрезки содержат одинаковое количество элементов. Формально это допущение означает, что количество элементов в наборе данных равно 2^k.

2. Все элементы за исключением одного выводятся по результатам проверки условия, то есть в выходной последовательности элементы двух отрезков чередуются. Таким образом, только последний элемент второго отрезка будет выведен в цикле переноса оставшейся части списка.

Merge(s1, e1, e2)

s1 – начало первого отрезка

e1, e2 – конец первого и второго отрезков соответственно

s2 ¬ e1+1 //начало второго отрезка	2
fS ¬ s1 //начало сливаемых отрезков	1
fE ¬ e2 //конец сливаемых отрезков	1
iR ¬ 1 //индекс промежуточного массива	1
//слияние до завершения одного из отрезков	
while (s1<=e1) and (s2<=e2) do	$k \cdot 3$
if list[s1]<list[s2] then	$(k-1) \cdot 3$
result[iR] ¬ list[s1]	$(k/2) \cdot 3$
s1 ¬ s1+1	$(k/2) \cdot 2$
else	
result[iR] ¬ list[s2]	$(k/2-1) \cdot 3$
s2 ¬ s2+1	$(k/2-1) \cdot 2$
end if	
iR ¬ iR+1	$(k-1) \cdot 2$
end while	
//перенос оставшейся части отрезка	
if (s1<=e1) then	1
for i ¬ s1 to e1 do	
result[iR] ¬ list[i]	
iR ¬ iR+1	
end for	
else	
for i ¬ s2 to e2 do	$1+1+3 \cdot 1$
result[iR] ¬ list[i]	$3 \cdot 1$
iR ¬ iR+1	$2 \cdot 1$
end for	
end if	

```
                //возвращение результата
                //в исходную последовательность
  iR ¬ 1                                          1
  for i ¬ fS to fE do                             1+1+k·3
      list[i] ¬ result[iR]                        k·3
      iR ¬ iR+1                                   k·2
  end for
end Merge
```

Рассмотрим основные блоки алгоритма слияния:

1. В худшем случае цикл слияния while выполнится k–1 раз, где k – количество элементов во входном наборе данных. Тогда условие цикла выполнится k раз (одна проверка условия при выходе из цикла). В цикле слияния будут выведены все элементы первого отрезка (секция then выполнится $k/2$ раз) и все элементы второго отрезка за исключением последнего (секция else выполнится $k/2–1$ раз).

2. При переносе оставшейся части один раз выполнится цикл for в секции else (во втором отрезке остался один элемент).

3. Возвращение результатов из промежуточного массива во входную последовательность потребует k итераций цикла for.

На основании указанного справа от алгоритма количества операций можно получить трудоемкость процедуры слияния отрезков в худшем случае:

$$F_m^{\wedge}(k) = 2+1+1+1+k\cdot3+(k-1)\cdot3+(k/2)\cdot5+(k/2-1)\cdot5+(k-1)\cdot2+1+$$
$$+1+1+3+3+2+1+1+1+k\cdot3+ k\cdot3+ k\cdot2=21k+9.$$

Построение дерева рекурсии

Исходя из сделанных выше предположений, рассмотрим случай, когда количество элементов в исходном списке $n = 2^d$.

Первый уровень дерева образован начальным вызовом процедуры *MS* на наборе данных длины n (рисунок 4.4). Второй

уровень дерева образуют два рекурсивных вызова на наборах данных длины $n/2$, и т.д. На каждом уровне алгоритм делит входной набор данных на две части. Лист соответствует вызову процедуры *MS* на наборе данных длины 1, что приводит к останову рекурсии. Следовательно, глубина уровня листьев $d = \log_2 n$, а количество уровней, то есть высота дерева $h=d+1$. В этом случае мы имеем полное бинарное дерево, содержащее $k_{лист} = 2^d = n$ листьев. Основываясь на свойстве полного бинарного дерева (число внутренних вершин на единицу меньше числа листьев), получим количество внутренних узлов дерева $k_{внутр} = n-1$.

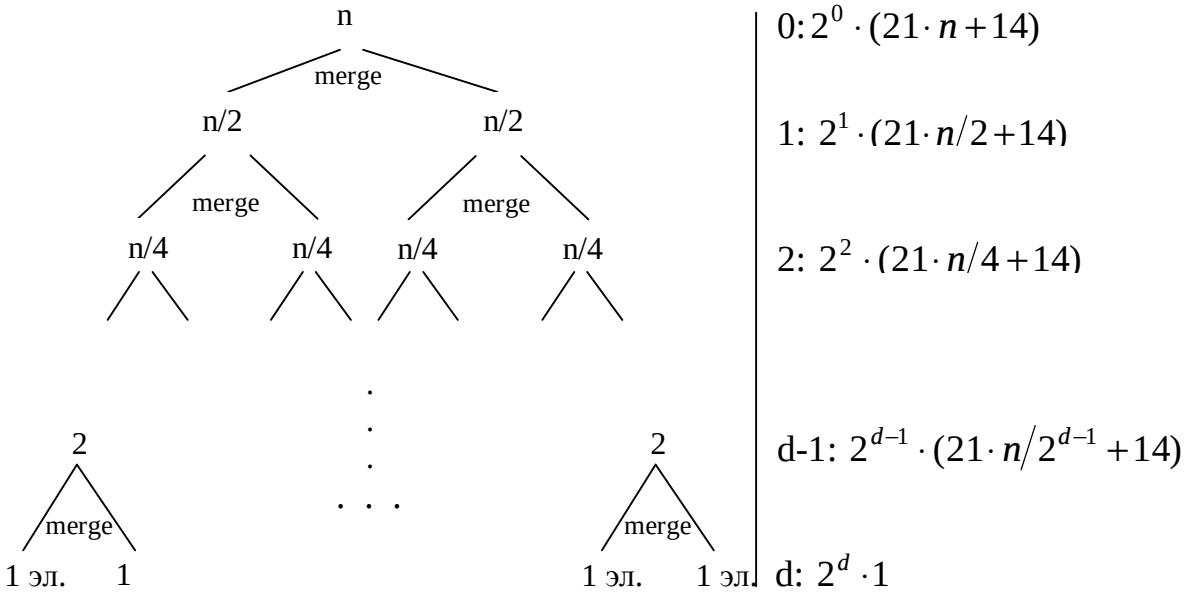


Рисунок 4.4 – Дерево рекурсивных вызовов процедуры сортировки слиянием

Анализ дерева рекурсивных вызовов

Общая трудоемкость алгоритма в худшем случае будет складываться из трудоемкости непосредственно процедуры $F^{\wedge}_{MS}(n)$ и трудоемкости механизма вызовов $F_{вызовов}(n)$:

$$F^{\wedge}_{A}(n) = F^{\wedge}_{MS}(n) + F_{вызовов}(n)$$

Для нахождения $F^{\wedge}_{MS}(n)$ выпишем функции трудоемкости для внутренних узлов и листьев. Во внутреннем узле выполняется проверка длины переданного массива (1 операция), вычисление середины массива (3 операции), вычисление индекса начала второй части m–1 (1 операция) и вызов процедуры *Merge*, трудоемкость которой $F^{\wedge}_{m}(k)$ на некотором уровне i определена выше:

$$f_{Ai} = (1+3+1)+(21k+9) = 21k+14.$$

Данные операции выполняются с различными длинами массива k: n, $n/2$, $n/4$, …, 2, причем 1 раз с длиной n на уровне корня (глубина 0), 2 раза с длиной $n/2$ на втором уровне (глубина 1), 4 раза с длиной $n/4$ на третьем уровне (глубина 2), и т.д., $n/2 = 2^{d-1}$ раз с длиной $2 = n/2^{d-1}$ на предпоследнем уровне (глубина d-1).

Таким образом, вклад операций вызова во внутреннем узле на уровне i (см. подписи справа на рисунке 4.4)

$$f_{Ai} = 21n/2^{i} + 14 \ (i=0..d\text{-}1),$$

а количество узлов на этом уровне

$$k_i = 2^{i} \ (i=0..d\text{-}1).$$

На уровне листьев содержится $k_{лист} = n$ (определено выше) узлов и при каждом вызове осуществляется лишь одна проверка условия $f_{Алист}$=1.

Подставляя найденные значения в формулу (*), получим:

$$F^{\wedge}_{MS}(n) = \sum_{i=0}^{d-1} 2^{i} \cdot (21 \cdot n/2^{i} + 14) + 1^{*} n = 21\sum_{i=0}^{d-1} n + 14\sum_{i=0}^{d-1} 2^{i} + n =$$

$$= 21n \cdot d + 14(2^{d} - 1) + n;$$

$$F^{\wedge}_{MS}(n) = 21n\log_2 n + 15n - 14.$$

Теперь определим вклад трудоемкости механизма вызовов процедур $F_{вызовов}(n)$. Он будет складываться из трудоемкостей процедур *MS* и *Merge* во внутренних узлах и трудоемкости вызова процедуры *MS* в листьях.

Трудоемкость вызова процедуры *MS* равна (два параметра, четыре регистра и адрес возврата): 2(2+4+1), а процедуры *Merge* (три

параметра, четыре регистра и адрес возврата): 2(3+4+1). Количество внутренних узлов $k_{внутр} = n–1$, листьев $k_{лист} = n$. В данном случае будет использовать формулу (**), тогда получим:

$$F_{вызовов}(n) = (2(2+4+1)+2(3+4+1))\cdot(n-1)+2(2+4+1)\cdot n = 44n-30.$$

Сводя воедино результаты, получим:

$$F_A^{\wedge}(n) = (21n\log_2 n+15n-14)+(44n-30) = 21n\log_2 n+59n-44.$$

Если количество чисел на входе алгоритма не равно степени двойки, то необходимо проводить более глубокий анализ, основанный на изучении неполного рекурсивного дерева, однако при любых наборах данных оценка главного порядка $\Theta(n\log_2 n)$ не измениться.

4.3 Анализ сложности рекурсивных алгоритмов

Анализ сложности рекурсивных алгоритмов основывается на тех же методах, что и анализ трудоемкости. Однако, так же, как и при анализе сложности итеративных алгоритмов, специфика асимптотических оценок позволяет делать некоторые допущения, упрощающие процедуру анализа. В основном это выражается в возможности учета только значащих операций и в возможности применения большего спектра методов решения рекуррентных соотношений, позволяющих получать только асимптотические оценки. Также необходимо отметить возможность не учитывать округление получаемых в процессе анализа дробей до ближайшего большего или меньшего целого. Например, рекурсивная функция $T(n) = 2T(\lfloor n/2 \rfloor)+5$ асимптотически растет так же быстро, как $T(n) = 2T(n/2)+5$.

Получение асимптотической оценки трудоемкости алгоритма сортировки слиянием путем анализа механизма декомпозиции задачи.

При анализе алгоритмов сортировки принято в качестве значащих операций выбирать либо операции проверки условия, по результатам которых принимается решение о перемещении

элементов (порядке вывода элементов в данном случае), либо операции перемещения элементов (копирования). В данном случае рассмотрим анализ сложности по операциям проверки условия, так как он более показателен с точки зрения получаемых результатов и их интерпретации.

1) Анализ сложности процедуры слияния. При анализе сложности откажемся от принятого в процессе анализа трудоемкости предположения о строгом чередовании элементов при их выводе в выходную последовательность и определим сложность алгоритма в худшем и лучшем случаях. Обозначим входные списки через A и B, а количество элементов входных списков – n_A и n_B соответственно.

В случае, когда все элементы списка A меньше всех элементов списка B, алгоритм выполняет n_A сравнений. Если все элементы списка B оказываются меньше всех элементов списка A, то число сравнений равно n_B.

Если первый элемент списка A больше первого элемента списка B, но все элементы списка A меньше второго элемента списка B, тогда будет сделано не только n_A сравнений элементов списка A с $B[2]$, но и сравнение $A[1]$ *с* $B[1]$, то есть n_A+1. Таким образом, первая описанная ситуация является лучшим случаем, а каждое чередование элементов из A и B в выходной последовательности добавляет одно сравнение.

Если значения элементов списков A и B идут «через один», по результатам сравнения переносятся все элементы кроме одного, и число сравнений в этом худшем случае равно $n_A + n_B - 1$.

Предположим (так же, как и при анализе трудоемкости), что начальное число элементов списка является степенью двойки. Тогда при каждом вызове список длиной n разбивается на два подсписка длиной $n/2$. Это означает, что на этапе слияния потребуется в лучшем случае $n_A = n_B = n/2$ сравнений, а в худшем число сравнений будет равно $n_A + n_B - 1 = n/2 + n/2 - 1 = n - 1$.

2) Анализ алгоритма сортировки слиянием. Сопоставим приведенный выше алгоритм сортировки слиянием с типовым рекурсивным алгоритмом:

– процедура *MS* вызывается рекурсивно, пока длина рассматриваемого списка превышает единицу *SizeLimit*=1;

– нерекурсивное решение не требует операций проверки условия $f_{DIR}=0$;

– разбиение списка на две части происходит при вычислении значения переменной *m*, что также не требует сравнений $f_{DIV}=0$;

– организуется два рекурсивных вызова *nSmal* = 2 процедуры *MS* на наборах данных длины *smalSizes*[1] = *smalSizes*[2] = *n*/2;

– объединение решений осуществляется в процедуре слияния, сложности которой была в лучшем $f^{\vee}_{COM}=n/2$ и худшем $f^{\wedge}_{COM}=n-1$ случаях были определены выше.

Отсюда получим рекурсивные функции сложности в худшем (*W*) и лучшем (*B*) случаях соответственно:

$$W(n) = 2W(n/2)+n\text{-}1 \text{ при } n>1,$$
$$W(0) = W(1) = 0;$$

$$B(n) = 2B(n/2) + n/2 \text{ при } n>1,$$
$$B(0) = B(1) = 0.$$

Сведение этих рекуррентных зависимостей к замкнутому виду методом итераций (будет рассмотрен в следующем пункте) дает:

$$W(n) = n\log_2 n - n + 1,$$

$$B(n) = \frac{n}{2}\log_2 n.$$

Очевидно, что асимптотически сложность в худшем и лучшем случаях равны: $W(n)=\Theta(n\log_2 n)$ и $B(n)=\Theta(n\log_2 n)$. Это означает, что алгоритм сортировки слиянием асимптотически ведет себя одинаково на всех наборах данных. Отсюда можно сделать вывод о сложности в среднем случае: $A(n)=\Theta(n\log_2 n)$.

Анализ сложности алгоритма сортировки слиянием по количеству операций копирования даст такой же результат, причем оценки, полученные для худшего и лучшего случаев, совпадают, так как при каждом вызове производится одинаковое количество операций копирования.

Определение сложности алгоритма сортировки слиянием путем анализа дерева рекурсивных вызовов.

Решим ту же задачу на основе анализа приведенного выше дерева рекурсивных вызовов алгоритма сортировки слиянием.

В листьях осуществляется нерекурсивное решение, которое не требует операций проверки условия. В каждом внутреннем узле (их количество равно n-1) вызывается процедура *Merge*.

В худшем случае сложность процедуры *Merge* равна n-1. Рассмотрим дерево рекурсивных вызовов, построенное при анализе трудоемкости. На первом уровне процедура вызывается 1 раз с длиной n на уровне корня (что дает $n-1$ операций), 2 раза с длиной $n/2$ на втором уровне ($2\cdot(n/2-1)$ операций), 4 раза с длиной $n/4$ на третьем уровне ($4\cdot(n/4-1)$ операций), и т.д., $n/2=2^{d-1}$ раз с длиной $2=n/2^{d-1}$ на предпоследнем уровне ($2^{d-1}\cdot(n/2^{d-1}-1)$ операций).

Отсюда получим сложность в худшем случае:

$$W(n)=\sum_{i=0}^{d-1}2^i\cdot(n/2^i-1)=\sum_{i=0}^{d-1}n-\sum_{i=0}^{d-1}2^i=n\cdot d-(2^d-1),$$

$$W(n)=n\log_2 n-n+1.$$

Путем аналогичных рассуждений для лучшего случая получим:

$$B(n)=\sum_{i=0}^{d-1}2^i\cdot\frac{n/2}{2^i}=\frac{n}{2}\cdot d,$$

$$B(n)=\frac{n}{2}\log_2 n.$$

4.4 Решение рекуррентных соотношений

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

Рекуррентное соотношение в общем виде можно представить следующим образом:

$$T(n) = \begin{cases} g(n, T(m)) & \text{для } n > n_0 \\ f(n) & \text{для } n \le n_0 \end{cases},$$

где $m<n$ – аргумент функции на следующем шаге рекурсии.

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

Основной задачей исследования рекурсивно заданных функций является получение $T(n)$ в явной или, как еще говорят, «замкнутой» форме, т.е. в виде аналитически заданной функции.

Наиболее простым методом сведения рекуррентных соотношений к замкнутому виду является использование общих решений для определенного вида функций. Однако разработан ряд математических методов, которые будут рассмотрены далее.

4.4.1 Метод итераций

Метод итераций, в отличие от других рассматриваемых методов, позволяет получить точный вид аналитически заданной функции, соответствующей рекурсивной функции.

Метод основан на последовательной подстановке в правую часть рекуррентного соотношения выражений для $T(m)$, $m<n$. Подстановки осуществляются с целью исключить из правой части все выражения $T(m)$ для $m>n_0$, оставляя только $T(1)\ldots T(n_0)$. Поскольку $T(1)\ldots T(n_0)$ всегда являются константами или функциями от n, то в результате получают аналитически заданную функцию $T(n)$.

В процессе решения рекуррентного соотношения методом итераций можно выделить следующие этапы:

1. Последовательное получение нескольких (обычно небольшое количество) выражений для $T(m)$, эквивалентных исходному.

2. Последовательная подстановка в правую часть исходной функции эквивалентных выражений для $T(m)$. При этом важно не упрощать результат до конца.

3. Анализ полученного выражения с целью выявления «шага» рекурсии и составление замкнутого выражения, включающего $T(1)\ldots T(n_0)$ и функцию от n (обычно в виде суммы).

Рассмотрим метод итераций на примере следующего рекуррентного соотношения:

$T(n) = 2T(n–2)–15;$

$T(2) = 40;$

$T(1) = 40.$

Из первого равенства получим эквивалентное выражение для $T(n–2)$. Для этого каждое вхождение n в это уравнение заменяется на $n–2$:

$T(n–2) = 2T(n–2–2)–15 = 2T(n–4)–15.$

Аналогично получим $T(n–4)$ и т.д.:

$T(n–4) = 2T(n–6)–15;$

$T(n–6) = 2T(n–8)–15;$

$T(n–8) = 2T(n–10)–15;$

$T(n–10) = 2T(n–12)–15.$

Результаты вычислений подставляем обратно в исходное уравнение. При этом результат не упрощается до конца.

$T(n) = 2T(n–2)–15 = 2(2T(n–4)–15)–15,$

$T(n) = 4T(n–4)–2\cdot15–15;$

$T(n) = 4(2T(n–6)–15)–2\cdot15–15,$

$T(n) = 8T(n–6)–4\cdot15–2\cdot15–15;$

$T(n) = 8(2T(n–8)–15)–4·15–2·15–15,$
$T(n) = 16T(n–8)–8·15–4·15–2·15–15;$

$T(n) = 16(2T(n–10)–15)–8·15–4·15–2·15–15,$
$T(n) = 32T(n–10)–16·15–8·15–4·15–2·15–15;$

$T(n) = 32(2T(n–12)–15)–16·15–8·15–4·15–2·15–15,$
$T(n) = 64T(n–12)–32·15–16·15–8·15–4·15–2·15–15.$

Рассмотрим полученное выражение. Во-первых, слагаемые с конца в каждом равенстве представляют собой число –15, умноженное на очередную степень двойки. Во-вторых, коэффициент при рекурсивном вызове функции T является степенью двойки. При этом аргумент функции T всякий раз уменьшается на 2.

Этот процесс завершится при достижении одного из граничных условий. При четном n граничным условием является 2. То есть аргумент функции меняется от $n - 2$ до 2 с шагом 2. Отсюда количество подстановок равно $\frac{(n-2)-2}{2} = \frac{n}{2} - 2$, что дает $\frac{n}{2} - 2 + 1 = \frac{n}{2} - 1$ слагаемых с множителем –15 («+1» – одно слагаемое присутствует в исходном выражении) и $\frac{n}{2} - 1$ – степень двойки перед T.

Например, рассмотрим случай n=14. В этом случае, согласно предыдущему предложению, осуществляется пять подстановок, имеется шесть слагаемых с множителем –15, и коэффициент при $T(2)$ будет равен 2^6. Именно такой результат получается при подстановке n=14 в последнее равенство.

При нечетном n граничным условием будет являться равенство аргумента единице. Аргумент функции меняется от $n - 2$ до 1 с

шагом 2. Отсюда количество подстановок равно $\frac{(n-2)-1}{2} = \frac{n-1}{2} - 1 = \lfloor n/2 \rfloor - 1$, что дает $\lfloor n/2 \rfloor$ слагаемых с множителем –15 и $\lfloor n/2 \rfloor$ – степень двойки перед T. Очевидно, что подстановка n=13 дает корректные результаты.

В ответе рассмотрим два случая (для четного и нечетного n):

$$T(n) = 2^{n/2-1} T(2) - 15 \sum_{i=0}^{n/2-2} 2^i \text{ при четном } n;$$

$$T(n) = 2^{\lfloor n/2 \rfloor} T(1) - 15 \sum_{i=0}^{\lfloor n/2 \rfloor - 1} 2^i \text{ при нечетном } n.$$

Раскрывая суммы и приводя подобные, получим:
для четного n

$$T(n) = 2^{n/2-1} \cdot 40 - 15 \cdot (2^{n/2-1} - 1) =$$
$$= 2^{n/2} \cdot 20 - 7{,}5 \cdot 2^{n/2} + 15 =$$
$$= 2^{n/2} \cdot 12{,}5 + 15;$$

для нечетного n

$$T(n) = 2^{\lfloor n/2 \rfloor} \cdot 40 - 15 \cdot (2^{\lfloor n/2 \rfloor} - 1) =$$
$$= 2^{\lfloor n/2 \rfloor} \cdot 25 + 15.$$

Если учесть, что для нечетного n справедливо равенство $\lfloor n/2 \rfloor = \frac{n-1}{2}$, то для этого случая результат можно представить в следующем виде:

$$T(n) = 2^{n/2} \cdot \frac{25}{\sqrt{2}} + 15.$$

При рассмотрении полученных выражений можно сделать вывод о том, что в случае асимптотического анализа при равных скоростях роста сложности базовых случаев достаточно получить выражение только для одного из них, так как оценки других будут отличаться не более чем на константу.

Решение рекуррентных соотношений методом анализа дерева рекурсии.

Процесс подстановки соотношения в себя можно изобразить в виде дерева рекурсии. Тогда процедура получения выражения в замкнутом виде будет аналогична процедуре получения функции трудоемкости на основе анализа дерева рекурсивных вызовов. Дерево рекурсивных подстановок строится следующим образом:

1. На первой итерации формируется дерево следующего вида:

– в корень дерева записывается свободный член рекуррентного уравнения;

– его потомками являются выражения рекурсивных функций правой части исходного соотношения.

2. На последующих итерациях для каждого из потомков строится аналогичная древовидная структура.

Проиллюстрируем процесс анализа на примере той же рекуррентной зависимости. На рисунке 4.5 приведены деревья рекурсии для исходного выражения, первой и последней подстановок (для случая четного n).

Первый уровень дерева образован свободным членом исходного рекуррентного выражения. Уровень листьев образован базовым случаем. Остальные уровни дерева (со второго по h–1, где h – высота дерева) соответствуют последовательности сделанных подстановок. Тогда высота дерева определяется как количество подстановок (определено выше), увеличенное на 2, и равна $n/2$.

Дерево является полным бинарным, поэтому количество листьев определяется, как $2^h/2 = 2^{n/2-1}$. Каждый лист дает слагаемое 40. Количество узлов на 1 уровне равно 1, на втором – 2, на предпоследнем – $2^{n/2-2}$. Каждый внутренний узел дает слагаемое –15. Суммируя вклады листьев и внутренних узлов, имеем:

$$T(n) = 40 \cdot 2^{n/2-1} - 15 \sum_{i=0}^{n/2-2} 2^i ,$$

что соответствует выражению для четного n, полученному выше.

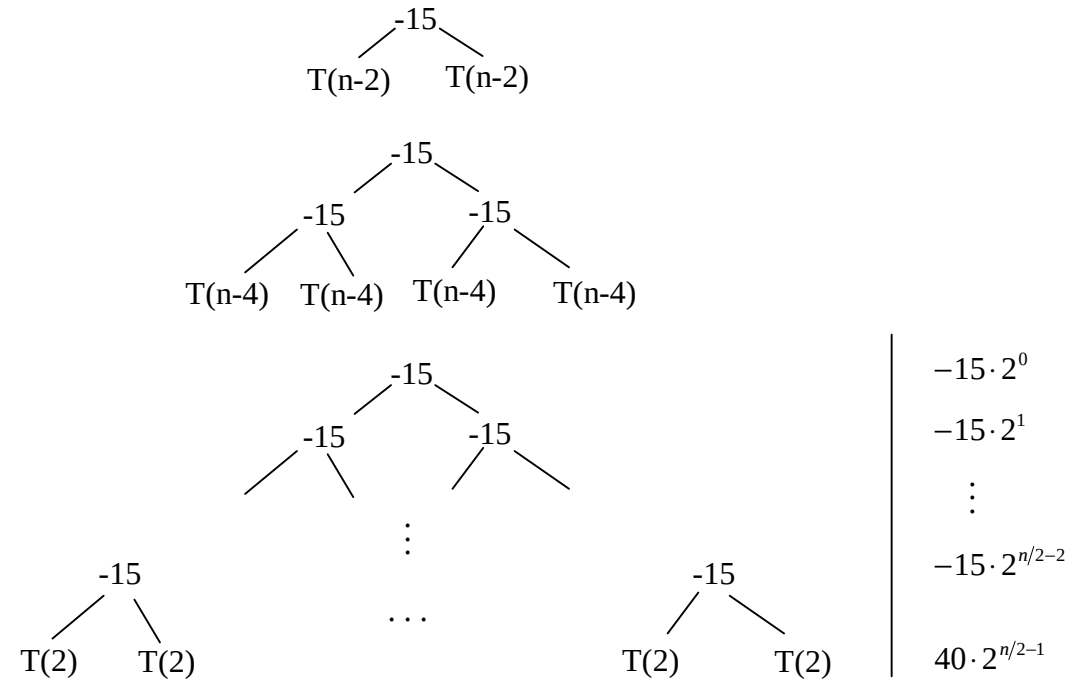


Рисунок 4.5 – Представление процесса подстановки рекуррентного выражения деревом рекурсии

4.4.2 Метод индукции

В общем смысле математическая индукция – это метод доказательства математических утверждений, основанный на принципе: утверждение A(X), зависящее от натурального параметра X, считается доказанным, если:

– доказано A(1) – базис индукции;

– из предложения, что A(n) верно для любого натурального n, доказано, что верно A(n+1).

Метод индукции заключается в том, что вначале необходимо сделать предположение об общем виде решения, а затем доказать верность ответа по индукции. Часто ответ содержит коэффициенты, которые надо выбрать так, чтобы рассуждение по индукции было справедливо. Индуктивный метод применим и к нижним, и к верхним оценкам.

Например, необходимо найти верхнюю оценку для функции, заданной соотношением

$$T(n) = 2T(\lfloor n/2 \rfloor) + n.$$

Предположим, что $T(n) = \mathrm{O}(n \log n)$, то есть, исходя и определения класса O, $T(n) \le c \cdot n \log n$ для некоторого c>0 (здесь и далее, если в логарифме не указывается основание, то подразумевается взятие логарифма по основанию 2).

Предположим, что сделанное предположение верно для $\lfloor n/2 \rfloor$, то есть имеет место оценка $T(\lfloor n/2 \rfloor) \le c \cdot \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$. Подставив ее в исходное рекуррентное соотношение, получим:

$$T(n) \le 2(c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor) + n \le$$

$$\le cn \log(n/2) + n = cn \log(n) - cn \log(2) + n = cn \log(n) - cn + n \le$$

$$\le cn \log(n).$$

Сделанные выше преобразования преследовали цель получить при последнем переходе в левой части неравенства выражение, наиболее близкое к сделанному предположению $cn \log n$. Теперь необходимо определить, существует ли такая константа c, для которой справедлив последний переход. Для этого перепишем его и решим полученное неравенство:

$$cn \log(n) - cn + n \le cn \log(n);$$

$$cn \ge n;$$

$$c \ge 1.$$

Таким образом, сделанное предположение верно для $c \ge 1$.

Теперь необходимо проверить базис индукции, т. е. доказать оценку для начального значения n. При n = 1 правая часть неравенства $T(n) \le cn \log n$ обращается в нуль, каким бы ни было c, в то время как $T(n)$ по определению величина положительная для всех n>0. Однако, асимптотическую оценку достаточно доказать для всех n, начиная с некоторого n_0.

Рассмотрим значение n=2, а для больших n будем рассуждать по индукции. В этом случае правая часть исходного предположения

больше нуля (при c>0), и в качестве базиса индукции можно принять n=2.

Таким образом, верхняя оценка заданной рекуррентной зависимости $T(n) = \mathrm{O}(n \log n)$ доказана. Аналогичным образом можно доказать нижнюю оценку $T(n) = \Omega(n \log n)$, откуда следует $T(n) = \Theta(n \log n)$.

Для того чтобы изначально сделать правильное предположение, необходим опыт и определенная доля везения, однако есть несколько способов, позволяющих облегчить этот выбор.

1. Аналогия.

Рассмотрим для примера соотношение $T(n) = 2T(\lfloor n/2 \rfloor + 10) + n$. Оно отличается от рассмотренного выше только добавочным слагаемым 10 в правой части. В силу свойств асимптотических оценок, для больших n справедливо считать разницу между $\lfloor n/2 \rfloor + 10$ и $\lfloor n/2 \rfloor$ несущественной. Тогда можно ожидать, что первичная оценка $T(n) \le cn \log n$ будет справедлива и для второго соотношения и может служить отправной точкой для доказательства по индукции.

2. Последовательные приближения.

Можно начать с доказательства достаточно грубых (заведомо справедливых) оценок, которые легко доказываются, а затем уточнять их. Например, рассмотрим соотношение $T(n) = 2T(n/2) + \Theta(n)$, где $\Theta(n)$ – некоторая линейно растущая функция. Для n>1 есть очевидная нижняя оценка $T(n) = \Omega(n)$, так как даже в случае нерекурсивного решения на первом шаге оценка линейна $\Theta(n)$. Верхняя оценка $T(n) = \mathrm{O}(n^2)$ легко доказывается. Далее можно постепенно сближать их, стремясь получить более точные верхнюю и нижнюю оценки, отличающиеся не более чем на константу. Для рассмотренного примера единственным промежуточным классом между n и n^2 является $n \log n$, поэтому можно попытаться доказать $T(n) = \mathrm{O}(n \log n)$ и $T(n) = \Omega(n \log n)$.

3. Усиление оценки.

Иногда рассуждение по индукции сталкивается с трудностями, хотя начальное предположение сделано верно. Это происходит потому, что доказываемое по индукции утверждение недостаточно сильно. В этом случае может помочь вычитание члена меньшего порядка.

Рассмотрим соотношение

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

Можно предположить, что в этом случае $T(n) = O(n)$, и это действительно так. Попытаемся доказать сделанное предположение – $T(n) \le cn$. Предположим, что оценка верна для $\lfloor n/2 \rfloor$ и $\lceil n/2 \rceil$, то есть $T(\lfloor n/2 \rfloor) \le c \cdot \lfloor n/2 \rfloor$ и $T(\lceil n/2 \rceil) \le c \cdot \lceil n/2 \rceil$. Подставив оценки в исходное рекуррентное соотношение, получим:

$$T(n) \le c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1.$$

Дальнейшим шагом должно быть $cn + 1 \le cn$, однако такое неравенство несправедливо для всех c. То есть утверждение $T(n) \le cn$ недоказуемо. В такой ситуации можно попытаться доказать более слабую оценку, например, $O(n \log n)$. Но в данном случае это неверно, так как первая оценка является справедливой и асимптотически более близкой. Тогда можно попытаться не ослабить, а усилить предположение индукции.

Новая гипотеза: $T(n) \le cn - b$ для некоторых положительных констант b и c. Подстановка в правую часть дает:

$$T(n) \le (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 = cn - 2b + 1 \le cn - b.$$

Последний переход законен для любых $c > 0$ и $b \ge 1$. Теперь необходимо доказать базис индукции. Очевидно, что последний переход справедлив для любого $n > 0$. Однако необходимо также потребовать, чтобы выполнялось условие $cn - 2b + 1 > 0$. Пусть c=1 и b=1, тогда из неравенства получим $n > 0$, то есть базисом индукции является n_0=2.

Таким образом, сделанная оценка верна, и для определенных выше значений констант c и b можно получить:

$$T(n) \le cn - b \Rightarrow T(n) = \mathrm{O}(n).$$

4.4.3 Теорема о рекуррентных оценках

Данная теорема является мощным средством анализа асимптотической сложности рекурсивных алгоритмов. Метод используется для рекуррентных соотношений, которые возникают, когда алгоритм разбивает задачу размера n на a подзадач размера n/b, эти подзадачи решаются рекурсивно каждая за время $T(n/b)$ и результаты объединяются. При этом затраты на разбиение и объединение описываются функцией $f(n)$. В данном случае округление n/b не производится, так как округление в большую и меньшую сторону приводят к одинаковым результатам.

Теорема. Пусть $a \ge 1$ и $b>1$ – константы, $f(n)$ – положительная для $n>n_0$ функция, $T(n) = aT(n/b) + f(n)$ – определенная для положительных n функция, тогда:

1) если $f(n) = \mathrm{O}(n^{\log_b a - e})$ для некоторого $e>0$, то $T(n) = \Theta(n^{\log_b a})$;

2) если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;

3) если $f(n) = \Omega(n^{\log_b a + e})$ для некоторого $e>0$ и если $af(n/b) \le cf(n)$ для некоторой константы $c<1$ и достаточно больших n, то $T(n) = \Theta(f(n))$.

В каждом из трех случаев сравнивается порядок роста функции $f(n)$ с $n^{\log_b a}$. Если одна из этих функций растет быстрее другой, то она и определяет порядок роста $T(n)$ (случаи 1 и 3). Если обе функции одного порядка (случай 2), то появляется дополнительный логарифмический множитель.

В первом случае недостаточно, чтобы $f(n)$ была просто меньше, чем $n^{\log_b a}$ – необходим «зазор» размера n^e для некоторого $e>0$. Точно так же в третьем случае $f(n)$ должна быть больше $n^{\log_b a}$ с запасом e и к тому же удовлетворять условию «регулярности».

Три указанных случая не исчерпывают всех возможностей: может оказаться, например, что функция $f(n)$ растет медленнее, чем $n^{\log_b a}$, но зазор недостаточно велик для того, чтобы воспользоваться первым утверждением теоремы. Аналогичная «щель» есть и между случаями 2 и 3. Также функция может не обладать свойством регулярности. В этих случаях применение теоремы невозможно.

Рассмотрим применение теоремы на примерах (также рассмотрим случай, когда ее применение невозможно).

Пример 1. $T(n) = 9T(n/3) + n$.

В этом случае:

$a = 9$, $b = 3$, $f(n) = n$;

$n^{\log_b a} = n^{\log_3 9} = n^2$;

$f(n) = \Theta(n)$, $n = n^{\log_3 9 - e} = n^{2-e}$, следовательно, $f(n) = \mathrm{O}(n^{\log_b a - e})$ с запасом e=1.

Отсюда по первому утверждению теоремы заключаем, что $T(n) = \Theta(n^2)$.

Пример 2. $T(n) = T(2n/3) + 1$.

$a = 1$, $b = 3/2$, $f(n) = 1$;

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$;

$f(n) = \Theta(1)$, следовательно, $f(n) = \Theta(n^{\log_b a})$.

Отсюда по второму утверждению теоремы заключаем, что $T(n) = \Theta(\log n)$.

Пример 3. $T(n) = 3T(n/4) + n\log n$.

$a = 3$, $b = 4$, $f(n) = n\log n$;

$n^{\log_b a} = n^{\log_4 3} \approx n^{0,793}$;

$f(n) = \Theta(n\log n)$, следовательно, $f(n) = \Omega(n^1)$, то есть $f(n) = \Omega(n^{\log_b a + e})$ с запасом $e \approx 0{,}207$.

Имеет место третий случай, проверим условие регулярности:

$$af(n/b) = \frac{3n}{4}\log(n/4) \le \frac{3n}{4}\log n;$$

$cf(n) = cn\log n$.

Условие $af(n/b) \le cf(n)$ выполняется, если

$$\frac{3n}{4}\log n \le cn\log n,$$

откуда для n>1 получим $c \ge 3/4$.

По третьему утверждению теоремы заключаем, что $T(n) = \Theta(n\log n)$.

Пример 4. $T(n) = 2T(n/2) + n\log n$.

$a = 2, b = 2,\ f(n) = n\log n;$

$n^{\log_b a} = n^{\log_2 2} = n;$

$f(n) = n\log n$ асимптотически больше чем $n^{\log_b a} = n$, но зазор недостаточен, то есть нельзя найти такие положительные e и c, для которых доказуемо $n\log n \ge cn^{1+e}$. Следовательно, для приведенного примера применение теоремы невозможно.

4.5 Вопросы для самоконтроля

1. В чем заключаются отличия анализа трудоемкости рекурсивных и итеративных алгоритмов?
2. Из каких составляющих складывается трудоемкость рекурсивного алгоритма?
3. Какие параметры учитываются при анализе трудоемкости механизма вызова процедуры?
4. Опишите методику получения рекурсивной функции трудоемкости на основе анализа декомпозиции задачи, применяемую при анализе.
5. Опишите методику получения функции трудоемкости на основе анализе дерева рекурсивных вызовов.
6. Каково соотношение между оценками трудоемкости, получаемыми на основе анализа механизма декомпозиции задачи и путем анализа дерева рекурсивных вызовов?
7. Какие допущения принимаются при анализе сложности рекурсивных алгоритмов?

8. Перечислите основные методы приведения рекуррентного соотношения к замкнутому виду. В чем заключается отличие получаемых в каждом из них результатов?

9. Поясните процедуру приведения рекуррентного соотношения к замкнутому виду методом итераций.

10. Поясните сущность представления процесса подстановок деревом рекурсии. Каким образом анализируется такое дерево?

11. Поясните процедуру получения оценок сложности рекуррентных соотношений методом индукции.

12. Какими способами можно упростить выбор начального предположения при доказательстве по индукции?

13. На основании каких положений выделяются три случая теоремы о рекуррентных оценках?

14. В каких случаях применение теоремы о рекуррентных оценках невозможно?

4.6 Задачи

Задача 4.1.

Методом итераций приведите к замкнутому виду рекуррентное соотношение, полученное при анализе лучшего случая алгоритма сортировки слиянием (в разделе теории):

$B(n) = 2B(n/2) + n/2$ при $n>1$,

$B(0) = B(1) = 0$.

Ответ: $B(n) = \frac{n}{2}\log_2 n$.

Задача 4.2.

Решите задачу 4.1, представив процесс подстановки в виде дерева рекурсии.

Ответ: $B(n) = \frac{n}{2}\log_2 n$.

Задача 4.3.

Методом итераций приведите к замкнутому виду рекуррентное соотношение, полученное при анализе худшего случая алгоритма сортировки слиянием (в разделе теории):

$W(n) = 2W(n/2)+n-1$ при $n>1$,

$W(0) = W(1) = 0$.

Ответ: $W(n) = n\log_2 n - n + 1$.

Задача 4.4.

Методом итераций приведите к замкнутому виду следующее рекуррентное соотношение:

$T(n) = 4T(n/2) - 1$ при $n > 4$,

$T(n) = 5$ при $n \le 4$.

Ответ: $T(n) = \frac{7}{24}n^2 + \frac{1}{3}$.

Задача 4.5.

Решите задачу 4.4, представив процесс подстановки в виде дерева рекурсии.

Ответ: $T(n) = \frac{7}{24}n^2 + \frac{1}{3}$.

Задача 4.6.

Методом итераций приведите к замкнутому виду следующее рекуррентное соотношение:

$T(n) = 2T(n/2) + n^2$ при $n > 1$,

$T(n) = n$ при $n \le 1$.

Ответ: $T(n) \approx 3n^2$ для достаточно больших n.

Задача 4.7.

Решите задачу 4.6, представив процесс подстановки в виде дерева рекурсии.

Ответ: $T(n) \approx 3n^2$ для достаточно больших n.

Задача 4.8.

Методом итераций приведите к замкнутому виду следующее рекуррентное соотношение:

$T(n) = 2T(n/2) + \log_2 n$ при $n > 1$,

$T(n) = 1$ при $n \le 1$.

Ответ: $T(n) = 3n - \log_2 n - 2$.

Задача 4.9.

Решите задачу 4.8, представив процесс подстановки в виде дерева рекурсии.

Ответ: $T(n) = 3n - \log_2 n - 2$.

Задача 4.10.

Методом индукции найдите верхнюю оценку скорости роста рекурсивных функций из задач а) 4.1 и б) 4.3.

Задача 4.11.

Методом индукции найдите верхнюю оценку скорости роста следующей рекурсивной функции:

$T(n) = 2T(n-1) + 1$.

Задача 4.12.

Методом индукции найдите верхнюю оценку скорости роста следующей рекурсивной функции:

$T(n) = T(n-1) + \log n$.

Задача 4.13.

Методом индукции найдите верхнюю оценку скорости роста следующей рекурсивной функции:

$T(n) = 3T(n/2) + n$.

Задача 4.14.

Методом индукции найдите верхнюю оценку скорости роста следующей рекурсивной функции:

$T(n) = T(n/2) + n^2$.

Задача 4.15.

Методом индукции найдите верхнюю оценку скорости роста следующей рекурсивной функции:

$T(n) = 3T(n/3) + 12$.

Задача 4.16.

Методом индукции найдите верхнюю оценку скорости роста следующей рекурсивной функции:

$T(n) = 4T(n/2) + n$.

Задача 4.17.

Используя теорему о рекуррентных оценках, определите порядок роста рекурсивных функций из задач а) 4.1 и б) 4.3.

Задача 4.18.

Используя теорему о рекуррентных оценках, определите порядок роста следующих рекурсивных функций:

а) $T(n) = 4T(n/2) + \log n$,

б) $T(n) = T(n/2) + \sqrt{n}$.

Задача 4.19.

Используя теорему о рекуррентных оценках, определите порядок роста следующих рекурсивных функций:

а) $T(n) = 4T(n/2) + n^2/4$,

б) $T(n) = 9T(n/3) + 2n - 3$.

Задача 4.20.

Используя теорему о рекуррентных оценках, определите порядок роста следующих рекурсивных функций:

а) $T(n) = 3T(n/3) + n^{3/2}\log n$,

б) $T(n) = 4T(n/2) + n\log n$.

Задача 4.21.

Используя теорему о рекуррентных оценках, определите порядок роста следующих рекурсивных функций:

а) $T(n) = 4T(n/2) + n^3 - n^2$,

б) $T(n) = 8T(n/2) + n^3/4$.

Задача 4.22.

Абстрактная вычислительная машина для вывода на экран может использовать только команду write (x), где x – десятичные цифры от 0 до 9. Вывод произвольного целого положительного числа n на экран реализуется следующим рекурсивным алгоритмом:

```
Print (n)
    if n<10 then
        write (n)
    else
        print (n div 10)
        write (n mod 10)
    end if
end Print
```

Рассматривая операцию вывода на экран как элементарную, осуществите анализ трудоемкости алгоритма в зависимости от разрядности k числа n:

а) получите рекурсивную функцию трудоемкости алгоритма на основе анализа механизма декомпозиции задачи, приведите ее к замкнутому виду методом итераций;

б) определите трудоемкость алгоритма путем анализа дерева рекурсивных вызовов и возвратов;

в) определите трудоемкость механизма вызова процедуры (в предположении о сохранении 4 регистров).

Ответ:

а, б) $f_A(k) = 4k - 2$;

в) $f_{вызовов}(k) = 12k$.

Задача 4.23.

Для нахождения минимального элемента массива используется рекурсивная функция *MinR (l, r)*, которая возвращает наименьший из элементов, расположенных в позициях с *l* до *r* массива *a*.

```
MinR (l, r)
    if l=r then
        min ¬ a[l]
    else
        i ¬ (l+r) div 2
        m1 ¬ MinR(l, i)
        m2 ¬ MinR(i+1, r)
        if m1<m2 then
            min ¬ m1
        else
            min ¬ m2
        end if
    end if
    return (min)
end MinR
```

Предполагая, что количество элементов исходного массива n является положительной целой степенью двух, осуществите анализ трудоемкости алгоритма:

а) получите рекурсивную функцию трудоемкости алгоритма на основе анализа механизма декомпозиции задачи, приведите ее к замкнутому виду методом итераций;

б) определите трудоемкость алгоритма путем анализа дерева рекурсивных вызовов и возвратов;

в) определите трудоемкость механизма вызова процедуры (в предположении о сохранении 4 регистров).

Сравнив найденные оценки трудоемкости с результатом, полученным в задаче 2.10, сделайте вывод о целесообразности использования рекурсивного алгоритма для решения подобных задач.

Ответ:

а), б) $f_A(n) = 12n - 9$;

в) $f_{вызовов}(n) = 32n - 16$.

Задача 4.24.

Имеется элемент a некоторой алгебраической системы с операцией умножения и натуральное число n. Для нахождения величины a^n используется следующий рекурсивный алгоритм:

```
u (a, n)
    if n=1 then
        u ¬ a
    else
        t ¬ n div 2
        if n mod 2 = 0 then
            u ¬ u(a, t)* u(a, t)
        else
            u ¬ u(a, t)* u(a, t)*a
        end if
```

```
    end if
    return (u)
end u
```

Предполагая, что вероятность выполнения условия во внутреннем операторе if p=0,5, осуществите анализ трудоемкости алгоритма:

а) получите рекурсивную функцию трудоемкости алгоритма на основе анализа механизма декомпозиции задачи, приведите ее к замкнутому виду методом итераций (для упрощения процедуры приведения к замкнутому виду предполагается, что n является положительной целой степенью двух);

б) определите трудоемкость алгоритма путем анализа дерева рекурсивных вызовов и возвратов;

в) определите трудоемкость механизма вызова процедуры (в предположении о сохранении 4 регистров).

Сравните найденную оценку трудоемкости с асимптотической оценкой $F_A(n) = \Theta(n)$ итеративного алгоритма (реализуется последовательным перемножением a).

Ответ:

а) $f_A(n) \approx \frac{19}{2} n - \frac{15}{2}$;

б) $f_A(n) = \frac{19}{2} 2^{\lfloor \log_2 n \rfloor} - \frac{15}{2}$;

в) $f_{вызовов}(n) = 32 \cdot 2^{\lfloor \log_2 n \rfloor} - 16$.

Задача 4.25.

С целью повышения эффективности приведенного в предыдущей задаче алгоритма преобразуем его к следующему виду:

```
u (a, n)
    if n=1 then
```

```
        u ¬ a
    else
        t ¬ u(a, n div 2)
        if n mod 2 = 0 then
            u ¬ t*t
        else
            u ¬ t*t*a
        end if
    end if
    return (u)
end u
```

Решите предыдущую задачу для данного алгоритма, сравните результаты.

а) $f_A(n) \approx \frac{15}{2}\log_2 n + 2;$

б) $f_A(n) = \frac{15}{2}\lfloor \log_2 n \rfloor + 2;$

в) $f_{вызовов}(n) = 16\lfloor \log_2 n \rfloor + 16.$

Задача 4.26.

Для алгоритма сортировки слиянием, приведенного в теоретическом разделе, используя метод анализа дерева рекурсивных вызовов, определите функцию трудоемкости в лучшем случае. Сравните полученный результат с оценкой трудоемкости худшего случая, сделайте выводы.

Ответ:

$$F_A^{\vee}(n) = \frac{33}{2} n \log_2 n + 70n - 53.$$

Задача 4.27.

Осуществите анализ сложности алгоритма из задачи 4.22 по количеству операций целочисленного деления в зависимости от разрядности k числа n:

а) выпишите рекуррентное соотношение для заданной операции на основе анализа механизма декомпозиции задачи, решите рекуррентное соотношение (найдите верхнюю оценку сложности) методом индукции;

б) получите асимптотическую оценку количества операций путем анализа дерева рекурсии.

Ответ:

а) $f_A(k) = O(k)$;

б) $f_A(k) = k - 1$, $f_A(k) = \Theta(k)$.

Задача 4.28.

Осуществите анализ сложности алгоритма из задачи 4.23 по количеству операций проверки условия (в качестве значащей выбрать одну из двух имеющихся операций) в зависимости от размерности массива:

а) выпишите рекуррентное соотношение для заданной операции на основе анализа механизма декомпозиции задачи, решите рекуррентное соотношение с использованием теоремы о рекуррентных оценках;

б) получите асимптотическую оценку количества операций путем анализа дерева рекурсии.

Ответ:

а) $f_A(n) = \Theta(n)$;

б) $f_A(n) = n - 1$, $f_A(n) = \Theta(n)$.

Задача 4.29.

Осуществите анализ сложности алгоритма из задачи 4.24 по количеству операций умножения в зависимости от числа n:

а) выпишите рекуррентное соотношение для худшего случая на основе анализа механизма декомпозиции задачи, решите рекуррентное соотношение (найдите верхнюю оценку сложности) методом индукции;

б) получите асимптотическую оценку худшего случая путем анализа дерева рекурсии.

Ответ:

а) $f_A^{\wedge}(n) = \mathrm{O}(n)$;

б) $f_A^{\wedge}(n) \approx 2(n-1)$, $f_A^{\wedge}(n) = \Theta(n)$.

Задача 4.30.

Осуществите анализ сложности алгоритма из задачи 4.25 по количеству операций умножения в зависимости от числа n:

а) выпишите рекуррентное соотношение для худшего случая на основе анализа механизма декомпозиции задачи, решите рекуррентное соотношение с использованием теоремы о рекуррентных оценках;

б) получите асимптотическую оценку худшего случая путем анализа дерева рекурсии.

Ответ:

а) $f_A^{\wedge}(n) = \Theta(\log_2 n)$;

б) $f_A^{\wedge}(n) \approx 2\log_2 n$, $f_A^{\wedge}(n) = \Theta(\log_2 n)$.

5 Основы теории сложности алгоритмов

5.1 Теоретический предел трудоемкости задачи

Рассматривая некоторую алгоритмически разрешимую задачу D, и анализируя один из алгоритмов ее решения, мы можем получить оценку трудоемкости этого алгоритма в худшем случае – $F_A^{\wedge}(D) = \mathrm{O}(g(D))$. Такие же оценки мы можем получить и для других известных алгоритмов решения данной задачи. При рассмотрении задачи с этой точки зрения возникает справедливый вопрос – существует ли функциональный нижний предел F_{lim} для $g(D)$ и если «да», то существует ли алгоритм, решающий задачу с такой трудоемкостью в худшем случае.

Другая 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:

$$\min\left\{F_A^{\wedge}(D)\right\} = \Theta(F_{lim}).$$

Если имеется возможность на основе теоретических рассуждений доказать существование оценивающей функции и получить ее, то мы можем утверждать, что любой алгоритм, решающий данную задачу работает не быстрее, чем с оценкой F_{lim} в худшем случае:

$$F_A^{\wedge}(D) = \Omega(F_{lim}).$$

Приведем ряд примеров:

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

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), что приводит нас к оценке $F_{lim} = \Theta(n^2)$. При этом лучший известный на сегодня алгоритм умножения матриц имеет оценку $\Theta(n^{2,34})$. Расхождение между гипотетическим 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 трудоемкости.

5.2 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. Поиск ответа на этот вопрос привел к необходимости введения сложностных классов задач.

5.2.1 Класс P (задачи с полиномиальной сложностью)

Задача называется полиномиальной, то есть относится к классу P, если существует константа k и алгоритм, решающий задачу с $F_A(n) = O(n^k)$, где n – длина входа алгоритма.

Задачи класса P – это задачи, решение которых возможно за «реальное» или «разумное» время. Отметим следующие преимущества алгоритмов из этого класса:

1. Для большинства задач из класса P константа k меньше 6.
2. Класс P инвариантен по модели вычислений для широкого класса моделей. То есть мощность класса не зависит от выбора конкретной модели вычислений. Например, класс задач, которые могут быть решены за полиномиальное время на последовательной машине с произвольным доступом, совпадает с классом задач, полиномиально разрешимых на машинах Тьюринга.

3. Класс P обладает свойством естественной замкнутости вследствие того, что сумма, композиция или произведение полиномов есть полином. Например, композиция двух полиномиальный алгоритмов (выход первого подается на вход второго) также имеет полиномиальную оценку трудоемкости.

В связи с тем, что решение задач класса P возможно за реальное время, их также называют «практически разрешимыми» задачами.

5.2.2 Класс NP (полиномиально проверяемые задачи)

Существует достаточно обширный класс задач, трудоемкость решения которых факториальна или экспоненциальна. Другими словами, на больших объемах входных данных для этих задач неизвестен алгоритм их решения за разумное время. Такие задачи называются «практически неразрешимыми». Как будет показано ниже, единственный способ найти оптимальное или близкое к оптимальному решение такой задачи за разумное время состоит в том, чтобы каким либо образом получить предположительный ответ и проверить его правильность.

Рассмотрим, например, задачу о сумме.

Дано: 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 $\sum_{i=1}^{n} a_i \cdot x_i = V$. Или: может ли быть представлено число V в виде суммы каких либо элементов массива A.

Если какой-то алгоритм выдает предположительный результат – массив X, то проверка правильности этого результата может быть выполнена с полиномиальной сложностью: проверка $\sum_{i=1}^{n} a_i \cdot x_i = V$ потребует $\Theta(N)$ операций.Таким образом, задача относится к классу NP, если ее решение некоторым алгоритмом может быть быстро (полиномиально) проверено.

5.2.3 Соотношение между классами P и NP

После введения понятий классов сложности была сформулирована 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?

Очевидно, что любая задача, принадлежащая классу P, принадлежит и классу NP, так как она может быть полиномиально проверена. Например, если какой-либо алгоритм сортировки выдал ответ для входного набора данных длиной N, то его корректность может быть проверена за время $\Theta(N)$ путем организации N-1 операций сравнения элементов.

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.

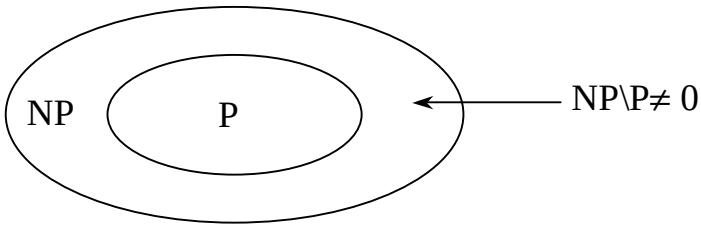


Рисунок 5.1 – Предполагаемое соотношение между классами P и NP

5.2.4 Класс NPC (NP-полные задачи)

Класс NPC представляет собой подмножество наиболее сложных задач из класса NP.

Понятие NP-полноты (NPC – NP-complete) основывается на понятии сводимости одной задачи к другой. Задача называется NP-полной, если имеется возможность найти способ сведения к ней всех остальных задач класса NP.

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 задачу 2 в терминах задачи 1 (свести задачу 2 к задаче 1), то мы решаем задачу 2.

Таким образом, если задача 1 задана множеством конкретных проблем D_{A1}, а задача 2 – множеством D_{A2}, и существует функция f_s (алгоритм), сводящая конкретную постановку задачи 2 (d_{A2}) к конкретной постановке задачи 1 (d_{A1}): $f_s(d_{A2} \in D_{A2}) = d_{A1} \in D_{A1}$, то задача 2 сводима к задаче 1. Если при этом трудоемкость функции (алгоритма) $F_A(n) = \mathrm{O}(n^k)$, то есть алгоритм сведения принадлежит классу P, то говорят, что задача 2 полиномиально сводится к задаче 1. Причем если задача 2 полиномиально сводится к задаче 1, и задача 1 решается за полиномиальное время, то и задача 2 также решается за полиномиальное время.

Например, первая задача состоит в том, чтобы вернуть значение «да» в случае, если одна из нескольких логических переменных имеет значение «истина», и вернуть «нет» в противоположном случае. Вторая задача заключается в том, чтобы найти максимальное значение в списке целых чисел. Пусть мы знаем решение задачи о поиске максимума, а задачу про логические переменные решать не умеем. Необходимо свести задачу о логических переменных к задаче о максимуме целых чисел.

Для этого потребуется алгоритм преобразования набора значений логических переменных в список целых чисел, который значению «ложь» сопоставляет число 0, а значению «истина» – число 1. Затем полученный массив целых числе передается на вход алгоритма поиска максимального элемента в списке. Максимальный элемент может быть либо нулем, либо единицей. Такой ответ можно преобразовать в ответ в задаче о логических переменных, возвращая «да», если максимальное значение равно 1, и «нет», если оно равно 0.

Поиск максимального значения выполняется за линейное время, а редукция первой задачи ко второй тоже требует линейного времени. Следовательно, задачу о логических переменных тоже можно решить за линейное время.

Принято говорить, что задача задается некоторым языком, тогда если задача 1 задана языком L_1, а задача 2 – языком L_2, то полиномиальная сводимость второй задачи к первой обозначается следующим образом: $L_2 \leq_p L_1$.

Определение принадлежности задачи к классу NPC требует проверки выполнения следующих двух условий: во-первых, задача должна принадлежать классу NP ($L \in$ NP), и, во-вторых, к ней полиномиально должны сводиться все задачи из класса NP ($L_x \leq_p L$, для каждого $L_x \in NP$).

На практике нет необходимости осуществлять редукцию для каждой NP задачи. Для того, чтобы доказать NP-полноту некоторой NP задачи A, достаточно свести к ней какую-нибудь NP-полную задачу B. Редуцировав задачу B к задаче A, мы показываем, что и любая NP задача может быть сведена к A за два шага, первый из которых – ее редукция к B.

Для класса NPC доказана следующая теорема. Если существует задача, принадлежащая классу NPC, для которой существует полиномиальный алгоритм решения ($F_A(n) = \mathrm{O}(n^k)$), то класс P совпадает с классом NP, то есть P=NP. Это означает, что если удастся найти полиномиальный алгоритм решения какой-либо NP-полной задачи, то все задачи класса NP могут быть решены за полиномиальное время (в силу доказанности возможности их сведения к решенной NP-полной задаче).

В настоящее время доказано существование сотен NP-полных задач, но ни для одной из них пока не удалось найти полиномиального алгоритма решения. В настоящее время исследователи предполагают следующее соотношение классов, показанное на рисунке 5.1, – P≠NP, то есть NP\P≠0, и задачи из класса

NPC не могут быть решены (сегодня) с полиномиальной трудоемкостью (то есть классы P и NPC не пересекаются).

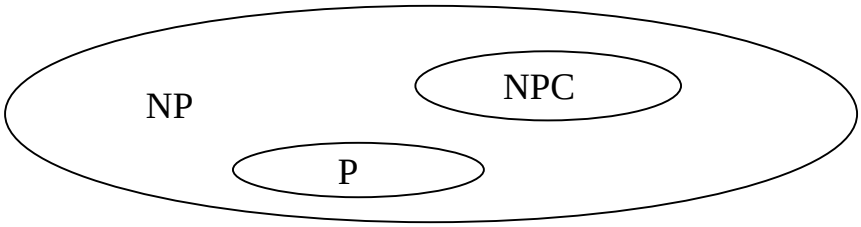


Рисунок 5.1 – Предполагаемое соотношение между классами P, NP и NPC.

5.2.5 Примеры типовых задач класса NP

Каждая из задач, относящаяся к классу NP, является либо оптимизационной, либо задачей о принятии решения. Целью оптимизационной задачи обычно является конкретный результат, представляющий собой минимальное или максимальное значение. В задаче о принятии решения обычно задается некоторое граничное значение, и нас интересует, существует ли решение, большее, меньшее или равное указанной границе. Оптимизационный вариант является более общим, и задача оптимизации всегда может быть преобразована к задаче принятия решения.

Задача о коммивояжере.

Имеется N городов, связанных дорогами. Заданы стоимости путешествия между городами. Необходимо найти маршрут движения коммивояжера, который обеспечивал бы посещение всех городов по одному разу. В оптимизационном варианте задача о коммивояжере является задачей минимизации стоимости пути. В варианте принятия решения задача выглядит так: определить, существует ли путь коммивояжера со стоимостью, меньшей заданной константы C.

Раскраска графа.

Вершины графа можно раскрасить в разные цвета, которые обычно обозначаются целыми числами. Нас интересуют такие раскраски, в которых концы каждого ребра окрашены разными цветами. Очевидно, что в графе с N вершинами можно покрасить вершины в N различных цветов.

Оптимизация: найти минимальное число цветов, необходимых для раскраски вершин графа.

Принятие решения: можно ли раскрасить вершины в C или менее цветов.

Раскладка по ящикам.

Пусть имеется несколько ящиков единичной емкости и набор объектов различных размеров s_1, s_2, ..., s_n.

Оптимизация: найти наименьшее количество ящиков, необходимое для раскладки всех объектов.

Принятие решения: можно ли упаковать все объекты в B или менее ящиков.

Упаковка рюкзака.

Имеется набор объектов объемом s_1, s_2, ..., s_n стоимости w_1, w_2, ..., w_n.

Оптимизация: необходимо упаковать рюкзак объемом K так, чтобы его стоимость была максимальной.

Принятие решения: можно ли добиться, чтобы суммарная стоимость упакованных объектов была по меньшей мере W.

Задача о суммах элементов подмножеств.

Пусть у нас есть множество объектов различных размеров s_1, s_2, ..., s_n и некоторая положительная верхняя граница L.

Оптимизация: найти набор объектов, сумма размеров которых наиболее близка к L и не превышает этой верхней границы.

Принятие решения: существует ли набор объектов с суммой размеров L.

Задача об истинности КНФ-выражения.

Задача ставится только в варианте принятия решения: существуют ли у переменных, входящих в выражение, такие значения истинности, подстановка которых делает все выражение истинным. Как число переменных, так и сложность выражения не ограничены, поэтому число комбинаций значений истинности может быть очень велико.

Задача планирования работ.

Пусть имеется набор работ и известно время, необходимое для завершения каждой из них, t_1, t_2, …, t_n, сроки d_1, d_2, …, d_n, к которым эти работы должны быть обязательно завершены, а также штрафы p_1, p_2, …, p_n, которые будут наложены при незавершении каждой работы в установленные сроки.

Оптимизация: установить порядок работ, минимизирующий накладываемые штрафы.

Принятие решения: существует ли порядок работ, при котором величина штрафа будет не больше P.

5.3 Недетерминированные полиномиальные алгоритмы

Сложность всех известных детерминированных (точных) алгоритмов, решающих практически неразрешимые задачи, либо экспоненциальна, либо факториальна. Существует иной подход к решению таких задач, основанный на использовании так называемых недетерминированных полиномиальных алгоритмов (именно отсюда возникло название класса NP). Термин недетерминированные в их названии означает, что они являются вероятностными и могут не давать точного решения. Однако, такие алгоритмы позволяют за разумное время по крайней мере приблизится к требуемому ответу.

Недетерминированные полиномиальные алгоритмы характеризуются следующим двухшаговым подходом к решению задачи:

1. На первом шаге имеется недетерминированный алгоритм, в общем случае генерирующий случайное возможное решение такой задачи. Иногда такая попытка оказывается успешной, и мы получаем оптимальный или близкий к оптимальному ответ, иногда безуспешной (ответ далек от оптимального).

2. На втором шаге проверяется, действительно ли ответ, полученный на первом шаге, является решением исходной задачи.

Каждый из этих шагов по отдельности требует полиномиального времени. Следовательно, общая сложность такого алгоритма также полиномиальна. Но на практике неизвестно, сколько раз необходимо повторить эту процедуру, чтобы получить точное решение (число операций в этом случае может иметь экспоненциальный или факториальный порядок роста).

В случае решения задач оптимизации, чем большее количество раз повторяются эти два шага, тем более близкий к точному ответ дает алгоритм. Необходимо отметить, что для оптимизационных задач на втором шаге речь не идет о проверке оптимальности ответа, так как точное решение неизвестно. Оценивается только его соотношение с наиболее близким к оптимальному (минимальным или максимальным) ответом, полученным на предыдущих итерациях.

Например, при решении задачи о коммивояжере таким способом на первом шаге случайным образом генерируется некоторое упорядочивание городов. Поскольку это недетерминированный процесс, каждый раз будет получаться новый порядок. Процесс генерации можно реализовать за полиномиальное время (генерируется случайный номер, выбирается из списка город с этим номером и удаляется из списка). Такая процедура выполняется за $O(N)$ операций, где N – число городов. На втором шаге происходит подсчет стоимости путешествия по городам в указанном порядке, что также требует $O(N)$ операций. Оба шага полиномиальны, но количество обращений к ним делает решение задачи времяемким.

5.4 Вопросы для самоконтроля

1. Что понимается под теоретическим пределом трудоемкости задачи? Какова практическая ценность информации о нем?
2. Перечислите и охарактеризуйте сложностные классы задач.
3. В чем заключается основная проблема теории сложности?
4. Каковы особенности класса NP-полных задач?
5. Приведите примеры задач класса NP.
6. Опишите общую схему работы и перечислите особенности недетерминированных полиномиальных алгоритмов.

Список литературы

1. Кормен, Т. Алгоритмы: построение и анализ [Текст]/ Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М.: МЦНМО, 2000. – 960 с.

2. Макконенелл, Дж. Основы современных алгоритмов [Текст]/Дж. Макконелл. – М.: Техносфера, 2004. – 368 с.

3. Ульянов, М.В. Математическая логика и теория алгоритмов [Текст] / М.В. Ульянов, М.В. Шептунов; в 3ч.;ч. 2. Теория алгоритмов .–М.:МГАПИ,2003. – 80 с.

4. Ахо, А. Структуры данных и алгоритмы [Текст]/ А. Ахо, Д. Хопкрофт, Д. Ульман. – М.: Издательский дом «Вильямс», 2000. – 384 с.

5. Кнут, Д. Искусство программирования для ЭВМ; в 3 т.; т. 1. Основные алгоритмы [Текст]/ Д. Кнут. – М.: Мир, 1976. – 734 с.

6. Кнут, Д. Искусство программирования для ЭВМ; в 3 т.; т. 3. Сортировка и поиск [Текст]/ Д. Кнут. – М.: Мир, 1976. – 844 с.

7. Вирт, Н. Алгоритмы и структуры данных [Текст]/ Н. Вирт. – М.: Мир, 1989. – 360 с.

Приложение А
Основные свойства сумм и формулы суммирования

В приведенных ниже формулах C и L – независящие от i постоянные.

1. Основные свойства, полезные при упрощении сумм:

1.1. $\sum_{i=1}^{n} C \cdot f(i) = C \cdot \sum_{i=1}^{n} f(i)$.

1.2. $\sum_{i=L}^{n} f(i) = \sum_{i=0}^{n-L} f(i+L)$, в частности $\sum_{i=L}^{n} i = \sum_{i=0}^{n-L} (i+L)$.

1.3. $\sum_{i=L}^{n} f(i) = \sum_{i=0}^{n} f(i) - \sum_{i=0}^{L-1} f(i)$, откуда следует следующее равенство.

1.4. $\sum_{i=0}^{n} f(i) = \sum_{i=0}^{L-1} f(i) + \sum_{i=L}^{n} f(i)$.

1.5. $\sum_{i=0}^{n} (f_1(i) + f_2(i)) = \sum_{i=0}^{n} f_1(i) + \sum_{i=0}^{n} f_2(i)$.

2. Формулы суммирования:

2.1. $\sum_{i=1}^{n} 1 = n$.

2.2. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

2.3. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3 + 3n^2 + n}{6}$.

2.4. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

2.5. Для произвольного целого числа $C \neq 1$: $\sum_{i=0}^{n} C^i = \frac{C^{n+1} - 1}{C - 1}$.

2.6. При $|C| < 1$ и достаточно больших n: $\sum_{i=0}^{n} C^i \approx \frac{1}{1 - C}$.

2.7. $\sum_{i=1}^{n} i 2^i = (n-1)2^{n+1} + 2$.

2.8. $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$.

2.9. $\sum_{i=1}^{n} \log_2 i \approx n \log_2 n - 1{,}5$.

Учебное издание

Фролов Алексей Иванович

ОСНОВЫ АНАЛИЗА АЛГОРИТМОВ

Учебное пособие

Редактор Г.А. Константинова
Технический редактор М.Н. Малахов

Орловский государственный технический университет
Лицензия ИД 00670 от 5.01.2000
АНО «Центр Интернет-образования»

Подписано к печати 12.05.2008 г. Формат 60×84 1\16
Печать офсетная. Усл. печ. л. 5,6. Тираж 100 экз.
Заказ №38/08 оп

Отпечатано с готового оригинал-макета
на полиграфической базе ОрелГТУ

302020, г. Орел, Наугорское шоссе, 29