ГОСУДАРСТВЕННОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизированные системы управления»

МАТЕМАТИЧЕСКАЯ ЛОГИКА И ТЕОРИЯ АЛГОРИТМОВ

Методические указания
по выполнению курсовой работы
для студентов специальности 23 01 02
«Автоматизированные системы обработки информации и управления»

Могилев 2009

УДК 621.01
ББК 36.4
М87

Рекомендовано к опубликованию
учебно-методическим управлением
ГУ ВПО «Белорусско-Российский университет»

Одобрено кафедрой «Автоматизированные системы управления»
«12» мая 2009 г., протокол № 13

Составитель канд. техн. наук, доц. С. А. Альховик

Рецензент канд. мат. наук, доц. Л.В. Плетнев

Методические указания по выполнению курсовой работы
для студентов специальности 23 01 02
«Автоматизированные системы обработки информации и управления»

Учебное издание

МАТЕМАТИЧЕСКАЯ ЛОГИКА И ТЕОРИЯ АЛГОРИТМОВ

Ответственный за выпуск	С. К. Крутолевич
Технический редактор	А. А. Подошевко
Компьютерная верстка	Н. П. Полевничая

Подписано в печать . Формат 60x84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл.-печ. л. . Уч.-изд. л. . Тираж 31 экз. Заказ №

Издатель и полиграфическое исполнение
Государственное учреждение высшего профессионального образования
«Белорусско-Российский университет»
ЛИ № 02330/375 от 29.06.2004 г.
212000, г. Могилев, пр. Мира, 43

1 Организация выполнения курсовой работы

Целью курсовой работы является приобретение студентом навыков анализа, теоретического и экспериментального исследования алгоритмов.

Выполнение курсовой работы осуществляется студентом на основе выданного ему индивидуального задания.

Во время выдачи задания уточняется и детализируется содержание каждой части работы, определяются календарные сроки окончания основных этапов, уточняется график индивидуальных консультаций.

Законченная и оформленная курсовая работа, подписанная студентом, предоставляется руководителю для рецензирования. Срок составления рецензии не должен превышать семи дней. В рецензии преподаватель должен отметить каждую ошибку и неточность с указанием, в чем заключается сущность ошибки. Недопустима расстановка вопросительных и других знаков без соответствующих разъяснений. Все исправления в тексте и замечания на полях рецензируемой работы необходимо писать чернилами, отличными от чернил, которыми написана работа. В рецензии должны быть представлены подробный анализ недостатков и ошибок и уровень соответствия, конкретно и четко сформулированы все требования, предъявляемые студенту. Курсовая работа направляется на доработку, если количество ошибок и погрешностей позволяют отнести ее к низкому уровню соответствия. Допустимые погрешности и ошибки при определении учебных достижений представлены в таблице 1.

Таблица 1 – Допустимые погрешности и ошибки при определении учебных достижений студентов

Шкала соответствия	Уровень соответствия	Ошибки/несущественные погрешности/существенные погрешности	Сумма баллов при тестировании, %
Соответствие	Высокий	1/0/0	95–100
		2/1/0	90–94
		3/1/1	85–89
	Средний	4/2/1	80–84
		5/2/3	75–79
		6/3/2	68–74
	Минимально необходимый	7/4/3	51–67
Несоответствие	Низкий	8/5/4	40–49
		9/6/5	30–39
		10/10/10	<30

Примечания.

1 Погрешностями при определении учебных достижений считаются: неточные выражения; нерациональные приемы вычислений и математических выражений; арифметические ошибки в расчетах и построениях; неточное выполнение записей, рисунков, схем, графиков, моделей; непоследовательное

выполнение действий и операций.

2 К несущественным ошибкам относятся: неточности определения понятий (неполный охват основных признаков определяемого понятия или замена одного или нескольких из основных признаков второстепенными); неточности изложений законов, правил, принципов, теорий, методов решения задач и алгоритмов; нерациональный способ решения задачи или план ответа (нарушение логики изложения материала, подмена основных понятий второстепенными); отсутствие ссылок на использованные источники; несоблюдение требований ГОСТа и небрежное оформление пояснительной записки и графического материала.

3 К существенным ошибкам относятся: подмена понятий в изложении основных понятий, формулировок законов, правил, положений теории, формул, величин; незнание фундаментальных понятий и категорий, объектов, моделей, постоянных характеристик и параметров систем и процессов; неумение выделять главное в ответе, делать выводы и обобщения, неумение письменно оформить материал; неумение применять теоретические знания для решения задач; отсутствие или неполное выполнение раздела работы, если его объем явно указан в задании или методических указаниях.

При повторном рецензировании преподаватель должен проверить лишь выполнение (исправление) его предыдущих замечаний. Указание новых замечаний не допускается. Если работа удовлетворяет требованиям, предъявляемым к ней, она допускается к защите, о чем руководитель делает надпись на пояснительной записке. Защита состоит в коротком докладе студента по выполненной работе и в ответах на вопросы.

2 Содержание курсовой работы

Примерная структура расчетно-пояснительной записки приведена в таблице 2. Основными требованиями к расчетно-пояснительной записке являются четкость и логическая последовательность изложения материала, убедительность аргументации, краткость и ясность формулировок. В тексте записки не должно быть общих фраз, очевидных выводов и т. п. Объем пояснительной записки не более 40 страниц текста, иллюстраций, таблиц.

Содержание разделов расчетно-пояснительной записки курсовой работы должно соответствовать приведенным ниже рекомендациями.

Пояснительная записка начинается **аннотацией**, в которой в лаконичной форме излагаются основные результаты выполненной работы.

Расположение наименований разделов и подразделов в пояснительной записке отображается в разделе «**Содержание**», в котором указываются номера страниц по сквозной нумерации.

Во **введении** обосновывается актуальность темы курсовой работы.

В **разделах с 1 по 4** приводятся результаты анализа, теоретического и экспериментального исследования алгоритма, дается подробное описание разработанного программного обеспечения.

Таблица 2 – Структура пояснительной записки

Наименование раздела	Рекомендуемый объем, с.
Титульный лист	1
Задание	1
Аннотация	1
Содержание	1
Перечень условных обозначений	1
Введение	1
1 Анализ и теоретическое исследование алгоритма	10
2 Разработка технологии экспериментального исследования алгоритма	5
3 Описание разработанного программного обеспечения	10
4 Экспериментальное исследование алгоритма	10
Заключение	1
Список использованных источников	1

В **заключении** приводятся основные выводы по результатам проделанной работы.

3 Оформление курсовой работы

Оформление курсового проекта должно соответствовать требованиям (ГОСТ 2.105-95). Текстовая часть пояснительной записки выполняется либо чертежным шрифтом по ГОСТ 2304-81 с высотой букв не менее 5 мм, либо машинным способом шрифтом Times New Roman, высотой букв 13 пунктов, через полтора интервала.

4 Методические рекомендации по выполнению курсовой работы

4.1 Введение в анализ алгоритмов

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

В качестве формальной системы будем рассматривать абстрактную машину, включающую процессор с фоннеймановской архитектурой, поддерживающий адресную память и набор «элементарных» операций, соотнесенных с языком высокого уровня.

В целях дальнейшего анализа примем следующие допущения:

- каждая команда выполняется не более чем за фиксированное время;

- исходные данные алгоритма представляются машинными словами по β битов каждое.

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

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

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

S_d – память для хранения промежуточных результатов;

S_r – память для организации вычислительного процесса (память, необходимая для реализации рекурсивных вызовов и возвратов).

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

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

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

$$\psi_A = c_1 \cdot F_a(N) + c_2 \cdot M + c_3 \cdot S_d + c_4 \cdot S_r,$$

где c_i – веса ресурсов.

4.1.2 *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 и $|D| = N$.

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

- обозначим это подмножество через D_N: $D_N = \{D \in D_N,: |D| = N\}$;
- обозначим мощность множества D_N через $M_{DN} \rightarrow M_{DN} = |D_N|$.

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

$F_a\hat{}(N)$ – *худший случай* – наибольшее количество операций, совершаемых

алгоритмом А для решения конкретных проблем размерностью N: $F_a^{\wedge}(N) = \max \{F_a(D)\}$ – худший случай на D_N;

$F_a^{\vee}(N)$ – *лучший случай* – наименьшее количество операций, совершаемых алгоритмом А для решения конкретных проблем размерностью N: $F_a^{\vee}(N) = \min \{F_a(D)\}$ – лучший случай на D_N;

$\overline{F}_a(N)$ – *средний случай* – среднее количество операций, совершаемых алгоритмом А для решения конкретных проблем размерностью N: $F_a(N) = (1 / M_{DN}) \cdot \sum \{F_a(D)\}$ – средний случай на D_N.

4.1.3 *Классификация алгоритмов по виду функции трудоёмкости.* В зависимости от влияния исходных данных на функцию трудоемкости алгоритма может быть предложена следующая классификация, имеющая практическое значение для анализа алгоритмов.

*Количественно-зависимые по трудоемкости алгоритмы.*Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа и не зависит от конкретных значений:

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

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

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

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

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

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

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

Количественно-параметрические по трудоемкости алгоритмы. Однако в большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от значений входных данных; в этом случае:

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

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

Порядково-зависимые по трудоемкости алгоритмы. Среди разнообразия параметрически-зависимых алгоритмов выделим еще одну группу, для которой

количество операций зависит от порядка расположения исходных объектов.

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

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

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

Примерами таких алгоритмов могут служить некоторые алгоритмов сортировки, алгоритмы поиска минимума и максимума в массиве.

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

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

В асимптотическом анализе приняты следующие обозначения.

Оценка Θ (тетта).

Пусть f(n) и g(n) – положительные функции положительного аргумента, $n \geq 1$ (количество объектов на входе и количество операций – положительные числа), тогда $f(n) = \Theta(g(n))$, если существуют положительные c_1, c_2, n_0, такие, что $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, при $n > n_0$.

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

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

Примеры:

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

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

Оценка O (О большое).

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

$$\exists\ c > 0,\ n0 > 0 : 0 \leq f(n) \leq c \cdot g(n),\ \forall n > n0.$$

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

Например, для всех функций: $f(n) = 1/n$, $f(n) = 12$, $f(n) = 3\cdot n+17$, $f(n) = n\cdot \mathrm{Ln}(n)$, $f(n) = 6\cdot n^2 +24\cdot n+77$ будет справедлива оценка $O(n^2)$.

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

Оценка Ω (Омега).

В отличие от оценки O, оценка Ω является оценкой снизу, т. е. определяет

класс функций, которые растут не медленнее, чем g(n), с точностью до постоянного множителя:

$$\exists\ c > 0,\ n0 > 0$$
$$0 \le c \cdot g(n) \le f(n).$$

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

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

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

4.2 Трудоемкость алгоритмов и временные оценки

4.2.1 *Элементарные операции в языке записи алгоритмов.* Для получения функции трудоемкости алгоритма, представленного в формальной системе введенной абстрактной машины, необходимо уточнить понятия «элементарных» операций, соотнесенных с языком высокого уровня.

В качестве таких «элементарных» операций предлагается использовать следующие:

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

2) одномерная индексация a[i] : (адрес (a)+i·длина элемента);

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

4) операции сравнения: $a < b$;

5) логические операции (l1) {or, and, not} (l2).

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

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

- конструкция «Следование».

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

$$F_{\text{«следование»}} = f_1 + \ldots + f_k,$$

где k – количество блоков;

- конструкция «Ветвление».

Общая трудоемкость конструкции «Ветвление» требует анализа вероятности выполнения переходов на блоки «Then» (p) и «Else» (1 - p) и определяется как

$$F_{\text{«ветвление»}} = f_{then} \cdot p + f_{else} \cdot (1 - p);$$

- конструкция «Цикл».

После сведения конструкции к элементарным операциям ее трудоемкость определяется как

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

4.2.2 Примеры анализа простых алгоритмов.

4.2.2.1 Задача суммирования элементов квадратной матрицы.

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

Алгоритм выполняет одинаковое количество операций при фиксированном значении n, и, следовательно, является количественно-зависимым. Применение методики анализа конструкции «Цикл» дает $F_A(n) = 1+1+ n \cdot (3+1+ n \cdot (3+4))=7 n^2+4 \cdot n +2 = \Theta(n^2)$, заметим, что под n понимается линейная размерность матрицы, в то время как на вход алгоритма подается n^2 значений.

4.2.2.2 Задача поиска максимума в массиве.

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

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

- худший случай.

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

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

- лучший случай.

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

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

- средний случай.

Алгоритм поиска максимума последовательно перебирает элементы массива, сравнивая текущий элемент массива с текущим значением максимума. На очередном шаге, когда просматривается *к-ый* элемент массива, переприсваивание максимума произойдет, если в подмассиве из первых *к* элементов максимальным элементом является последний. Очевидно, что в случае равномерного распределения исходных данных, вероятность того, что максимальный из *к* элементов расположен в определенной (последней) позиции, равна *1/к*. Тогда в массиве из *n* элементов общее количество операций переприсваивания максимума определяется как

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

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

$$F_A(n) = 1 + (n – 1) (3+2) + 2 (Ln(n) + \gamma) = 5\ n +2\ Ln(n) – 4 +2\ \gamma = \Theta(n).$$

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

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

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

- неадекватность формальной системы записи алгоритма и реальной системы команд процессора;

- наличие архитектурных особенностей, существенно влияющих на наблюдаемое время выполнения программы, таких как конвейер, кеширование памяти, предвыборка команд и данных и т. д.;

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

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

Пооперационный анализ.

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

$$T_A(N) = \sum F_{аопi}(N) \cdot \bar{t}_{опi}.$$

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 (рисунок 1), создаются соответствующие тестовые программы, и определяется среднее время на операцию в данном типе задач – $\bar{t}_{\text{тип задачи}}$.

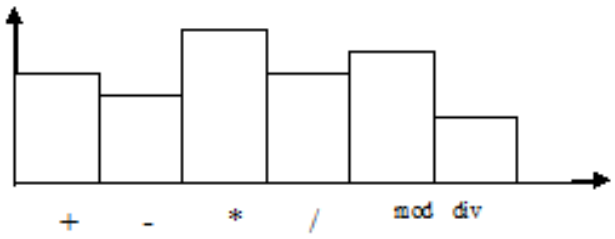


Рисунок 1 – Возможный вид частотной встречаемости операций

На основе полученной информации оценивается общее время работы алгоритма в виде

$$T_A(N) = F_A(N) \cdot \bar{t}_{\text{тип задачи}}.$$

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:

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

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

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

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

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

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

End.

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

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

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

End.

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

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

Введем параметры q и r, устанавливающие соотношения между

временами выполнения операции умножения, сложения и присваивания для операндов действительного типа.

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

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

Тогда приведенные к t_+ временные оценки имеют вид:

$$TA1 = 4 \cdot q \cdot t_+ + 2 \cdot t_+ + 2 \cdot r \cdot t_+ = t_+ \cdot (4 \cdot q + 2 + 2 \cdot r);$$

$$TA2 = 3 \cdot q \cdot t_+ + 5 \cdot t_+ + 5 \cdot r \cdot t_+ = t_+ \cdot (3 \cdot q + 5 + 5 \cdot r).$$

Равенство времен будет достигнуто при условии

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

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

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

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

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

4.3 Теория сложности вычислений и сложностные классы задач

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

Другая, более точная, формулировка имеет следующий вид: какова оценка сложности самого «быстрого» алгоритма решения данной задачи в худшем случае? Очевидно, что это оценка самой задачи, а не какого-либо алгоритма ее

решения. Таким образом, мы приходим к определению понятия функционального теоретического нижнего предела трудоемкости задачи в худшем случае:

$$F_{thlim} = \min \{ \Theta (F_a^{\wedge}(D)) \}.$$

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

$$F_a^{\wedge}(D) = \Omega (F_{thlim}).$$

Приведем ряд примеров.

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

2 Задача умножения матриц – для этой задачи можно сделать предположение, что необходимо выполнить некоторые арифметические операции со всеми исходными данными, теоретическое обоснование какой–либо другой оценки на сегодня неизвестно, что приводит нас к оценке $F_{thlim}=\Theta (n^2)$. Отметим, что лучший алгоритм умножения матриц имеет оценку $\Theta (n^{2,34})$. Расхождение между теоретическим пределом и оценкой лучшего известного алгоритма позволяет предположить, что либо существует, но еще не найден, более быстрый алгоритм умножения матриц, либо оценка $\Theta (n^{2,34})$ должна быть доказана, как теоретический предел.

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

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

Класс P (задачи с полиномиальной сложностью).

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

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

1 Для большинства задач из класса P константа k меньше 6.

2 Класс P инвариантен по модели вычислений (для широкого класса моделей).

3 Класс P обладает свойством естественной замкнутости (сумма или произведение полиномов есть полином).

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

Класс NP (полиномиально проверяемые задачи).

Представим себе, что некоторый алгоритм получает решение некоторой

задачи – соответствует ли полученный ответ поставленной задаче и насколько быстро мы можем проверить его правильность?

Рассмотрим, например, задачу о сумме.

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 $\sum a_i x_i = V$.

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

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

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

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

Класс NPC (NP – полные задачи).

Понятие NP – полноты основывается на понятии сводимости одной задачи к другой.

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 возможно переформулировать (свести) задачу 2 в терминах задачи 1, то мы решаем задачу 2.

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

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

Принято говорить, что задача задается некоторым языком. Тогда если задача 1 задана языком L1, а задача 2 – языком L2, то полиномиальная сводимость обозначается как $L2 \le_p L1$.

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

В настоящее время доказано существование сотен NP–полных задач, но ни для одной из них пока не удалось найти полиномиального алгоритма решения.

4.4 Пример полного анализа алгоритма решения задачи о сумме

4.4.1 *Формулировка задачи и асимптотическая оценка.* Словесно задача о сумме формулируется как задача нахождения таких чисел из данной совокупности, которые в сумме дают заданное число. Классически задача формулируется в терминах целых чисел.

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

Детальная формулировка:

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

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

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

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

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

1) V содержит одно слагаемое $\Rightarrow C_N^1 = N$ вариантов;

2) V содержит два слагаемых $\Rightarrow C_N^2 = (N\cdot(N-1))/(1\cdot2)$ вариантов;

3) V содержит три слагаемых $\Rightarrow C_N^3 = (N\cdot(N-1)\cdot(N-2))/(1\cdot2\cdot3)$ вариантов

и т. д. до проверки одного варианта с N слагаемыми.

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

$$\hat{F}_A(N, V) = O(N\cdot2^N).$$

4.4.2 *Алгоритм точного решения задачи о сумме (метод перебора).* Определим вспомогательный массив, хранящий текущее сочетание исходных чисел в массиве S, подлежащих проверке на V – массив Cnt[j]. Элемент массива равен «0», если число S[j] не входит в V, и равен «1», если число S[j] входит в V.

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

Могут быть предложены следующие две реализации механизма полного перебора вариантов:

1) перебор по всевозможным сочетаниям из *k* элементов по N, т. е. сначала алгоритм пытается представить V как один из элементов массива S, затем перебираются все возможные пары, затем все возможные тройки и т. д.;

2) перебор по двоичному счётчику, реализованному в массиве Cnt.

Вторая идея алгоритмически более проста и сводится к решению задачи

об увеличении двоичного счётчика в массиве Cnt на «1»:

- при 00...0111 увеличение на «1» приводит к сбросу всех правых «1» и установке в «1» следующего самого правого «0»;

- при 00...1000, когда последний элемент счетчика равен «0», увеличение на «1» приводит к переустановке последнего элемента в массиве Cnt с «0» в «1».

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

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

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

4.4.3 *Анализ алгоритма точного решения задачи о сумме.* Рассмотрим лучший и худший случай для данного алгоритма.

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

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

Получим детальную оценку для худшего случая, используя принятую

методику подсчета элементарных операций:

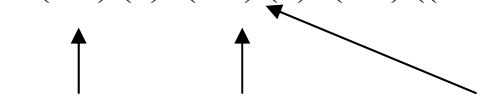
$$F_a\hat{}(N) = 2+N\cdot(3+2)+2+(2^N-1)\cdot\{2+N\cdot(3+5)+1+1+f_{cnt}+2+2\}.$$

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

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

Таким образом:

$$f_{cnt} = (1/2)\cdot(2)+(1/2)\cdot(2)+(1/2)\cdot((1/2)\cdot1\cdot6+(1/4)\cdot2\cdot6+(1/8)\cdot3\cdot6+\ldots) =$$



$\tilde{f}_{cnt} = 2$

$\tilde{f}_{cnt} = N\cdot6+2$

$f = \Theta(1)$

Р-чётных Р-нечётных выход из While

$$=2 + 1/2 \cdot 6 \cdot (1/2^1+2/2^2+3/2^3+\ldots) = 2 + 3 \cdot (\sum k/2^k).$$

Так как $\sum k\cdot x^k = x/(1-x)^2$, то $\sum k\cdot(1/2)^k = (1/2)/(1-(1/2))^2 = 2$ и, следовательно, $f_{Cnt} = 8$ (и не зависит от длины счетчика).

Подстановка f_{Cnt} дает

$$F\hat{}_A(N) = 4+5\cdot N+(2^N-1)\cdot(8\cdot N+16)$$

и, окончательно,

$$F\hat{}_A(N) = 8\cdot N\cdot2^N+16\cdot2^N-3\cdot N-12,$$

что согласуется с асимптотической оценкой.

4.5 Рекурсивные алгоритмы и методы их анализа

4.5.1 *Рекурсивные функции.* По сути один и тот же метод применительно к различным областям носит различные названия (индукция, рекурсия и рекуррентные соотношения), различия касаются особенностей использования.

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

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

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

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

Примеры рекурсивного задания функций:

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

Таким образом, $f(n)=n$.

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

Последовательная подстановка дает $f(n) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = n!$

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

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

Пример 3.
$$\begin{cases} f(0) = 1; \\ f(1) = 1; \\ f(n) = f(n-1) + f(n-2), \ n \geq 2. \end{cases}$$

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

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

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

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

Итак, одна и та же функция может иметь различные рекурсивные определения $f(n) = 2^n$, как и в примере 4, что проверяется элементарной подстановкой.

Пример 6.

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

В этом случае мы можем получить решение в «замкнутой» форме, сопоставив значениям функции соответствующие степени двойки: $f(2) = 2 = 2^1$, $f(3) = 4 = 2^2$, $f(4) = 8 = 2^3$, $f(5) = 32 = 2^5$, $f(6) = 256 = 2^8$.

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

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

Рассмотрим пример рекурсивной функции, вычисляющей факториал.

```
F(n);
        If n=0 or n=1 (проверка возможности прямого вычисления)
         Then
              F ← 1
         Else
              F ← n*F(n-1); ( рекурсивный вызов функции)
Return (F);
End.
```

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

Можно заметить, что некоторая ветвь дерева рекурсивных вызовов обрывается при достижении такого значения передаваемого параметра, при котором функция может быть вычислена непосредственно. Таким образом, рекурсия эквивалентна конструкции цикла, в котором каждый проход есть выполнение рекурсивной функции с заданным параметром.

Рассмотрим пример для функции вычисления факториала (рисунок 2).

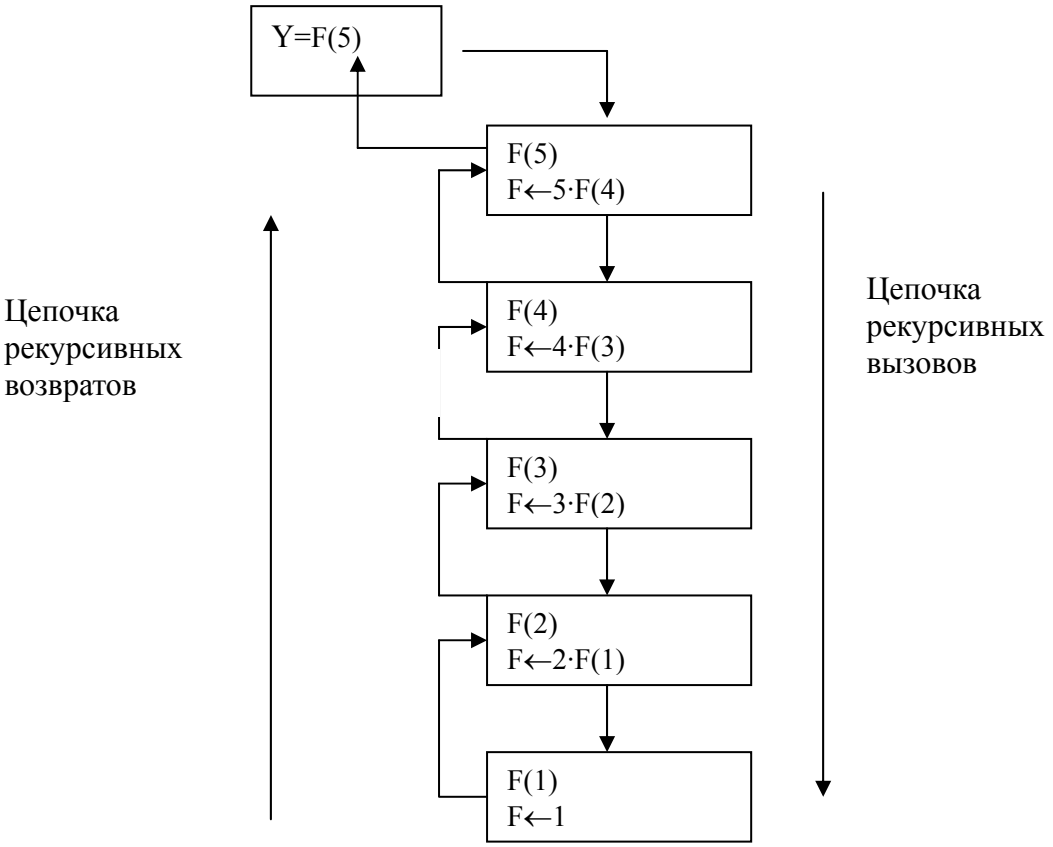


Рисунок 2 – Дерево рекурсии при вычислении факториала F(5)

Дерево рекурсивных вызовов может иметь и более сложную структуру, если на каждом вызове порождается несколько обращений – фрагмент дерева рекурсий для чисел Фибоначчи представлен на рисунке 3.

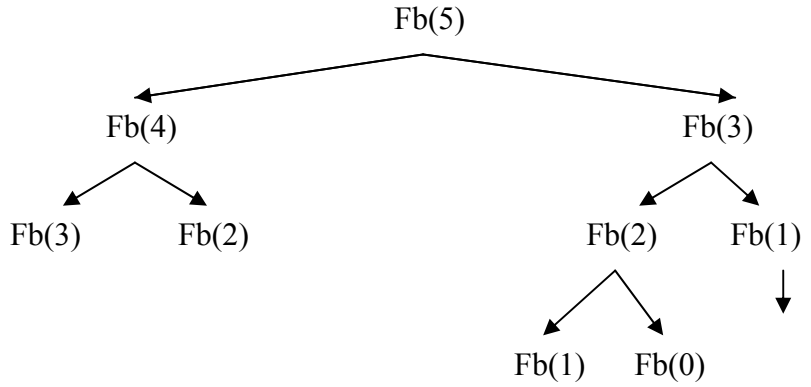


Рисунок 3 – Фрагмент дерева рекурсии при вычислении чисел Фибоначчи F(5)

Анализ трудоемкости механизма вызова процедуры. Механизм вызова функции или процедуры в языке высокого уровня существенно зависит от архитектуры компьютера и операционной системы. В рамках IBM PC-совместимых компьютеров этот механизм реализован через программный стек. Передаваемые в процедуру или функцию фактические параметры и

возвращаемые из них значения помещаются в программный стек специальными командами процессора. Дополнительно сохраняются значения необходимых регистров и адрес возврата в вызывающую процедуру. Схематично этот механизм иллюстрирован на рисунке 4.

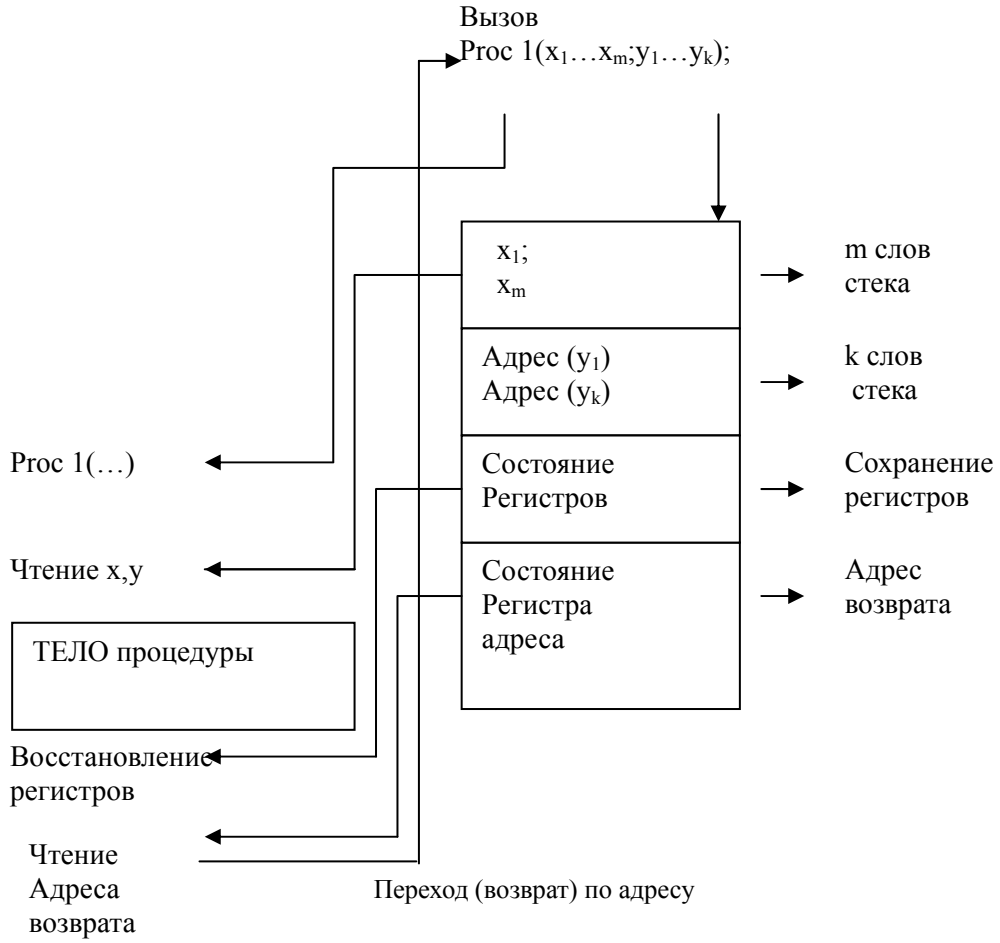


Рисунок 4 – Механизм вызова процедуры с использованием программного стека

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

Обозначив через m количество передаваемых фактических параметров, k – количество возвращаемых процедурой значений, r – количество сохраняемых в стеке регистров, имеем $f_{вызова} = m+k+r+1+m+k+r+1 = 2\cdot(m+k+r+1)$ элементарных операций на один вызов и возврат.

Анализ трудоемкости рекурсивных алгоритмов в части трудоемкости самого рекурсивного вызова можно выполнять разными способами в зависимости от того, как формируется итоговая сумма элементарных операций

– рассмотрением в отдельности цепочки рекурсивных вызовов и возвратов или совокупно по вершинам дерева рекурсивных вызовов.

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

$$f_A(n) = n\cdot 2\cdot (1+1+4+1)+(n-1)\cdot(1+3)+1\cdot 2=18\cdot n-2.$$

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

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

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

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

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

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

4.5.5 *Методы решения рекурсивных соотношений.* В математике разработан ряд методов, с помощью которых можно получить явный вид рекурсивно заданной функции – метод индукции, формальные степенные ряды, метод итераций и т.д. Рассмотрим некоторые из них.

1) Метод индукции.

Метод состоит в том, чтобы сначала угадать решение, а затем доказать его правильность по индукции.

Пример:

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

Предположение: $f(n) = 2^n$.

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

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

Заметим, что базис существенно влияет на решение, например:

если $f(0) = 0$, то $f(n) = 0$;

если $f(0) = 1/7$, то $f(n) = (1/7)\cdot 2^n$;

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

2) Метод итерации (подстановки).

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

Пусть $f(n) = 3 \cdot f(n/4)+n$, тогда

$$f(n) = n+3 \cdot f(n/4) = n+3 \cdot [n/4+3 \cdot f(n/16)] = n+3 \cdot [n/4+3\{n/16+3 \cdot f(n/64)\}].$$

Раскрывая скобки, получаем

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

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

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

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

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

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

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

U(n) – трудоемкость алгоритма объединения полученных решений.

Рассмотрим, например, известный алгоритм сортировки слиянием, принадлежащий Дж. фон Нейману.

На каждом рекурсивном вызове переданный массив делится пополам, что дает оценку для $d(n) = \Theta(1)$. Далее рекурсивно вызывается сортировка полученных массивов половинной длины (до тех пор, пока длина массива не станет равной единице), и сливаются возвращенные отсортированные массивы за $\Theta(n)$. Тогда ожидаемая трудоемкость на сортировку составит

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

Возникает задача о получении оценки сложности функции трудоемкости для произвольных значений a и b.

4.5.7 *Основная теорема о рекуррентных соотношениях.* Теорема принадлежит Дж. Бентли, Д. Хакену и Дж. Саксу.

Теорема. Пусть $a \ge 1$, $b > 1$ - константы, g(n) – функция. Далее

$$f(n)=a \cdot f(n/b)+g(n),$$

где $n/b = \lfloor (n/b) \rfloor$ или $\lceil (n/b) \rceil$.

Тогда:

Если $g(n) = O(n^{\log_b a-\xi})$, $\xi>0$, то $f(n) = \Theta(n^{\log_b a})$. Пример – $f(n) = 8f(n/2)+n^2$, тогда $f(n) = \Theta(n^3)$.

Если $g(n) = \Theta(n^{\log_6 a})$, то $f(n) = \Theta(n^{\log_b a} \cdot \log n)$. Пример – $f_A(n)= 2 \cdot f_A(n/2)+\Theta(n)$, тогда $f(n) = \Theta(n \cdot \log n)$.

Если $g(n) = \Omega(n^{\log_b a+e})$, $e > 0$, то $f(n) = \Theta(g(n))$. Пример – $f(n) = 2 \cdot f(n/2)+n^2$, имеем $n^{\log_b a} = n^1$ и, следовательно $f(n) = \Theta(n^2)$.

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

4.5.8 Прямой анализ рекурсивного дерева вызовов.

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

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

```
MS(A, p ,q, Bp, Bq)
If p≠q        (проверка останов рекурсии при p=q)
      then
            r←(p+q) div 2
            MS(A, p, r, Bp,Bq)        (рекурсивный вызов для первой части)
            MS(A, r+1, q, Bp,Bq)  (рекурсивный вызов для второй части)
            Merge(A, p, r, q, Bp, Bq) (слияние отсортированных частей)
Return (A)
End
```

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

Merge (A,p,r,q,Bp,Bq)	Количество операций в данной строке
Max ← A[r]	2
If Max <A[q] Then	2
Max ← A[q]	½·2
kp ← r - p + 1	3
p1 ← p – 1	2
For i ← 1 to kp (копирование первой части)	1+ kp·3
Bp [i] ← A[p1 + i]	kp· (4)
Bp[kp+1] ← Max	3
kq ← q - r	2

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

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

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

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

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

4.5.8.3 Подсчет вершин в дереве рекурсивных вызовов. Алгоритм, получая на входе массив из n элементов, делит его пополам при первом вызове, поэтому рассмотрим случай, когда $n = 2^k$, $k = \log_2 n$.

В этом случае имеем полное дерево рекурсивных вызовов глубиной k, содержащее n листьев. Фрагмент дерева показан на рисунке 5.

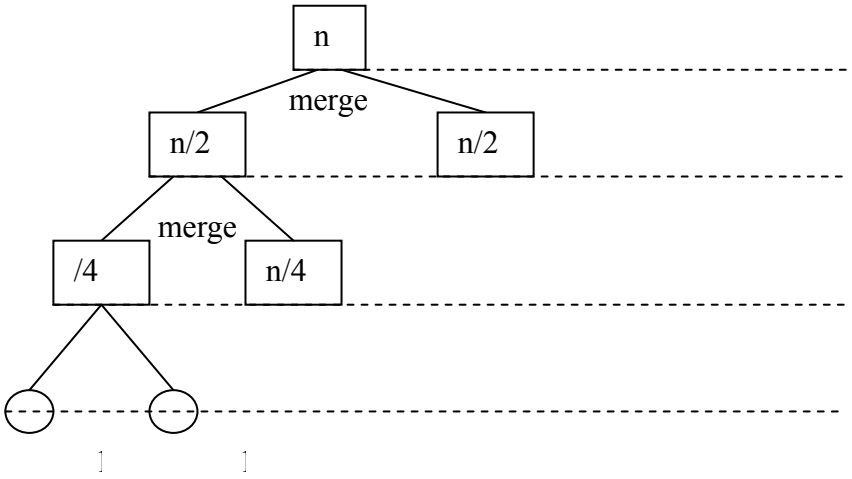


Рисунок 5 – Фрагмент рекурсивного дерева при сортировке слиянием

Обозначим количество вершин дерева через V:

$$V = n + n/2+n/4+n/8+...+1 = n\cdot(1+1/2+1/4+1/8+...+1) = 2n – 1 = 2^{k+1} – 1.$$

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

4.5.8.4 Анализ трудоемкости алгоритма сортировка слиянием. Для n листьев дерева выполняются вызов процедуры MS с вычислением r+1, проверка условия p = q и возврат в вызывающую процедуру для слияния, что в сумме с учетом трудоемкости вызова даёт

$$F1(n) = n\cdot2\cdot(5+4+1) + n\cdot2(\text{If } p = q \text{ и } r+1) = 22\cdot n.$$

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

$$Fr(n) = (n - 1)\cdot2\cdot(5+4+1) + (n - 1)\cdot(1+3+1) = 24\cdot n - 24.$$

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

Трудоемкость вызова составит (для шести параметров и четырех регистров)

$$Fm_{вызов}(n) = (n - 1)\cdot2\cdot(6+4+1) = 22\cdot n - 22.$$

Поскольку трудоемкость процедуры слияния для массива длиной m составляет 18·m + 23, и процедура вызывается n – 1 раз с длинами массива, равными n, n/2, n/4, …, причем 2 раза с длиной n/2, 4 раза с длиной n/4, то совокупно имеем

$$Fm_{слияние}(n) = (n - 1)\cdot23 + 18\cdot n + 2\cdot18\cdot(n/2) + 4\cdot18\cdot(n/4) + \ldots + =$$
$$= 18\cdot n\cdot(\log_2 n - 1) + 23\cdot(n - 1) = 18\cdot n\cdot\log_2 n + 5\cdot n - 23.$$

Учитывая все компоненты функции трудоемкости, получаем окончательную оценку:

$$Fa(n) = F1(n) + Fr(n) + Fm_{вызов}(n) + Fm_{слияние}(n) = 22\cdot n + 24\cdot n - 24 + 22\cdot n -$$
$$- 22 + 18\cdot n\cdot\log_2 n + 5\cdot n - 23 = 18\cdot n\cdot\log_2 n + 73\cdot n - 69.$$

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

5 Список литературы

1 **Ахо, А.** Теория синтаксического анализа, перевода и компиляции. Т. 1: Синтаксический анализ / А. Ахо. – М. : Мир, 1978. – 612 с. : ил.

2 **Ахо, А.** Структуры данных и алгоритмы: пер. с англ. / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М. : Вильямс, 2001. – 384 с. : ил.

3 **Вирт, Н.** Алгоритмы и структуры данных: пер. с англ. / Н. Вирт. – 2-е изд., испр. – СПб. : Невский диалект, 2001. – 352 с. : ил.

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

5 Искусство программирования : пер. с англ. : в 3 т. / Д. Кнут. – 3-е изд. – М. : Вильямс, 2001.

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

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

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

9 **Романовский, И. В.** Дискретный анализ : учеб. пособие для студентов, специализирующихся по прикладной математике / И. В. Романовский. – 2-е изд., испр. – СПб. : Невский диалект, 2000. – 240 с. : ил.

10 **Новиков, Ф. А.** Дискретная математика для программистов : учеб. пособие / Ф. А. Новиков. – СПб. : Питер, 2007. – 364 с.

11 **Карпов, Ю. Г.** Теория автоматов : учебник / Ю. Г. Карпов. – СПб. : Питер, 2003. – 208с.

12 **Плотников, А. Д.** Дискретная математика : учеб. пособие / А. Д. Плотников. – М. : Новое знание, 2005. – 288с.

13 **Хаггарти, Р.** Дискретная математика для программистов : пер. с англ. / Р. Хаггарти ; под ред. С. А. Кулешова. – 2-е изд., доп. – М. : Техносфера, 2005. – 400 с.

14 **Аляев, Ю. А.** Дискретная математика и математическая логика для экономистов: учебник / Ю. А. Аляев, С. Ф. Тюрин. – М. : Финансы и статистика, 2006. – 368с.

15 **Иванов, Б. Н.** Дискретная математика. Алгоритмы и программы : учеб. пособие / Б. Н. Иванов. – М. : Лаборатория Базовых Знаний, 2001. – 288 с.

16 **Гаврилов, Г. П.** Задачи и упражнения по дискретной математике : учеб. пособие / Г. П. Гаврилов, А. А. Сапоженко. – 3-е изд., перераб. – М. : Физматлит, 2005. – 416 с.

17 **Ульянов, М. В.** Математическая логика и теория алгоритмов. Ч. 1 : Математическая логика / М. В. Ульянов, М. В. Шептунов – М. : МГАПИ, 2003. – 47 с.

18 **Ульянов, М. В.** Математическая логика и теория алгоритмов. Ч. 2 : Теория алгоритмов / М. В. Ульянов, М. В. Шептунов – М. : МГАПИ, 2003. – 80 с.

19 **Судоплатов, С. В.** Элементы дискретной математики : учебник / С. В. Судоплатов, Е. В. Овчинникова. – М. : ИНФРА-М ; Новосибирск : НГТУ, 2003. – 280 с.

20 **Редькин, Н. П.** Дискретная математика. Курс лекций для студентов-механиков : учеб. пособие / Н. П. Редькин. – СПб. : Лань, 2003. – 96 с.

21 **Ерусалимский, Я. М.** Дискретная математика. Теория, задачи, приложения / Я. М. Ерусалимский. – 5-е изд., перераб. и дополн. – М. :

Вузовская наука, 2002. – 268 с. : ил.

22 **Москинова, Г. И.** Дискретная математика. Математика для менеджера в примерах и задачах : учеб. пособие / Г. И. Москинова. – М. : Логос, 2002. – 240 с. : ил.