

Министерство образования и науки Российской Федерации
КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра вычислительных технологий

А.И. МИКОВ, О.Н. ЛАПИНА

ВЫЧИСЛИМОСТЬ И СЛОЖНОСТЬ АЛГОРИТМОВ

Учебное пособие

Краснодар
2013

УДК 681.3.07

ББК 32.973.26-018.2.75

М 594

Рецензенты:

Доктор физико-математических наук, профессор

С.В. Русаков

Доктор физико-математических наук, профессор

Е.А. Семенчин

Миков, А.И., Лапина О.Н.

М 594 Вычислимость и сложность алгоритмов: учеб. пособие /
А.И. Миков, О.Н. Лапина. Краснодар: Кубан. гос. ун-т,
2013. 79 с. 80 экз.

Предлагаемое учебное пособие рассматривает фундаментальные проблемы компьютерных наук – существование алгоритмов, решающих общую задачу, и оценки сложности алгоритмов. Приведены методы построения функций сложности, примеры оценок для различных алгоритмов.

Адресуется студентам, обучающимся по направлениям магистратуры и бакалавриата 010300 – Фундаментальные информатика и информационные технологии, 010400 – Прикладная математика и информатика и 010500 – Математическое обеспечение и администрирование информационных систем.

УДК 681.3.07

ББК 32.973.26-018.2.75

© Кубанский государственный
университет, 2013

© Миков А.И., Лапина О.Н., 2013

ВВЕДЕНИЕ

Современные компьютерные науки рассматривают очень широкий круг вопросов, связанных с разработкой и использованием вычислительных машин, компьютерных сетей и систем. Отдельные проблемы пришли из классической математики, к примеру, численные методы решения дифференциальных и иных уравнений. При решении других проблем, например, создания человека-машинного интерфейса, пока не используются глубокие математические результаты. Теория вычислимости и сложности алгоритмов выделяется в ряду проблем компьютерных наук как достаточно сложным математическим аппаратом, так и очевидной прикладной направленностью. Она рассматривает такие фундаментальные вопросы, как возможность или невозможность создания алгоритмов, решающих определенную задачу, эффективность или неэффективность алгоритмов, пределы возможного увеличения производительности компьютеров.

Предлагаемое издание призвано дать теоретические основы в области анализа алгоритмов и задач для самостоятельной практической работы, а также для последующих технологических дисциплин. Предполагается предварительное изучение студентами дисциплин «Основы программирования», «Архитектура вычислительных систем», «Дискретная математика», желателен сетевой курс.

В пособии использованы материалы журнальных публикаций последних лет, а также результаты, считающиеся уже классическими.

Текст учебного пособия может служить преподавателю основой для лекционного курса, а студенту – для самостоятельной подготовки и в качестве источника для формулирования тем квалификационных работ.

1. СЛОЖНОСТЬ АЛГОРИТМОВ С ОДНИМ ИСПОЛНИТЕЛЕМ

1.1. ПОНЯТИЕ СЛОЖНОСТИ АЛГОРИТМА

Традиционно в программировании понятие сложности алгоритма связано с использованием ресурсов компьютера: сколько процессорного времени требует программа для своего выполнения, как много памяти машины при этом расходуется? Учет памяти обычно ведется по объему данных. Время рассчитывается в относительных единицах так, чтобы эта оценка, по возможности, была одинаковой для различных машин.

Постоянная задача для программистов – сделать программу, работающую хотя бы немного быстрее и попытаться заставить ее работать в условиях ограниченной памяти. Для ряда областей ресурсы настолько критичны, что может возникнуть проблема целесообразности всего проекта из-за неэффективной работы программы. К таким областям относятся системы реального времени (*real-time systems*).

В данной главе будут рассмотрены две характеристики сложности алгоритма – времененная и емкостная. Не будем обсуждать логическую сложность разработки алгоритма. Не будем также обсуждать сложность текста алгоритма, так как она характеризует исполнителя, а не метод решения задачи.

Временную сложность будем подсчитывать в исполняемых командах: количество арифметических операций, сравнений, пересылок. *Емкостная сложность* будет определяться количеством скалярных переменных, элементов массивов, элементов записей или просто количеством байт.

В общем случае количество операций и требуемая память зависят от исходных данных, т.е. являются функциями исходных данных. Такие функции называются функциями сложности.

С точки зрения анализа сложности, сравнения алгоритмов, их классификации хотелось бы, чтобы функции сложности выражались с помощью обычных элементарных функций. Однако

это не всегда возможно, так как исходные данные могут быть нечисловыми (графы, строки символов, звуки и т.п.).

Поэтому сложность алгоритма α рассматривается как функция от некоторого числового параметра V , характеризующего исходные данные. Обозначим: $T_\alpha(V)$ – временная сложность алгоритма α ; $S_\alpha(V)$ – емкостная сложность.

Параметр V , характеризующий данные, называют объемом данных или *сложностью данных*. Выбор параметра V зависит не только от вида данных, но и от вида алгоритма или от задачи.

Рассмотрим примеры:

1. Задача вычисления факториала числа x ($x > 0$).

Программа итеративного решения задачи имеет следующий вид:

```
function Factorial (x: integer): integer;
var m, i: integer;
begin m := 1;
      for i := 2 to x do
        m := m*i;
      Factorial := m;
end.
```

Количество операций здесь подсчитать легко: один раз выполняется оператор присваивания: $m := 1$; тело цикла (умножение и присваивание) выполняется $(x - 1)$ раз; один раз выполняется присваивание $Factorial := m$. Если сложность каждой элементарной операции считать равной единице, то временная сложность приведенного алгоритма будет равна: $1 + 2(x - 1) + 1 = 2x$. Из этого анализа ясно, что за параметр V удобно принять x .

2. Задача отыскания скалярного произведения двух векторов $A = (a_1, a_2, \dots, a_k)$ и $B = (b_1, b_2, \dots, b_k)$. Вектор входных данных $X = (A, B)$. Стандартный алгоритм выполняет циклическое сложение попарных произведений компонент векторов. Количество операций будет пропорционально k , т.е. можно взять $V = k$ – сложность алгоритма зависит от количества компонент и не зависит от самих компонент векторов.

Эти примеры показывают, что для оценки сложности важны значения исходных данных и количество исходных данных. Второй пример иллюстрирует ситуацию, когда сложность алгоритма

может не зависеть от некоторых исходных данных. Собственно говоря, до анализа сложности алгоритма, *a priori*, не всегда можно сформулировать параметр V . И лишь после построения оценки становится ясно, какая характеристика является значимой для данного алгоритма.

Отыскание функций сложности алгоритмов важно как с прикладной, так и с теоретических точек зрения.

Задача минимизации временной сложности может быть поставлена так: среди всех алгоритмов, решающих задачу P , отыскать такой алгоритм α , для которого функция $T_\alpha(X)$ будет принимать минимальные значения на выбранном подмножестве S значений исходных данных ($X \in S$). Задача минимизации емкостной сложности возникает реже, в силу архитектурных особенностей современных ЭВМ. Программе может быть доступна практически неограниченная область памяти — виртуальная память. Недостаточное количество основной памяти приводит лишь к некоторому замедлению работы из-за обменов с дисков. Поэтому в дальнейшем будем рассматривать в основном временную сложность алгоритмов.

С теоретической точки зрения важным является вопрос, как далеко можно продвинуться по пути уменьшения сложности алгоритма, когда наступает предел, свойственный самой задаче, за которым дальнейшие попытки усовершенствовать алгоритм не приведут к успеху.

Другой вопрос связан с классификацией алгоритмов. Как быстро растет функция сложности алгоритма $T_\alpha(X)$, с ростом значений аргумента? Существуют алгоритмы с линейной зависимостью временной сложности; полиномиальной зависимостью. А существуют алгоритмы, сложность которых увеличивается быстрее любого полинома. Поэтому множество исследований посвящено вопросу: «возможен ли для данной задачи полиномиальный алгоритм либо нет?».

1.2. ФУНКЦИЯ СЛОЖНОСТИ ЦИКЛИЧЕСКИХ АЛГОРИТМОВ

Отыскание функции сложности производится на основе анализа текста алгоритма. Для определенности будем считать, что алгоритм записан на языке Паскаль и содержит арифметические операции, операции сравнения и пересылки скалярных данных.

Для построения функции сложности алгоритма необходимо подсчитать количество элементарных операций. Если алгоритм содержит простой линейный участок, то подсчитываем количество операций, необходимых для выполнения каждого оператора программы (находим вес оператора). Сложность линейного алгоритма будет равна сумме весов операторов. Если встречаются вызовы процедур, то вызов не рассматривается как одна операция, подсчитывается количество операций в процедуре плюс сам оператор вызова (с единичной сложностью).

Если алгоритм содержит *разветвления* (условные операторы *if*, *case* и т.п.), это означает, что вычислительный процесс в зависимости от исходных данных может направиться по одной из конечного числа ветвей. При вычислении функции сложности необходимо подсчитать сложность для каждой ветви. Тогда мы можем рассматривать либо худший случай, либо средний случай. Максимальное значение сложности будет сложностью алгоритма в худшем случае.

Для среднего случая, кроме веса ветвей нужно оценить еще и частоты ветвей. Под частотой ветви понимается отношение числа выполнений операторов ветви к общему числу выполнений программы. Тогда сложность программы будет вычисляться как сумма произведений весов ветвей на их частоты.

Если программа содержит *операторы цикла*, необходимо знать количество элементарных операций в теле цикла – вес цикла и количество повторений цикла. Если выражения, определяющие начальное и конечное значение параметра цикла – константы, мы можем подсчитать, сколько раз выполняется цикл, и вес цикла умножить на этот коэффициент. Проведем анализ алгоритма, содержащего вложенные *циклы for*.

Процедура MULTIPLY перемножает две квадратные $n \times n$ матрицы $A = \{a_{ij}\}$ и $B = \{b_{ij}\}$, получая матрицу $C = \{c_{ij}\}$.

```
procedure MULTIPLY (n: integer; a, b: matrix; var c: matrix);
var i, j, k: integer;
begin
  for i := 1 to n do
    for j := 1 to n do
      begin c[i, j] := 0;
      for k := 1 to n do
        c[i, j] := c[i, j] + a[i, k]*b[k, j];
      end;
  end.
```

Тип данных `matrix` определен вне процедуры как двумерный массив. Простой анализ текста показывает, что за параметр сложности данных V можно взять размер матриц n . Процедура представляет собой цикл по переменной i , тогда ее сложность

$$T_a(X) = n^* \text{ (сложность тела цикла 1).}$$

В свою очередь

$$\text{(сложность тела цикла 1)} = n^* \text{ (сложность тела цикла 2).}$$

Тело цикла 2 состоит из одного оператора присваивания и цикла 3. Таким образом

$$T_a(X) = n^* (n^* (1+n^* \text{ (сложность тела цикла 3)})).$$

Тело внутреннего цикла включает умножение, сложение и присваивание (3 операции) окончательно получаем

$$T_a(n) = n^* (n^* (1 + n^* 3)) = 3n^3 + n^2.$$

Здесь размер матрицы однозначно определяет сложность алгоритма. Полученный результат является одновременно и верхней и средней оценкой сложности.

В случае, если выражения, определяющие начальное и конечное значения параметра цикла `for` зависят от исходных данных, мы можем оценить количество повторений либо в худшем случае, либо в среднем. Рассмотрим пример процедуры.

```
for i := 1 to x do
  for j := i to x do
  begin
    тело цикла; end.
```

Входная переменная здесь принимает любое целое значение из промежутка $x \in [1 \div 5]$.

Тело цикла выполняется $x + (x-1) + (x-2) + \dots + 1 = \frac{x(x+1)}{2}$ раз.

Верхняя граница сложности равна

$$\max_x \left(\frac{x(x+1)}{2} \right) = \frac{5(5+1)}{2} = 15.$$

Среднее значение сложности, если все значения x равновероятны

$$\sum_{x=1}^5 (1/5)x(x+1)/2 = (1/10) \sum_{x=1}^5 x(x+1) = 7.$$

Циклы while и repeat, анализировать сложнее, так как условия повторения цикла вычисляются в теле самого цикла.

Рассмотрим в качестве примера вычисление вещественного значения корня $z = \sqrt[p]{x}$ методом Ньютона.

Задача сводится к отысканию корня уравнения $z^p - x = 0$. Корень отыскивается с помощью итерационного процесса

$$z_{n+1} = \frac{p-1}{p} z_n + \frac{x}{p z_n^{p-1}}.$$

Выбираем некоторое начальное значение z_0 . Окончание процесса происходит, когда корень найден с заданной точностью eps .

```
function P_root (p: integer; x, z0, eps : real) : real;
  var i: integer;
      y, zn, z: real;
  begin z:=z0;
    repeat
      zn:=z; y:=zn;
      for i:=2 to p-1 do y:=y*zn;
      z:=(p - 1)*zn/p + x/(p*y);
    until abs(z - zn) < eps;
  end. P_root :=z;
```

Для вычисления сложности алгоритма необходимо определить количество повторений цикла. Из теории численных мето-

дов известно, что требуемое количество итераций для достижения нужной точности $n \geq \log_2 \log_2 \varepsilon^{-1}$. Этот процесс сходится очень быстро. Уже при 5 итерациях достигается точность выше 10^{-9} .

Детальный анализ сложности алгоритма показывает, что не всегда количество элементарных операций, выполняемых алгоритмом на одном входе длины N , совпадает с количеством операций на другом входе такой же длины. Примерами таких алгоритмов могут служить ряд алгоритмов сортировки, алгоритмы поиска минимума и максимума в массиве.

Рассмотрим алгоритм поиска заданного элемента в векторе:

```
function Locate (data: integer): integer;
var i: integer;
    fl: boolean;
begin
    fl := false; i := 1;
    while (not fl) and (i <= N) do
        begin
            if A[i] = data then fl := true
            else i := i + 1;
        end;
    if (not fl) then i := 0;
    Locate := i;
end.
```

Если искомый элемент находится в конце списка, то программе придется выполнить N шагов – это наихудший случай. Наилучший случай будет, когда искомый элемент находится в списке на первой позиции. Тогда алгоритму придется сделать только один шаг. Оба эти случаи маловероятны.

Если же элементы списка беспорядочно смешаны, то искомый элемент может находиться в любом месте списка. Чтобы найти требуемый элемент, в среднем потребуется сделать $N/2$ сравнений.

Таким образом, говоря о сложности алгоритма, обычно рассматривают три разных варианта, особенности каждого из которых следует четко понимать.

Средняя сложность, или ожидаемая сложность, – среднее время работы алгоритма. При этом говорить о среднем можно только в предположении о существовании случайного распределения для входов программы. Обычно среднюю сложность считают в предположении, что все случаи равновероятны (например, в предыдущем примере поиска заданного элемента можно считать, что все возможные положения элемента могут появляться с равной вероятностью).

Максимальная сложность, называемая также сложностью в худшем случае, дает время, на котором алгоритм работает дольше всего.

Минимальная сложность, называемая также сложностью в лучшем случае, дает время для случая, на котором алгоритм работает быстрее всего. Этот критерий используется редко, чаще говорят о сложности в худшем случае или средней сложности.

1.3. ФУНКЦИЯ СЛОЖНОСТИ РЕКУРСИВНЫХ АЛГОРИТМОВ

Рекурсивный алгоритм – это такой алгоритм, в котором, по крайней мере, один из шагов формируется как обращение, к этому же алгоритму (прямая рекурсия), или к другому алгоритму, определяемому через первый (косвенная рекурсия).

Рассмотрим сначала случай прямой рекурсии с единственным рекурсивным вызовом в теле процедуры. Примером такого алгоритма является алгоритм Евклида вычисления наибольшего общего делителя.

Процедура вычисления НОД (GCD) для двух целых чисел m и n .

```
function GCD (n, m: integer): integer;
begin if m=0 then GCD := n
      else GCD := GCD (m, n mod m)
end.
```

В этой процедуре вектор исходных данных $X = (n, m)$. Текст процедуры содержит вызов процедуры с измененным по некоторому правилу вектором исходных данных $Y = f(X)$. Кроме этого, в теле содержится некоторое вычисление $h(X)$ и операции срав-

нения $c(X, S)$ значения X со значением S , при котором рекурсивный процесс должен заканчиваться.

Обозначим *точное* значение сложности с входными данными X через $T_a(X)$, тогда

$$T_a(X) = T_a(Y) + T_h(X) + T_c(X, S), \text{ или}$$

$$T_a(X) = T_a(f(X)) + T_f(X) + T_h(X) + T_c(X, S).$$

Второе равенство представляет собой рекуррентное уравнение для определения значений неизвестной функции $T_a(X)$ через известные значения $f(X)$, $T_f(X)$, $T_h(X)$, $T_c(X, S)$. Кроме этого имеется начальное значение $T_a(S)$ с известной функцией сложности вычисления (нерекурсивного) значения S . Таким образом, имеется система

$$T_a(X) = T_a(f(X)) + T_f(X) + T_h(X) + T_c(X, S);$$

$$T_a(S) = T_c(S, S) + T_s(S).$$

Это частный случай рекуррентного уравнения. Такие уравнения имеют развитую теорию решений. В некоторых случаях возможно аналитическое решение. Рассмотрим его на примере рекурсивного вычисления факториала F .

```
function F (x: integer): integer;
begin if x=1 then F := 1
else F := x * F(x - 1)
end.
```

Здесь $X = x$, $S = 1$, $f(X) = X - 1$, $T_f(X) = 1$, $T_h(X) = 2$, $T_s(X, S) = 1$, $T_s(S) = 1$.

Таким образом, система уравнений превращается в

$$T_a(x) = T_a(x - 1) + 4, \quad T_a(1) = 2.$$

Найдем ее решение:

$$T_a(1) = 2 = 4 * 1 - 2;$$

$$T_a(2) = T_a(1) + 4 = 2 + 4 = 4 * 2 - 2;$$

$$T_a(3) = T_a(2) + 4 = (2 + 4) + 4 = 4 * 3 - 2;$$

.....

$$T_a(x) = 4 * x - 2.$$

Верхняя оценка $T_a(X)$ может быть получена подобным образом, но с использованием рекуррентных неравенств.

$$T_\alpha(V) \leq T_\alpha(f(V)) + T_f(V) + T_h(V) + T_c(V, V_0);$$

$$T_\alpha(V_0) \leq T_c(V_0, V_0) + T_s(V_0).$$

Оценка среднего значения сложности требует учета вероятностных законов и может быть более трудной.

Рассмотрим теперь более общий случай рекурсии с несколькими рекурсивными вызовами. Эти вызовы могут иметь разные аргументы $Y1, Y2, \dots, Yk$, где $Y1 = f1(X), Y2 = f2(X), \dots, Yk = fk(X)$. Рекуррентное уравнение для функции сложности имеет вид

$$T_\alpha(X) = T_\alpha(f1(X)) + T_\alpha(f2(X)) + \dots + T_\alpha(fk(X)) + T_{f1}(X) + \\ + T_{f2}(X) + \dots + T_{fk}(X) + T_h(X) + T_c(X, S);$$

$$T_\alpha(S) = T_c(S, S) + T_s(S).$$

В реальных рекурсивных алгоритмах встречаются различные функции $fi(X)$. Однако обычно эти функции подчиняются определенным закономерностям. Рассмотрим наиболее часто встречающиеся случаи.

Одинаковые функции $f1(X) = f2(X) = \dots = fk(X)$. Рекуррентное уравнение для функции сложности имеет вид

$$T_\alpha(X) = k * T_\alpha(f(X)) + k * T_f(X) + T_h(X) + T_c(X, S);$$

$$T_\alpha(S) = T_c(S, S) + T_s(S).$$

Решение этого уравнения зависит от соотношения k и $f(X)$. Если $k > 1$, то значения $T_\alpha(X)$ быстро возрастают с ростом X , если функция $f(X)$ недостаточно быстро уменьшает X .

В качестве примера проведем анализ решения известной головоломки «Ханойская башня». Задача состоит в следующем.

Имеются три стержня a, b и c . На стержень a нанизаны n дисков уменьшающихся размеров. Внизу находится самый большой диск, а наверху – самый маленький. Необходимо перемещать диски со стержня на стержень по одному, так чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра. В итоге необходимо, чтобы все диски оказались на

стержне b . Стержень c можно использовать для временного хранения дисков.

С алгоритмической точки зрения элементарным шагом является перенос одного диска. Рассмотрим рекурсивное решение этой задачи.

Задачу перемещения n дисков со стержня a на стержень b можно представить состоящей из двух подзадач размера $n - 1$. Сначала нужно перенести $n - 1$ диск со стержня a на стержень c , оставив на стержне a один наибольший диск. Затем этот диск нужно переместить с диска a на b . Затем, $n - 1$ диск со стержня c переносится на стержень b . Далее решается задача с $n - 2$ дисками и т.д. Перемещения дисков выполняется с помощью рекурсии. Хотя фактические перемещения дисков не столь очевидны, применение рекурсии здесь вполне естественно.

Рассмотрим процедуру $\text{CARRY}(n, a, b, c)$, которая переносит башню высоты n со стержня a на стержень b , используя c как вспомогательный. Тогда

$\text{CARRY}(n-1, a, c, b)$ – перенести верхние $n-1$ дисков на c ;
 $\text{CARRY}(1, a, b, -)$ – перенести 1 нижний диск с a на b ;
 $\text{CARRY}(n-1, c, b, a)$ – перенести диски с c на большой диск, находящийся на b .

Вторая часть здесь тривиальная, выделим ее в отдельную нерекурсивную процедуру $\text{ONEDISK}(a, b)$. Теперь можно написать рекурсивную процедуру:

```
procedure CARRY(n, a, b, c: integer);
begin if n>1 then
begin
    CARRY(n-1, a, b, c);
    ONEDISK(a, b);
    CARRY(n-1, c, b, a);
end;
end.
```

Сложность процедуры ONEDISK – константа, обозначим ее h . Входной набор $X = (n, a, b, c)$. Очевидно, что a, b, c не влияют на сложность. Поэтому можно принять $V = n$.

Рекурсивное уравнение примет вид

$$\begin{cases} T_\alpha(n) = 2T_\alpha(n-1) + 2 + h + 1 \\ T_\alpha(1) = 1 + 0 \end{cases}$$

или

$$\begin{cases} T_\alpha(n) = 2T_\alpha(n-1) + h + 3 \\ T_\alpha(1) = 1. \end{cases}$$

Найдем решение этого уравнения. Выпишем значение функции сложности для $V=n-1$

$$T_\alpha(n-1) = 2T_\alpha(n-2) + h + 3.$$

Подставив в уравнение, получим

$$T_\alpha(n) = 2 * (2T_\alpha(n-2) + h + 3) + h + 3.$$

Затем

$$T_\alpha(n-2) = 2T_\alpha(n-3) + h + 3.$$

Тогда уравнение примет вид

$$T_\alpha(n) = 2 * (2 * (2T_\alpha(n-2) + h + 3) + h + 3) + h + 3$$

и т.д., т. е. очевидно решение этого уравнения имеет вид

$$T_\alpha(n) = 2^n v + w,$$

где v, w – параметры, которые нужно определить. Уравнение для их определения получаются из первоначальных рекуррентных соотношений:

$$\begin{cases} 2^n v + w = 2(2^{n-1} v + w) + h + 3 \\ T_\alpha(1) = 2v + w = 1 \end{cases};$$

или

$$\begin{cases} 2^n v + w = 2^n v + 2w + h + 3 \\ 2v + w = 1 \end{cases};$$

тогда

$$\begin{cases} w + h + 3 = 0 \\ v = \frac{1}{2}(1 - w); \end{cases}$$

$$\begin{cases} w = -(h+3) \\ v = \frac{1}{2}(1+h+3) = 2 + \frac{h}{2}. \end{cases}$$

Итого, получим аналитическое выражение для функции сложности

$$T_\alpha(n) = \left(2 + \frac{h}{2}\right)2^n - (h+3).$$

Найденная оценка сложности относится к классу экспоненциальных функций временной сложности.

Второй тип рекуррентных уравнений, которые часто встречаются в анализе алгоритмов, может быть задан следующими формулами:

$$\begin{cases} T_\alpha(n) = aT_\alpha(n/m) + b * n^k \\ T_\alpha(1) = b. \end{cases}$$

В том случае, когда не удается получить явное выражение в виде элементарных функций, обычно получают асимптотическое решение. Приведем здесь результаты асимптотического решения

$$T_\alpha(n) = O(n^k), \quad \text{если } a < m,$$

$$T_\alpha(n) = O(n^k \log_m n), \quad \text{если } a = m,$$

$$T_\alpha(n) = O(n^{\log_m a}), \quad \text{если } a > m.$$

Функции сложности, образующие прогрессию. Функции сложности имеют вид прогрессии

$$f_1(X) = X - 1; \quad f_2(X) = X - 2 \dots \quad f_k(X) = X - k.$$

Примером такой процедуры является рекурсивная процедура для нахождения n -го числа Фибоначчи. Проведем ее анализ.

function Fr (n: integer) : integer;

```
begin if n=1 or n=2 then Fr := 1
      else Fr := Fr(n-1) + Fr(n-2);
end.
```

Здесь $X = n$, $T_{f1}(n) = T_{f2}(X) = 1$, $T_h(n) = 2$, $T_c(n) = 2$, $T_s(n) = 1$.

Получим уравнение для функции сложности

$$T_\alpha(n) = T_\alpha(n-1) + T_\alpha(n-2) + 6,$$

$$T_\alpha(2) = 3,$$

$$T_\alpha(1) = 3.$$

Решающим параметром в оценке сложности рекурсивной процедуры будет количество вызовов функции. Алгоритм можно представить в виде дерева вызовов. На 1-м уровне находится одна вершина с параметром n ; на 2-м уровне – две вершины с параметрами $n-1$; на третьем – вершины с параметрами $n-2$ и т.д. Левая ветвь будет иметь вершины с параметрами $n, n-1, n-2, \dots, 2$, т.е. всего $n-1$ уровень. Правая ветвь – $n, n-2, n-4, n-6, \dots$. Можно сделать вывод, что количество уровней не меньше $n/2$.

Если дерево имеет k уровней, то в нем $1+2+4+8+\dots+2^{k-1}=2^k-1$ вершин. Для нашего дерева $n/2 < k < n-1$. Получаем $2^{n/2} < \text{число_вызовов} < 2^{n-1}-1$. Как верхняя, так и нижняя границы быстро растут с увеличением n . Уже при $n=20$ требуется не менее 1000 вызовов.

Анализ сложности взаимно рекурсивных алгоритмов.

Если две процедуры A и B связаны косвенной (взаимной) рекурсией, то для каждой процедуры записываем выражение функции сложности и объединяем их в общую систему уравнений. Одна из процедур имеет нерекурсивную ветвь (например, B). Эта ветвь и задает начальные условия.

В простом случае, когда имеем однократный вызов процедуры B в теле процедуры A и процедуры A в теле процедуры B , уравнения имеют вид

$$T_A(X) = T_B(f_A(X)) + T_{fA}(X) + T_{hA}(X),$$

$$T_B(X) = T_A(f_B(X)) + T_{fB}(X) + T_{hB}(X) + T_{cB}(X, S),$$

$$T_B(S) = T_{cB}(S, S) + T_{cB}(S).$$

Во втором уравнении выразим $T_A(f_B(X))$ из первого

$$T_A(f_B(X)) = T_B(f_A(f_B(X))) + T_{fA}(f_B(X)) + T_{hA}(f_B(X)).$$

Подставим полученное выражение во второе уравнение. Два уравнения будут сведены к одному уравнению для определения

ния $T_B(X)$. Третье уравнение остается в качестве начального условия.

Таким образом, задача определения функции сложности в случае косвенной рекурсии будет сведена к задаче для прямой рекурсии.

Контрольные вопросы и задания

1. Назовите основные характеристики сложности алгоритмов.
2. Что такое сложность данных?
3. Как определяется сложность линейного алгоритма?
4. Как вычислить сложность алгоритма, содержащего разветвления?
5. Как определить сложность алгоритма, содержащего циклы?
6. Какова сложность алгоритма перемножения двух квадратных матриц?
7. Какова сложность алгоритма вычисления вещественного значения корня методом Ньютона?
8. Что такое средняя сложность алгоритма, максимальная сложность алгоритма?
9. Какие уравнения применяются для нахождения функций сложности рекурсивных алгоритмов?
10. Какова сложность итеративного и рекурсивного алгоритмов вычисления факториала?
11. Какова сложность алгоритма решения головоломки «Ханойская башня»?
12. Как получить оценку сложности рекурсивного алгоритма нахождения n -го числа Фибоначчи?
13. Как определить сложность алгоритма, содержащего взаимную рекурсию?
14. Каковы основные типы асимптотического решения функций сложности рекурсивных алгоритмов?

Задачи для самостоятельного решения

1. Оцените сложность алгоритмов решения следующих задач:
 - 1.1. В массиве из n целых чисел найти все пары элементов, сумма которых четна, и сформировать новый массив из этих сумм.
 - 1.2. В массиве из n целых чисел найти все тройки элементов, в которых сумма двух элементов равна третьему, подсчитать количество таких троек.
 - 1.3. Упорядочить массив из n целых чисел таким образом, чтобы элементы с четными и нечетными значениями чередовались (пока имеются элементы разной четности).
 - 1.4. В массиве из n целых чисел найти все элементы, равные квадрату другого элемента массива и составить массив из этих элементов.
 - 1.5. Вычислить число размещений $A(n,m) = n!/(m!(n-m)!)$.
2. Постройте рекурсивный алгоритм поиска минимального элемента в массиве из n чисел. Оцените сложность алгоритма.
3. Постройте алгоритм вычисления функции $f(X) = n^n$ со сложностью порядка $O(\log n)$.
4. Напишите программу, печатающую n -е число Фибоначчи, которая имела бы линейную сложность.
5. Написать процедуру балансировки бинарного дерева и оценить его сложность.
6. Написать процедуру проверки того, является ли неориентированный граф самодополнительным, по его матричному представлению, и оценить ее сложность.

2. СЛОЖНОСТЬ АЛГОРИТМОВ С p ИСПОЛНИТЕЛЯМИ

2.1. Сложность параллельных вычислений

В том случае, когда задачу решает не один исполнитель, а p исполнителей, теоретически возможно ускорение решения задачи в p раз. Например, мы хотим сложить два вектора, каждый из которых задан n компонентами $n = kp$, $k \geq 1$, k – целое. Если сложение двух компонент требует одной единицы времени (с учетом чтения чисел из памяти и записи в память результата), то с одним исполнителем эта задача имеет сложность n , а с p исполнителями – сложность $n/p = k$.

Однако теоретически возможное ускорение – редкость. Для достижения такого предела необходимо, чтобы:

1) задача разбивалась на p независимых подзадач;

2) все подзадачи были одинаковой сложности (решались за одинаковое время).

В примере со сложением векторов существенно то, что значение k – целое. В противном случае (например, $k = 9,5$) не выполняется второе условие, на последнем шаге половина исполнителей будет простоять и ускорение будет несколько меньше, чем в p раз.

Если задача разбивается на *зависимые* подзадачи, то отклонение от теоретического предела становится еще большим.

Рассмотрим задачу сложения n чисел с помощью p процессоров, работающих с общей памятью. Время доступа любого процессора к любому слову в памяти будем считать одинаковым.

Далее, в таблице, приведено минимальное количество последовательных шагов, которое должна выполнить вычислительная система из p процессоров при сложении n чисел. Ясно, что на первом шаге для параллельного выполнения сложений больше $n/2$ процессоров не требуется. После выполнения первого шага остается $n/2$ слагаемых при четном n или $(n - 1)/2 + 1$ слагаемых при нечетном n . Такое уменьшение количества слагаемых происходит с каждым шагом до тех пор, пока не останется одно слагаемое – сумма. Поэтому ниже «диагонали», в которой числа за-

писаны жирным шрифтом, столбцы содержат одно и то же значение – минимальное для данного n количество шагов. Для чисел n , равных степени двойки, эти минимальные значения равны $\log n$ (см. таблицу).

p	Число слагаемых, n												
	2	3	4	5	6	7	8	9	10	11	12	13	
1	1	2	3	4	5	6	7	8	9	10	11	12	
2	1	2	2	3	3	4	4	5	5	6	6	7	
3	1	2	2	3	3	3	4	4	4	5	5	5	
4	1	2	2	3	3	3	3	4	4	4	4	5	
5	1	2	2	3	3	3	3	4	4	4	4	4	
6	1	2	2	3	3	3	3	4	4	4	4	4	
7	1	2	2	3	3	3	3	4	4	4	4	4	
8	1	2	2	3	3	3	3	4	4	4	4	4	

Если же процессоров меньше, чем $n/2$, то уменьшение количества слагаемых от шага к шагу происходит по несколько иной закономерности.

В целом функция сложности определяется следующими уравнениями:

$$T(n) = \begin{cases} T(n-p)+1 & \text{при } n \geq 2p, \\ \lceil \log n \rceil & \text{при } n \leq 2p. \end{cases}$$

Решение уравнений можно получить после некоторых преобразований. Поскольку первое уравнение определяет величины $T(n)$ не для всех значений n , то заменим его на уравнение

$$T(n+x) = T(n+x-p) + 1.$$

Тогда $T(2p+x) = T(p+x) + 1$; $T(kp+x) = T(p+x) + (k-1)$.

Здесь $n = kp+x$; $k = \left\lfloor \frac{n}{p} \right\rfloor$; $x = n - \left\lfloor \frac{n}{p} \right\rfloor p$.

В итоге получаем замкнутое выражение для функции сложности, справедливое для любого количества слагаемых и процессоров

$$T(n) = \left\lfloor \frac{n}{p} \right\rfloor - 1 + \left\lceil \log \left(p + n - \left\lfloor \frac{n}{p} \right\rfloor p \right) \right\rceil.$$

Как видим, в этой задаче максимальное число процессоров используется на первых шагах (или даже только на первом шаге), а на заключительных шагах часть процессоров пристаивают.

Для большинства решаемых задач характерна неполная загрузка процессоров, связанная с особенностями самой задачи. Эффектостояния процессоров отражается в таком понятии как *степень распараллеливаемости*. Программа вычисления содержит как участки параллельно выполняемых команд, так и участки последовательно выполняемых команд (из-за логических, причинно-следственных связей), которые невозможно распараллелить. И в целом время выполнения программы на машине с процессорами не может быть меньше, чем время выполнения самого длинного последовательного участка программы.

Этот факт отметил известный американский конструктор многопроцессорных ЭВМ Gene Amdahl (Джин Амдал), сформулировав в 1967 г. утверждение, позже названное законом Амдала. Математически его можно сформулировать так.

Пусть α – доля последовательных вычислений в программе, $s(p)$ – ускорение вычислений при использовании p процессоров вместо одного процессора. Тогда

$$s(p) = \frac{p}{1 + \alpha(p-1)}.$$

Иначе говоря, при идеальном распараллеливании $\alpha = 0$ и ускорение достигает максимально возможного значения $s(p) = p$. Для чисто последовательной программы $\alpha = 1$ и $s(p) = 1$. В промежуточных случаях ускорение меньше числа процессоров.

Закон имеет в основном идеенное значение, так как отыскание для конкретной задачи величины α может быть крайне затруднительно математическими средствами.

2.2. СЛОЖНОСТЬ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ

В модели распределенного решения задачи имеется несколько компьютеров, удаленных друг от друга (в географическом смысле) и решающих одну задачу. Эта модель имеет два варианта.

В первом варианте делается акцент на вычислительных мощностях: индивидуальная задача настолько сложна (с точки зрения необходимых вычислительных ресурсов), что пользователю недостаточно своего компьютера для решения задачи в приемлемое время. Тогда задача делится на подзадачи (модули) и каждый модуль обрабатывается на своем компьютере. Компьютеры соединены сетью, и время от времени обмениваются между собой промежуточной информацией, необходимой для скоординированного решения подзадач. Обмен должен быть относительно небольшим и нечастым, иначе медленная сеть сведет на нет преимущества распараллеливания. Такой подход позволяет решить проблему отсутствия суперкомпьютера у каждого исследователя.

Этот вариант используют для того, чтобы попытаться решить «всем миром» какую-нибудь сложную актуальную или просто много лет неподдающуюся задачу. В Интернете можно найти довольно много таких проектов, включая поиск внеземных цивилизаций. Конечно, действительно серьезные задачи, связанные с государственной или коммерческой тайной (проектирования новых образцов техники или вооружений, космических и ядерных исследований), таким способом не решаются. Для них остается обычный путь – использование суперкомпьютеров.

Однако для «рядовых» задач такой способ приемлем. Тем более что создать распределенный кластер значительно проще и дешевле, чем купить суперЭВМ. Поэтому наращивание вычислительных мощностей в последние годы (по крайней мере, в России) идет по пути «кластеризации». Однако достойных «рядовых» задач не так много. В связи с этим объявляются конкурсы «на задачи» для супервычислителей.

Итак, первый вариант только технологически отличается от традиционного подхода. Его можно назвать *распределенным решением задач*.

Во втором подходе акцент делается на том, что задача по сути своей не может быть решена в одной точке пространства. Простым примером является задача информационного поиска. Предположим, что для поисковой программы доступны несколько баз данных (БД), принадлежащих разным организациям. Они, разумеется, находятся на разных компьютерах, возможно, в разных

странах. Информация в разных БД частично пересекается. Поисковый запрос сформулирован таким образом, что 1) неизвестно, в какой из БД искать необходимые данные, 2) возможно, с помощью одной БД не будет получен полный ответ и потребуется дополнительное обращение к некоторым другим БД.

Это нетривиальная задача. Нельзя все БД перебросить по сети в одну точку, чтобы организовать там процесс решения. Запрос для каждой отдельно взятой БД может быть «незаконным», поскольку не согласуется с ее схемой и содержит атрибуты, в данной БД неопределенные. В то же время задача потенциально разрешима: необходимые для запроса данные действительно есть в этом множестве баз данных (часть данных в одной БД, другая часть – в другой БД, третья часть – в третьей БД).

Такая задача является *существенно распределенной*. Она не может быть «стянута в одну точку» как задачи из первого варианта.

Распределенные структуры данных. Распределенные структуры данных состоят из элементов, связанных между собой двумя типами связей: логическими связями и топологическими (геометрическими) связями.

Примеры логических связей. В массиве элементы $A[i]$ и $A[i+1]$ связаны как предыдущий и последующий или как элементы, индексы которых отличаются на единицу. В строке соседние символы связаны как предыдущий и последующий. В записи (record) логические связи слабее: порядок следования элементов не имеет значения, они связаны только принадлежностью одной и той же записи. То же самое можно сказать об элементах абстрактного множества.

Топологические связи определяются взаимным расположением элементов структуры данных в вычислительной сети. Элементы могут находиться на одном сайте (узле), на соседних сайтах, на соседних с соседними и т.д. Иногда важно учитывать числовое расстояние между сайтами, иногда (например, в ГИС) – координаты сайтов, т.е. связи могут быть и геометрическими.

Логические и топологические связи могут быть согласованными: логически соседние элементы являются и топологически соседними. Это возможно, если логическая структура данных изоморфна структуре вычислительной сети. Возможна и обратная ситуация: логически близкие элементы топологически удалены. Промежуточная ситуация: логическое и топологическое расстояния никак не связаны между собой, т.е. элементы структуры данных случайным образом разбросаны по сайтам вычислительной сети.

Согласованность или не согласованность сами по себе не являются ни хорошими, ни плохими свойствами распределения данных. Все зависит от задачи, которую нужно решать, и от алгоритма решения этой задачи.

Основной оценкой эффективности является время решения задачи. Оно «складывается» из процессорного времени на сайтах вычислительной сети и времени передачи данных по каналам связи между сайтами. На самом деле «сложение» здесь относительное, поскольку многие процессы протекают параллельно. В этом сложность аналитической оценки эффективности. Аналитическую оценку производить, конечно, нужно. Но нужен и последующий вычислительный эксперимент, позволяющий оценить реальные затраты времени на решение задачи.

Одним из этапов разработки распределенного алгоритма является построение динамической распределенной структуры данных. Необходимо задать, как исходные данные будут распределены по узлам вычислительной сети, как и когда элементы или блоки элементов данных будут перемещаться с одних узлов на другие, какой будет заключительная конфигурация распределенной структуры данных.

Другой этап состоит в том, чтобы определить, какие вычислительные процессы будут происходить на каждом из узлов.

Третий этап состоит в том, чтобы определить систему сообщений между узлами (сайтами), позволяющую инициировать отдельные процессы и завершить работу алгоритма в целом правильным результатом.

Многие задачи в распределенных системах решаются путем пересылки сообщений согласно некоторой схеме, которая гарантирует участие всех сайтов. Эта схема зависит от топологических особенностей системы, т.е. от вида графов, представляющих связи между узлами-сайтами.

- **Expand(R):**

- **begin**

- **if** $Out(R) \not\subset R$ **then**

- **begin** $Out(R) \Rightarrow R;$

- **Expand(R)**

- **end;**

- **end.**

Первый вызов – $\text{Expand}(Out(\text{start}))$. Предполагается, что $Out(\emptyset) = \emptyset$, $\emptyset \subseteq \emptyset$.

Любой волновой алгоритм может использоваться как PIF-алгоритм. Действительно, пусть A – волновой алгоритм. Чтобы использовать A как PIF-алгоритм, возьмем в качестве процессов, изначально знающих M , инициаторов A . Информация M добавляется к каждому сообщению A . Это возможно, поскольку по построению инициаторы A знают M изначально, а пассивные сайты не посыпают сообщений, пока не получат хотя бы одно сообщение, т.е. пока не получат M . Событию $return(\text{OK})$ в волне предшествуют события в каждом процессе; следовательно, когда оно происходит, каждый процесс знает M , и событие $return(\text{OK})$ можно считать требуемым событием $notify$ в PIF-алгоритме.

Алгоритмом *обхода* называется алгоритм, обладающий следующими тремя свойствами.

1. В каждом вычислении – один сайт-инициатор, который начинает выполнение алгоритма, посыпая ровно одно сообщение.

2. Процесс сайта, получая сообщение, либо посыпает одно сообщение дальше, либо выполняет процедуру $return(\text{OK})$.

3. Алгоритм завершается в инициаторе и к тому времени, когда это происходит, процесс каждого сайта посыпает сообщение хотя бы один раз.

Из первых двух свойств следует, что в каждом конечном вычислении решение принимает один процесс. Говорят, что алгоритм завершается в этом процессе.

Действия в алгоритме для гиперкуба.

Для инициатора (выполняется один раз):

out (Request, 1) **via** ребро ($n - 1$)

Для каждого процесса при получении маркера (Request, k):

begin if $k=2^n$ **then return(OK)**

else begin (*пусть l – наибольший номер,
такой, что $2^l | k$ *)
out (Request, $k+1$) **via** ребро l
end;

end.

В этом алгоритме решение принимается после 2^n пересылок маркера.

Алгоритм выбора сайта. Во многих распределенных системах один из сайтов играет роль координатора при выполнении распределенного алгоритма. Иногда координатором является сайт, который инициировал выполнение алгоритма. Но не всегда один и тот же сайт на протяжении всего вычисления может оставаться координатором. Можно привести следующие причины смены координатора.

1. Выход из строя оборудования на сайте-координаторе, в связи с чем этот сайт не может продолжать управлять процессами в распределенной системе.

2. Метод координации, реализованный на сайте, оказался неэффективным.

3. Изменения в интенсивности использования различных сайтов системы, приводящие к тому, что координация из прежнего места становится менее эффективной, чем координация с других сайтов.

В подобных случаях требуется выбор нового координатора. Выбор среди сайтов нужно проводить и тогда, когда должен быть выполнен централизованный алгоритм и не существует заранее известного кандидата на роль инициатора алгоритма. Например,

в случае процедуры инициализации системы, которая должна быть выполнена в начале или после сбоя системы. Так как множество активных сайтов может быть неизвестно заранее, то невозможно назначить один сайт раз и навсегда на роль лидера.

Активные (функционирующие в данный момент) сайты должны самоорганизоваться и провести выборы координатора. Для выбора нужна какая-то информация о кандидатах. Для простоты будем считать, что с каждым сайтом связано некоторое число – оценка *est* (estimate). Выбран должен быть сайт с максимальным (или в отдельных задачах – минимальным) значением оценки.

Максимальное значение нетрудно найти, если все локальные оценки собрать в одном месте. Но сложность заключается в том, что неясно, кто бы мог взять на себя сбор информации, а также зависимость процедуры сбора от архитектуры системы.

Если топология распределенной системы – дерево или доступно оствовное дерево системы, выбор можно провести с помощью приведенного далее алгоритма. В этом алгоритме требуется, чтобы все концевые вершины были инициаторами алгоритма. Чтобы преобразовать алгоритм на случай, когда некоторые сайты также являются инициаторами, добавляется фаза *wake-up*. Сайты, которые хотят начать выборы, рассылают сообщение **<wakeup>** всем другим сайтам.

Когда сайт получит сообщение **<wakeup>** через каждый канал, он начинает выполнять алгоритм обхода (см. Миков А.И. Распределенные компьютерные системы и алгоритмы. 2009 г.), который расширен таким образом, чтобы вычислять идентификатор сайта с наибольшей оценкой, и чтобы каждый сайт выполнял процедуру *return(OK)*. Когда сайт выполняет эту процедуру, он знает идентификатор координатора; если этот идентификатор совпадает с идентификатором процесса, он становится *координатором*, а если нет – *проигравшим*.

В тексте алгоритма логическая переменная *sent* («отправлено») используется, чтобы каждый сайт послал сообщение **<wakeup>** не более одного раза, а переменная *counter* (счетчик) используется для подсчета количества сообщений **<wakeup>**, полученных сайтом.

```

var is_sent : boolean init false ;
counter: integer init 0 ;
recp[q] : boolean для всех q ∈ Out(this) init false ;
m : integer init this ;
state : (sleep, coordinator, lost) init sleep ;
begin if this - инициатор then
    begin is_sent := true ;
        forall q ∈ Out(this) do send <wakeup> to q
    end ;
    while counter < card(Out(this)) do
        begin receive <wakeup> ; counter := counter + 1 ;
            if not is_sent then
                begin is_sent := true ;
                    forall q ∈ Out(this) do send <wakeup> to q
                end ;
            end ;
        (* Начало алгоритма обхода *)
        while card{q : ¬recp[q]} > 1 do
            begin receive (token, r) from q ; recp[q] := true ;
                if est(r) > est(m) then m := r
            end ;
            send (token, m) to q0 with ¬recp[q0] ;
            receive (token, r) from q0 ;
            if est(r) > est(m) then m := r; (* return(OK) с ответом m *)
            if m = this then state := coordinator else state := lost ;
            forall q ∈ Out(this), q ≠ q0 do send (token, m) to q
        end.

```

Когда хотя бы один сайт инициирует выполнение алгоритма, все сайты посылают сообщения <wakeup> всем своим соседям, и каждый сайт начинает выполнение алгоритма для дерева после получения сообщения <wakeup> от каждого соседа. Все процессы завершают алгоритм для дерева с одним и тем же значением оценки, а именно, с наибольшей оценкой сайта. Единственный

сайт с такой оценкой закончит выполнение в состоянии *координатор*, а все остальные сайты – в состоянии *проигравший*.

Оценим сложность приведенного распределенного алгоритма.

Через каждый канал пересылаются по два сообщения **<wakeup>** и по два сообщения **<tok, r>**, откуда сложность сообщений равна $4N - 4$, где N – количество узлов в распределенной системе. В течение D единиц времени (здесь D – диаметр графа системы) после того, как первый процесс начал алгоритм, каждый процесс послал сообщения **<wakeup>**, следовательно, в течение $D + 1$ единиц времени каждый процесс начал волну. Легко заметить, что первое решение принимается не позднее, чем через D единиц времени после начала волны, а последнее решение принимается не позднее D единиц времени после первого, откуда полное время равно $3D + 1$.

Если порядок сообщений в канале может быть изменен (т.е. канал – не FIFO), процесс может получить сообщение **(token, r)** от соседа, *прежде чем* он получил сообщение **<wakeup>** от этого соседа. В этом случае сообщение **(token, r)** может быть временно сохранено или обработано как сообщения **(token, r)**, прибывающие позднее.

Контрольные вопросы и задания

1. Что такое независимые подзадачи?
2. Сколько чисел складывается на третьем шаге в задаче сложения n чисел с помощью p процессоров? Как это количество зависит от значений n и p ?
3. Сколько процессоров используется на третьем шаге в задаче сложения n чисел с помощью p процессоров? Как это количество зависит от значений n и p ?
4. Какое количество слагаемых остается к началу четвертого шага в задаче сложения n чисел с помощью p процессоров? Как это количество зависит от значений n и p ?
5. Как оценить долю α последовательных вычислений в задаче вычисления скалярного произведения векторов?

6. Что такое распределенная структура данных, чем отличаются логические связи от топологических?
7. Какими свойствами обладает алгоритм обхода?
8. В чем заключается метод поиска в ширину?
9. В чем заключается метод поиска в глубину?
10. Какова сложность алгоритма обхода, приведенного в параграфе 2.2?
11. Сколько связей имеется в архитектуре распределенной системы в виде гиперкуба?
12. В чем различие между алгоритмами обхода и алгоритмами выбора?
13. Какова сложность алгоритма выбора, приведенного в параграфе 2.2?
14. Почему в оценке сложности используется величина диаметра графа распределенной системы?
15. Что такое «остовное дерево»?
16. Чем характеризуется дисциплина FIFO?

Задачи для самостоятельного решения

1. Выведите формулу количества используемых процессоров в зависимости от номера шага в задаче суммирования n чисел с помощью p процессоров.
2. Оцените сложность перемножения квадратных матриц размером $n \times n$ на многопроцессорной ЭВМ с p процессорами и общей памятью.
3. Оцените сложность решения системы линейных алгебраических уравнений методом Гаусса на многопроцессорной ЭВМ с p процессорами и общей памятью.
4. Оцените сложность решения системы линейных алгебраических уравнений методом простой итерации на многопроцессорной ЭВМ с p процессорами и общей памятью.
5. Разработайте параллельный алгоритм, проверяющий изоморфны ли два графа. Графы заданы матрицами смежностей вершин, располагающимися в общей памяти. Предполагается, что вычислительная система содержит p универсальных процес-

солов, работающих независимо. Проведите оценку сложности алгоритма.

6. Разработайте параллельный алгоритм, проверяющий изоморфны ли два графа. Графы заданы матрицами смежностей вершин, располагающимися в общей памяти. Вычисления производятся массивом GPU, т.е. одновременно выполняется одна и та же команда над разными наборами данных. Оцените сложность разработанного алгоритма.

7. Разработайте распределенный алгоритм, вычисляющий максимальную степень вершины графа – узла распределенной вычислительной системы. Этот алгоритм относится к классу *self-aware*, т.е. исследуется граф самой вычислительной системы. Результат вычисления должен быть сообщен всем узлам распределенной системы. Оцените сложность алгоритма в зависимости от количества узлов в системе.

8. Разработайте распределенный алгоритм, вычисляющий диаметр графа – максимальное из расстояний между парами вершин – узлов распределенной вычислительной системы. Граф изображает структуру этой системы.

9. Рассмотрите алгоритм вычисления диаметра в виде последовательности шагов:

1. Выберите произвольную вершину в качестве корневой.
2. Выполните шириновидную обходу от корневой вершины.
3. Выберите вершину с наибольшим расстоянием от корневой.
4. Выполните шириновидную обходу от вершины, выбранной на предыдущем шаге.
5. Выберите вершину с наибольшим расстоянием от корневой.
6. Выполните шириновидную обходу от вершины, выбранной на предыдущем шаге.
7. Итак, диаметр графа равен количеству шагов, необходимых для достижения вершины, в которой не может быть выбрана новая вершина для дальнейшего обхода.

3. СЛОЖНОСТЬ ЗАДАЧ

3.1. ПОНЯТИЕ СЛОЖНОСТИ ЗАДАЧИ

В теории вычислений задачи подразделяют на *общие (массовые)* и *частные (индивидуальные)*. Индивидуальная задача требует конкретного ответа, выраженного, как правило, простыми типами данных – логическими (ответ да или нет), числовыми или простыми структурами данных – массивами и записями.

Индивидуальная задача «решить уравнение $5 \times x + 8 = 33$ » имеет решение в виде числа 5. Индивидуальная задача «решить уравнение $x^2 + 1,2 \times x + 0,8 = 24,1$ » имеет решение в виде двух чисел – записи. Задача решения конкретного дифференциального или интегрального уравнения требует представления решения в виде строки символов, если возможно аналитическое решение, или в виде массива значений искомой функции, если применяются численные методы.

Массовая задача включает *параметры* – переменные. Предполагается, что значения этих параметров могут выбираться из некоторой предметной области, поэтому они называются исходными данными. Массовая задача превращается в индивидуальную задачу при придаании параметрам конкретных значений. Одна и та же массовая задача может породить множество индивидуальных задач. Еще и поэтому массовую задачу называют общей: она порождает множество частных задач. Массовая задача обычно требует отыскания решения в виде алгоритма или еще более сложной конструкции.

Простая массовая задача «решить уравнение $a \times x + b = c$ » имеет решение в виде строки символов $x = (b - c)/a$. Но и эта строка может рассматриваться как описание последовательного алгоритма, который необходимо выполнить для нахождения решения индивидуальной задачи при получении конкретных значений параметров a, b, c .

Решение более сложных массовых задач чаще всего невозможно записать в виде конечной строки символов – формулы,

вычисления по которой производятся последовательно (без возвратов).

Для любой разрешимой задачи существует множество различных алгоритмов, ее решающих. Сложность самого простого из таких алгоритмов можно считать *сложностью задачи*. Если как мы видели, оценить сложность алгоритма иногда бывает весьма непросто, то оценить сложность задачи еще труднее. Для этого нужно найти оптимальный алгоритм, решающий задачу, и доказать его оптимальность.

В качестве примера можно упомянуть задачу перемножения двух квадратных матриц размера $n \times n$ на однопроцессорной машине. Общеизвестный алгоритм имеет сложность $O(n^3)$. Фолькером Штрассеном в 1969 г. был предложен рекурсивный алгоритм сложности $O(n^{2.81})$. В 1979 г. Д. Бини с коллегами предложили алгоритм с оценкой $O(n^{2.78})$. А. Шёнхаге в 1981 г. представил алгоритм с оценкой сложности $O(n^{2.70})$. Значительно уменьшили до $O(n^{2.376})$ оценку Д. Копперсмит и С. Виноград в 1990 г. Асимптотически еще более быстрый алгоритм, $O(n^{2.373})$, разработан в 2011 г. Вирджинией Василевской-Вильямс. Скорее всего, это не предел, и оптимальный алгоритм пока не известен, следовательно, не известна и сложность задачи перемножения матриц.

С практической точки зрения более важным является разделение задач на классы – «легкие задачи» и «трудные задачи». Границу между ними можно проводить по-разному, но все сошлись во мнении, что задачи, для решения которых требуется выполнить $O(n)$, $O(n^2)$, $O(n^3)$, ... операций – это легкие задачи. Задачи же с оценкой сложности $O(2^n)$ и более – сложные. Первую группу задач называют задачами *полиномиальной* сложности, поскольку их времененная сложность ограничивается сверху некоторым полиномом (быть может, очень большой, но конечной степени). Обозначим множество таких задач **P**. Вторую группу называют задачами *экспоненциальной* сложности, поскольку в общем случае (т.е. для исходных данных, наиболее «неудобных» для любого из алгоритмов, решающих задачу) требуется количество операций, увеличивающееся с ростом n по крайней мере экспоненциально. Обозначим множество этих задач **EXP**.

Понятие сведения одной задачи к другой. Положим, у нас есть решающий некоторую задачу алгоритм; можно ли его использовать для решения другой задачи, а если можно, то как? Как при этом связаны сложности этих двух задач? В качестве примера такого сведения можно привести задачи вычисления обратной матрицы и решения системы линейных уравнений. Алгоритм решения первой задачи может быть использован для решения второй задачи, но при этом потребуются некоторые дополнительные вычисления. Как известно, обе задачи имеют полиномиальную сложность. Сведение одной задачи к другой – это тоже задача! И в данном случае она также имеет полиномиальную сложность. Эту ситуацию обозначают следующей фразой: первая задача *полиномиально сводима* ко второй задаче.

Будем говорить, что задача B сводится к задаче A за полиномиальное время, если существует детерминированный полиномиальный алгоритм, преобразующий произвольный частный случай задачи B (частную задачу, полученную подстановкой значений вместо параметров общей задачи B) в некоторый частный случай задачи A , и второй детерминированный полиномиальный алгоритм, преобразующий решение задачи A в решение задачи B .

Понятие *переборной задачи* (точнее задачи, решаемой методом полного перебора). Положим, у нас имеется множество S из n элементов. Нам требуется отыскать какое-либо его подмножество, обладающее определенным свойством. Это можно сформулировать так: дан предикат (утверждение) $P(A)$, определенный на множестве 2^S всех подмножеств множества S , т.е. $A \in 2^S$, истинный на одних подмножествах (быть может, только на одном) и ложный на других (быть может, на всех). Пример: множество S состоит из n различных целых чисел $\{1, 3, 4, 7, 8, 14, 20, 23, 25, 28\}$; предикат $P(A) = \{\text{множество } A \text{ содержит только четные числа}\}$. В этом случае $P(\{3, 7, 14\}) = \text{false}$, $P(\{4, 20, 28\}) = \text{true}$.

Утверждение P называют еще *свойством* множества A .

Для конкретного множества A легко выяснить, обладает ли оно свойством P , но найти множество, обладающее свойством P , зачастую можно только, проверив для всех $A \in 2^S$, выполняется ли $P(A)$, $P(A) = ?$ **true**. Например, для упомянутого множества S целых чисел можно написать уравнение $P_1(X) = \text{true}$, где $P_1(X) =$

= {сумма чисел, входящих в X , равна 32}. Это известная задача о рюкзаке, в которой требуется найти подмножество заданного множества натуральных чисел, причем сумма чисел, входящих в подмножество, должна быть равна другому заданному числу m . Найти решение этого уравнения, т.е. найти подходящее множество X можно в общем случае только перебором и проверкой вариантов. Каждое подмножество может быть описано характеристической функцией $c: S \rightarrow \{0, 1\}$ так, что $c_i = 1$, если i -й элемент множества S принадлежит X и $c_i = 0$ в противном случае. Для решения задачи в худшем случае перебрать нужно все варианты, а их 2^n . В настоящее время не известно никакого способа вычисления X (под термином «вычисление» мы понимаем альтернативу перебору).

Теперь можно дать более формальное определение задачи, решаемой методом перебора.

Переборная задача – это задача с параметром сложности исходных данных n , решение которой может быть выражено в виде совокупности значений c_1, c_2, \dots, c_n , параметров x_1, x_2, \dots, x_n , каждый из которых может принимать только конечное число значений, причем формулировка задачи содержит явное указание на то, как вычислить (проверить), является ли данный набор $\{c_i\}$ удовлетворительным (допустимым) решением задачи, но не известно (или не существует) никакого способа вычислить значения $\{c_i\}$ по формуле или с помощью алгоритма полиномиальной сложности.

Разумеется, переборная задача имеет экспоненциальную сложность при вычислениях на однопроцессорной машине.

Понятие задачи распознавания. Задачи обычно формулируются так: дан набор исходных данных X ; требуется найти результат Y , удовлетворяющий условиям задачи. Для некоторых задач в качестве результата удобно рассматривать только значения «да» и «нет». Такие задачи формулируются как вопрос. Например, задача синтаксического анализа текста программы X : «Соответствует ли программа X правилам языка Паскаль?». Множество всех возможных программ L разделяется на два подмножества L_{yes} (множество всех правильных программ, ответ «да») и L_{no} (множество всех остальных программ, ответ «нет»). Алгоритм, решаю-

щий задачу, т.е. отвечающий «да» или «нет», по-существу распознает, принадлежит ли программа X множеству L_{yes} . То же можно сказать и о других задачах с двумя возможными ответами. Поэтому такие задачи называются задачами распознавания, а точнее – *задачами распознавания свойств*. Имеется в виду свойство, которым обладает (или не обладает) вход X .

С точки зрения математической логики задаче распознавания свойств соответствует предикат $P(X)$.

Если, формулируя задачу, мы сразу настраиваемся на то, что она будет решаться алгоритмическим способом с помощью некоторого вычислителя (а не с помощью, например, озарения), то мы должны договориться, что исходные данные X будут задаваться на языке, «понятном» вычислителю. Пусть A – алфавит символов, воспринимаемых вычислителем, $L = A^*$ – множество всех возможных цепочек (слов) в алфавите A . Тогда L_{yes} – множество слов, кодирующих вход X , для которых вычислитель выдает ответ «да». Поскольку в теории формальных языков множество $L_{yes} \subset L$ называется языком, то любую задачу распознавания можно сформулировать как задачу *распознавания языка*.

Для задач распознавания понятие полиномиальной сводимости формулируется чуть проще: задача B сводится к задаче A за полиномиальное время, если существует детерминированный полиномиальный алгоритм, преобразующий произвольный частный случай задачи B в некоторый частный случай задачи A так, что ответом для данного случая задачи B является «да» тогда и только тогда, когда ответом на соответствующий случай задачи A также является «да».

Два понятия, относящиеся к процессу решения задачи: детерминированное вычисление и недетерминированное вычисление. Детерминированное вычисление – обычный, классический способ (это понятие входит в определение алгоритма как одно из свойств). Недетерминированное вычисление имеет более общий характер: обычно оно ведет себя как детерминированное, но на некоторых шагах вдруг «размножается» и начинает существовать в двух или более параллельно работающих экземплярах, т.е. имитирует исполнение на многопроцессорной вычислительной машине с неограниченным количеством процессоров. Ясно, что

распараллеливание работы может уменьшать время вычислений. Но насколько? Может ли распараллеливание перевести задачу из одного класса сложности в другой? Ответ не очевиден. Возьмем в качестве примера задачу поиска элемента в бинарном дереве, причем в множестве ключей поиска не определено никакого отношения порядка. Это означает, что в худшем случае нужно посетить все n вершин. Если просмотр распараллелить, начиная с корня, одновременно просматривая левое и правое поддеревья, а внутри каждого из поддеревьев процессы вновь распараллелить и т.д., перемещаясь по ссылкам к концевым вершинам, то нам потребуется время, пропорциональное $\log n$, и количество процессоров, пропорциональное n . Таким образом, соотношение времен детерминированного и недетерминированного вычислений такое же, как и соотношение экспоненциального и линейного алгоритмов (если считать параметром сложности исходных данных количество уровней дерева), т.е. при распараллеливании задача переходит из одного класса в другой. Однако известно, что существуют плохо распараллеливаемые задачи, для которых увеличение количества процессоров сверх некоторой константы не приводит к уменьшению времени вычислений. Такие задачи, очевидно, останутся в своем классе при изменении способа вычисления.

3.2. КЛАССЫ СЛОЖНОСТИ

Кроме введенных выше классов сложности задач P и EXP в теории рассматривается еще несколько важных классов.

Первый класс задач – NP . Это класс задач, которые можно решить за полиномиальное время (P), но на машине, более мощной, чем обычная однопроцессорная машина, – на недетерминированном (N) вычислителе. Недетерминированный вычислитель содержит неопределенное количество процессоров (т.е. столько, сколько может понадобиться).

Отличие от p -процессорной машины, например, для рассмотренной ранее задачи суммирования чисел, состоит в том, что для недетерминированного вычислителя никогда не выполняется не-

равенство $n \geq 2p$, поэтому функция сложности всегда логарифмическая.

Недетерминированный вычислитель исполняет так называемые недетерминированные алгоритмы.

Недетерминированный алгоритм лишен свойства детерминированности обычных алгоритмов. Разрешаются шаги с неоднозначным результатом – шаги, вырабатывающие сразу несколько значений для одной и той же переменной. Для этих значений далее могут быть созданы несколько параллельно выполняющихся процессов (по процессу для каждого значения) для продолжения вычислений.

Важным примером задачи из класса NP является задача о выполнимости формулы пропозиционального исчисления (раздела математической логики).

Рассмотрим детальнее проблему выполнимости, так как она была одной из первых проблем в исследовании класса NP и, кроме того, любая задача может быть сформулирована либо как утверждение, истинность которого нужно проверить, либо как предикат (утверждение, включающее свободные переменные), для которого нужно найти значения свободных переменных, дающие получающемуся из предиката утверждению значение «истина». Поскольку пропозициональное исчисление близко тому, что в курсе «Дискретная математика» называется булевой алгеброй, мы рассмотрим эту задачу в терминах булевой алгебры.

Пусть $\Phi(x_1, x_2, \dots, x_n)$ – формула булевой алгебры, представленная в конъюнктивной нормальной форме (КНФ)

$$\Phi(x_1, x_2, \dots, x_n) = \& D_i, D_i = \vee x_j^{\sigma_j} \text{ } (\sigma_j = 0 \text{ или } 1).$$

Часть формулы, обозначенная D_i , называется дизъюнктом, а часть формулы, записанная как $x_j^{\sigma_j}$, называется литералом. Таким образом, литерал – это переменная или ее отрицание. В дизъюнкт могут входить только некоторые из n переменных (не обязательно все). Пример КНФ:

$$(x_2 \vee \neg x_4) \& (x_1 \vee \neg x_3 \vee x_5) \& (x_1) \& (x_3 \vee x_4).$$

Здесь знак \neg обозначает отрицание, т.е. $\neg x = x^0$.

Длина дизъюнкта – количество литералов в дизъюнкте. В приведенном примере имеются 4 дизъюнкта с длинами 2, 3, 1, 2.

Булева формула (выражение) называется *выполнимой*, если существует хотя бы один набор значений $\{\alpha_i\}$ такой, что $\Phi(\alpha_1, \alpha_2, \dots, \alpha_n) = 1$. Решение задачи о выполнимости состоит в ответе на вопрос, является ли заданное в форме КНФ выражение выполнимым.

С алгоритмической точки зрения для решения задачи нужно построить процедуру, которая воспринимала бы на входе строку символов длиной n – слово в алфавите $\{\&, \vee, (,), x_1, x_2, \dots, x_n\}$, являющееся записью выражения E , и по окончании работы выдавала один из двух ответов: «выполнимо» или «не выполнимо». Сложность такой процедуры является функцией сложности исходных данных n .

Теорема. Задача о выполнимости формулы пропозиционального исчисления, заданной в конъюнктивной нормальной форме, принадлежит классу **NP**.

Доказательство проводится конструктивно, через построение алгоритма для недетерминированной машины

for $i := 1$ **to** n **do** $x_i :=$ *выбор*($\{0, 1\}$);
if $\Phi(x_1, x_2, \dots, x_n)$ **then** успех **else** неудача.

Этот алгоритм работает следующим образом. Многозначная функция *выбор* создает при первом выполнении тела цикла сразу два экземпляра переменной x_1 , одно со значением 0, другое со значением 1. Для каждого экземпляра (варианта) создается процесс – копия следующего (условного) оператора. Эти копии можно записать так:

if $\Phi(0, x_2, \dots, x_n)$ **then** успех **else** неудача,
if $\Phi(1, x_2, \dots, x_n)$ **then** успех **else** неудача.

При втором выполнении тела цикла создаются 2 экземпляра переменной x_2 , со значениями 1 и 0. Для каждого экземпляра создается процесс – копия существующих процессов. В результате будет создано уже 4 процесса

if $\Phi(0, 0, \dots, x_n)$ **then** успех **else** неудача,
if $\Phi(1, 0, \dots, x_n)$ **then** успех **else** неудача,
if $\Phi(0, 1, \dots, x_n)$ **then** успех **else** неудача,
if $\Phi(1, 1, \dots, x_n)$ **then** успех **else** неудача.

По окончании оператора цикла возникнет 2^n процессов, каждый из которых будет назначен на свой процессор недетерминированной машины. Вычисления значений $\Phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ будут происходить параллельно и закончатся за ограниченное относительно длины формулы время. Процессы, которые вырабатывают значение **неудача**, сразу завершаются без последствий для других процессов. Процесс, который первым выработал значение **удача**, выдает его в качестве общего результата работы всей недетерминированной программы, затем завершается сам и завершает остальные процессы. Таким образом, выполнима или не выполнима формула (выражение) мы узнаем за полиномиальное время работы недетерминированной машины. Доказательство завершено.

Ясно, что любая задача, решаемая за полиномиальное время на обычной последовательной машине, может быть решена за полиномиальное время и на недетерминированной машине, из чего следует:

$$P \subseteq NP.$$

Верно ли обратное (и тогда $P = NP$) или существуют задачи, решаемые за полиномиальное время на недетерминированной машине, но требующие экспоненциального времени на однопроцессорной машине (тогда $P \subset NP, NP \setminus P \neq \emptyset$)?

В первом случае ничего нового в классификации задач от введения класса NP мы не получаем. Но появляется много надежд. Дело в том, что для некоторых задач из класса NP известны только экспоненциальные алгоритмы для их решения на однопроцессорной машине. Если $P = NP$, то есть надежда разработать и полиномиальные алгоритмы. Вопрос этот не праздный. На карту поставлено не только машинное время. В криптографии (науке о шифровании информации) существует целое направление – шифрование с открытым ключом. В его основе использова-

ние математических задач, решаемых алгоритмами экспоненциальной сложности. Если вдруг для них найдутся эффективные полиномиальные алгоритмы, то многие секреты перестанут быть секретами.

Но и во втором случае появляется много вопросов. Что собой представляет класс $NP \setminus P$? Чем задачи из этого класса отличаются от других задач из EXP ? В настоящее время развита довольно большая теория задач класса NP в предположении, что $NP \neq P$. Поэтому дальше мы будем обсуждать строение класса NP .

Введем еще один класс задач — NP -полные (NPC) задачи (проблемы). Проблема T называется NP -полной, если она удовлетворяет двум условиям:

1. $T \in NP$.
2. Если $R \in NP$, то R сводится к T за полиномиальное время.

NP -полные проблемы являются своего рода универсальными. Нет необходимости придумывать алгоритмы для других задач, достаточно свести эти задачи к какой-либо NP -полной проблеме и воспользоваться алгоритмом ее решения.

В частности, С. Кук доказал в 1971 г., что проблема выполнимости для пропозиционального исчисления является NP -полной.

NP -полные проблемы, как следует из определения, являются наиболее трудными в классе NP .

Значение класса NP -полных проблем состоит еще и в том, что по иронии судьбы, ему принадлежат очень многие известные и важные в прикладном отношении задачи. Перечислим некоторые из них.

1. Задача изоморфизма подграфу.

Даны два графа, G_1 и G_2 . Множество вершин первого графа обозначим V_1 , а второго — V_2 . Пусть $|V_1| > |V_2| = n$. Требуется ответить на вопрос: найдется ли в графе G_1 подграф H , изоморфный графу G_2 .

2. Задача о клике.

Дан граф G с m вершинами и целое положительное число n . Граф называется *кликой*, если каждая вершина в нем связана реб-

ром с каждой. Количество вершин в клике назовем ее мощностью. Верно ли, что в данном графе G имеется клика мощности не менее чем n ?

3. Гамильтонов цикл.

Дан обыкновенный граф G с n вершинами. Некоторые ребра этого графа образуют простые циклы. Простым называется цикл, в котором вершины не повторяются. Гамильтонов цикл – это последовательность вершин и ребер графа, содержащая все вершины графа G по одному разу, но, может быть, содержащая не все ребра. Существует ли в графе простой цикл, проходящий через все вершины графа?

4. Задача коммивояжера.

Дан граф G с n вершинами. Каждому ребру графа приписано положительное целое число d_i , задающее «длину» ребра. Кроме этого, задано некоторое положительное целое число L . Требуется ответить на вопрос: найдется ли в графе G маршрут, проходящий через **все** вершины графа G , такой, что его длина не превышает L .

Другие модификации задачи о коммивояжере требуют отыскания маршрута минимальной длины, гамильтонова цикла и т.п. Неизменным остается требование однократного прохождения через все вершины.

5. Вершинное покрытие.

Дан обыкновенный граф G с m вершинами и целое положительное число n . *Вершинным покрытием* называется подмножество $U \subseteq V$ множества V вершин графа такое, что любое ребро графа G инцидентно хотя бы одной из вершин множества U (вторая вершина ребра может либо принадлежать, либо не принадлежать множеству U). Существует ли вершинное покрытие не более чем из n вершин?

6. Разбиение.

Дано множество положительных целых чисел x_1, x_2, \dots, x_n . Можно ли разбить его на два подмножества так, чтобы суммы чисел в обоих подмножествах были равны?

7. Трехмерное сочетание.

Даны три множества: A, B, C . Мощности их равны, $|A| = |B| = |C| = m$, но множества не пересекаются. Зафиксировано некоторое множество троек $N \subseteq A \times B \times C$, $|N| = n$. Трехмерным сочета-

нием называется подмножество $M \subseteq N$, $|M| = m$, в котором никакие две разные тройки не имеют ни одной равной координаты. Содержит ли множество N трехмерное сочетание?

Кроме класса NP -полных проблем рассматривают еще и класс NP -трудных проблем.

Проблема T называется NP -трудной, если она удовлетворяет условию: «Если $R \in NP$, то R сводится к T за полиномиальное время».

Заметим, что это условие совпадает со вторым условием для NP -полных проблем. Следовательно, NP -полнная проблема – это NP -трудная задача, принадлежащая классу NP .

3.3. ПАРАМЕТРИЗОВАННАЯ СЛОЖНОСТЬ

Ранее мы рассматривали сложность алгоритма (или задачи) как функцию $\tau = f(V)$ одной переменной. В качестве этой переменной рассматривался объем (сложность) исходных данных. Во многих случаях при таком подходе может быть получен точный результат. Например, сложность вычисления скалярного произведения векторов длины V , сложность перемножения квадратных матриц размера V .

Но также мы знаем, что для ряда задач сложность данных (по отношению к алгоритму) невозможно охарактеризовать одним параметром так, чтобы получалась функциональная зависимость $\tau = f(V)$. Например, алгоритм упорядочения массива чисел из V элементов будет выполнять различное число шагов, зависящее не только от значения V , но и от того, как числа расположены перед началом работы алгоритма. Как правило, для многих реальных алгоритмов упорядочение чисел, уже расположенных в нужном порядке, требует гораздо меньше времени, чем упорядочение чисел, расположенных первоначально в обратном порядке. Сложность решения задачи изоморфизма графов зависит не только от количества V вершин в графе, но и от количества ребер W , и от ряда других характеристик.

В таких случаях обычно вводятся две функции сложности – нижняя граница и верхняя граница. Чаще всего они имеют раз-

ный порядок роста, поэтому неясно, на что ориентироваться при сравнении нескольких алгоритмов, решающих одну и ту же задачу. Тогда вводится средняя оценка сложности на основе вероятностного подхода. Не преуменьшая ее теоретического значения, все-таки мы вправе задаться вопросом: а какую пользу она принесет в определении количества шагов алгоритма для вполне конкретных (не случайных) исходных данных?

Точное решение проблемы оценки сложности можно искать на пути характеризации исходных данных набором параметров, $\tau = f(p_1, p_2, \dots, p_n)$. Конечно, и здесь много подводных камней. Фактически, нам требуется любые данные (и не числовые тоже) характеризовать конечным набором чисел. Даже из общетеоретических соображений ясно, что в общем случае это невозможно. В качестве примера можно снова привести задачу определения изоморфизма графов. Однозначное кодирование графа конечным набором чисел означало бы, что мы решили задачу отыскания полного набора инвариантов графа, не зависящего от его размера. В теории графов эта задача не решена.

Правда, однозначная идентификация не требуется – достаточно, чтобы набор параметров разделял графы на классы с одинаковой сложностью по отношению к исследуемому алгоритму. Но здесь возникает новая проблема – для разных алгоритмов, решающих одну и ту же задачу, мы будем получать функции сложности, зависящие, может быть, от разных наборов параметров. Следовательно, мы не сможем сравнивать между собой и эти функции, и эти алгоритмы.

Тем не менее, безуспешные попытки решить проблему $P = ?$ NP вынуждают исследователей рассматривать функции сложности двух переменных $\tau = f(V, k)$. Такую оценку называют *параметризованной сложностью*. Задача состоит в том, чтобы выяснить, где скрываются истоки экспоненциальной сложности некоторых задач и, возможно, показать, что по важному параметру V сложность полиномиальна, а экспоненциальная зависимость проявляется по отношению к «второстепенному» параметру k .

Контрольные вопросы и задания

1. В чем состоит отличие общей задачи (массовой) от частной (индивидуальной) задачи?
2. В чем отличие класса NP-полных задач от класса NP-трудных задач? Может ли какая-либо задача принадлежать первому из этих классов и не принадлежать второму?
3. Является ли задача распознавания целого неотрицательного числа на простоту задачей полиномиальной сложности?
4. К какому классу сложности относится общая задача о рюкзаке? К какому классу относится задача о «супервозрастающем рюкзаке»?
5. Какова сложность задачи, если она может быть разделена на конечное число задач полиномиальной сложности, решаемых последовательно?
6. Какова сложность задачи, если она может быть разделена на последовательно решаемые задачи полиномиальной сложности, и количество этих задач выражается полиномом от сложности исходных данных?
7. Граф задан матрицей смежностей (квадратной матрицей, в которой каждой вершине графа соответствует строка и столбец; если две вершины смежны, то на пересечении соответствующей строки и столбца находится 1, в противном случае – 0. К какому классу относится сложность задачи распознавания того, является ли этот граф деревом?
8. Граф задан матрицей смежностей. К какому классу относится сложность задачи распознавания того, является ли этот граф двудольным?
9. Граф задан матрицей смежностей. К какому классу относится сложность задачи распознавания того, является ли этот граф самодополнительным?
10. К какому классу сложности относится задача распознавания принадлежности слов длины n автоматному языку с конечным алфавитом?
11. В чем состоит особенность параметрической сложности по отношению к традиционному определению сложности? Как

это влияет на отнесение задачи к тому или иному классу сложности?

12. Как можно применить подходы к параметризованной оценке сложности для алгоритмов на графах? Какие характеристики (инварианты) исходного графа можно выбрать в качестве параметров функции сложности?

Задачи для самостоятельного решения

1. Сформулируйте алгоритм полиномиального сведения задачи отыскания решения системы линейных уравнений к задаче вычисления определителя матрицы.

2. К какому классу относится задача линейного программирования (вспомните дисциплину «Методы оптимизации», постановку оптимизационной задачи и способы ее решения) – полиномиальной сложности или экспоненциальной сложности?

3. Данна строка длины n , составленная из символов конечного алфавита $A = \{a_1, a_2, \dots, a_k\}$. Задача состоит в том, чтобы определить, является ли эта последовательность циклической. Найдите верхнюю и нижнюю оценки сложности решения этой задачи. Как сложность зависит от количества k символов в алфавите?

4. Данна строка длины n , составленная из символов конечного алфавита, и m правил автоматной грамматики. Задача состоит в том, чтобы определить, относится ли эта строка к языку, порождаемому данной грамматикой. Найдите верхнюю и нижнюю оценки сложности этой задачи.

5. Оцените сложность задачи построения дополнения обычновенного неориентированного графа G , содержащего p вершин, если исходный граф задан матрицей смежностей. Дополнением графа называется неориентированный граф, содержащий те же вершины, что и граф G , но содержащий только те ребра полного графа, которые отсутствуют в графе G .

6. Компонентой графа называется максимальный связный подграф. Если граф связан, то он является единственной компонентой. Несвязный граф состоит из нескольких компонент. Оцените сложность задачи вычисления количества компонент в произвольном графе с p вершинами.

7. Проведите анализ сложности задачи определения того, является ли граф с p вершинами планарным. Используйте теорему Понtryгина – Куравского и те факты, что полный граф с пятью вершинами не планарен, и полный двудольный граф с тремя вершинами в каждой доле также не планарен.

4. СУЩЕСТВОВАНИЕ АЛГОРИТМОВ

4.1. ПРОБЛЕМА СУЩЕСТВОВАНИЯ АЛГОРИТМА

В предыдущей главе обсуждалась сложность решения задач. Решением частной задачи является найденный объект: число, вектор, матрица, граф, маршрут в конкретной сети. Сложность решения такой задачи – это некоторое натуральное число – количество элементарных шагов.

Решением общей задачи является алгоритм, с помощью которого может быть решена любая частная задача, получаемая из данной общей задачи. Сложность общей задачи – это функция, аргументом которой является сложность исходных данных. Это сложность наилучшего алгоритма (из многих существующих и потенциально возможных), решающего задачу. В тени оставался вопрос о том, существует ли хотя бы один алгоритм, решающий эту общую задачу.

Рассмотрим в качестве примера задачу о диофантовых уравнениях.

Диофантовы уравнения (названы в память греческого математика Диофанта, жившего в III в. н. э.) имеют вид

$$P(x_1, x_2, \dots, x_n) = Q(x_1, x_2, \dots, x_n),$$

где P и Q – полиномы, например:

$$ax^2 + bx + c = y^2; \quad ax^n + bx^n = cz^n; \quad ax^3 + bx^2y + cxy^2 + dy^3 = 1.$$

Коэффициенты a, b, c и степень n – целые числа; решения – значения x, y, z – также должны быть целыми числами. Конечно, решения не всегда существуют: вспомним хотя бы известную теорему Ферма о том, что уравнение $x^n + y^n = z^n$, $n > 2$, не имеет целочисленных решений. Теорема была сформулирована Пьером де Ферма в середине XVII в. без доказательства и доказана Эндрю Уайлсом в 1995 г.

Математик Давид Гильберт сформулировал задачу (известную как 10-я проблема Гильберта): указать способ, при помощи которого возможно после конечного числа операций установить,

разрешимо ли это уравнение в целых числах. Иначе говоря, разработать алгоритм, на вход которого подается запись конкретного уравнения (текст), а на выходе появляется по окончании работы одно из двух текстовых значений: «решение существует», «решение не существует».

Эта задача распознавания проще, чем задача нахождения решения, т.е. определения числовых значений x, y, z, \dots , удовлетворяющих диофантову уравнению. Но все равно она достаточно амбициозна, если учесть тот срок, который потребовался математикам для ответа на такой же вопрос, сформулированный в теореме Ферма.

Неудивительно, что на решение 10-й проблемы Гильберта ушло 70 лет, и ответ был неожиданным: Ю.Матиясевич доказал, что такого алгоритма не существует!

Общая задача определяется:

- 1) списком параметров lp – свободных переменных, конкретные значения которых не определены;
- 2) формулой условий – свойств, которыми должен обладать ответ (решение задачи).

Решением такой задачи $z(lp)$ со списком параметров lp является *алгоритм* $\alpha(lp)$ – тоже кодируемый строкой символов, интерпретируемой (выполняемой) некоторым вычислителем, причем важно то, что некоторые участки строки интерпретируются многократно (циклически) потенциально неограниченное число раз. Более точно можно сказать, что решением массовой задачи являются алгоритм + вычислитель.

Частная задача получается из общей задачи, если всем параметрам общей задачи придать конкретные значения (задать исходные данные).

Некоторые массовые задачи решения не имеют. Это значит, что не существует алгоритма как решения массовой задачи $z(lp)$. Если же для каждого параметра задачи из списка lp задать конкретное значение из предметной области, т.е. превратить массовую задачу в индивидуальную, то решение вполне может найтись. Различие состоит в том, что для разных частных задач z_1, z_2, z_3, \dots , полученных из одной общей задачи z , могут потребоваться разные алгоритмы $\alpha_1, \alpha_2, \alpha_3, \dots$. Поэтому более точно: не сущ-

ствует общего алгоритма, решающего массовую задачу. Возможно, что только для некоторых параметров задаются конкретные значения, т.е. список lp уменьшается до более короткого списка lp' , но при этом «упрощенная» общая задача получает решение.

В связи с этой проблемой возникли задачи о задачах $Z(z(lp))$. Задача Z о задаче $z(lp)$ может быть сформулирована так: существует ли алгоритм $\alpha(lp)$, решающий задачу $z(lp)$?

Для того чтобы такие задачи можно было решать на строго научной основе, математическими методами, должно быть введено строгое понятие алгоритма. Формулировок таких понятий в теории алгоритмов было введено несколько. Наиболее известными являются машина Тьюринга, машина Поста, алгорифмы Маркова. Было доказано, что они эквивалентны в том смысле, что алгоритм, представленный в одной форме, может быть переведен в другую.

4.2. Машины Тьюринга и алгорифмы Маркова

В 1937 г. американский ученый Алан Тьюринг (A. Turing) предложил следующий способ описания алгоритма.

Данные, с которыми работает алгоритм, записываются на ленте. Эта лента разделена на ячейки. В каждой ячейке может быть записан только один символ, либо ячейка может быть пустой. Лента машины Тьюринга неограничена влево и вправо, но в любой момент времени на ней записано только конечное число символов.

У машины имеется головка чтения/записи M . Машина может читать содержимое только той ячейки ленты, которая в данный момент находится напротив головки чтения/записи. Кроме этого машина имеет *внутреннюю память конечного объема k* . Память имеет k различных состояний. Соответственно будем считать, что машина Тьюринга в любой момент времени находится в одном из k состояний.

Каждая конкретная машина характеризуется набором *команд*, которые она может выполнять. Этот набор команд называется *программой* машины Тьюринга.

Какая команда программы будет выполняться в данный момент, определяется двумя параметрами: читаемым головкой символом и состоянием машины.

Результаты выполнения команд:

1. Новый символ, записанный на ленту в ту ячейку, напротив которой находится в данный момент головка.

2. Перемещение головки на одну ячейку вправо или влево вдоль ленты.

3. Переход машины в новое состояние.

В частном случае новый символ может быть равен старому, перемещение может отсутствовать, состояние может остаться прежним.

Формат команды имеет следующий вид:

где a – читаемый в ячейке ленты символ, b – символ, записываемый в эту ячейку вместо символа a , q – текущее состояние машины Тьюринга, r – новое состояние машины, D – направление движения головки (относительно ленты).

Символы выбираются из конечного алфавита $A = \{a_1, a_2, \dots, a_l\}$. Обычно используют трехсимвольный алфавит $A_2 = \{e, 0, 1\}$, где e означает пустой символ, отсутствие информации в ячейке. С помощью 0 и 1 будут кодироваться все данные.

Иногда используют двухсимвольный алфавит $A_1 = \{e, 1\}$. В этом случае числа кодируются только единицами: нуль – 1 единица, один – 2 единицы, натуральное число x кодируется $x+1$ единицами. Это единичная система счисления.

Множество состояний обозначим $Q = \{q_1, q_2, \dots, q_k\}$. Обычно состояние q_1 считается начальным состоянием.

Направление движения D выбирается из множества $D = \{L, R, S\}$, где L – движение головки влево, R – движение головки вправо, S – отсутствие движения.

Команда $1q_30q_6L$ означает: машина находится в состоянии q_3 , в ячейку с символом 1 записывается символ 0, производят сдвиг головки влево и переходит в состояние q_6 . То есть команда может рассматриваться как отображение пар (a, q) в тройку (b, r, D) .

Это отображение является однозначным: для произвольной пары существует не более одной тройки. Но не для всех пар существуют тройки (отображение является частичным).

Работа машины Тьюринга. В начальный момент времени на ленте находится некоторая исходная строка символов. Состояние соответствует некоторому начальному состоянию.

Момент старта рассматривается как нулевой момент времени. В начальном положении головка находится напротив самого левого непустого символа. В момент старта выполняется первая команда. В результате выполнения команды машина перейдет в новое состояние и к новому символу, шаг за шагом, постепенно изменяя содержимое ленты, пока для некоторой пары (f, q) не окажется команды в программе. Такая ситуация считается завершающей. Оставшаяся запись на ленте считается записью результата.

Таким образом, машина Тьюринга реализует вычисление некоторой функции: отображает исходную строку символов в результат.

Существует несколько способов представления программы машины Тьюринга (множества команд). Наиболее употребительные:

1. Двумерная таблица (рис. 1).
2. Диаграмма (нагруженный псевдограф).

Табличное представление. Строки таблицы помечаются символами алфавита a , а столбцы – различными состояниями машины q . Каждой команде программы соответствует одна клетка в таблице. Например, для команды $a q b r D$ в ячейку таблицы, которая находится на пересечении строки a и столбца q , вписывается тройка $b r D$. Для некоторых пар (a, q) в программе нет команд, следовательно, клетки остаются пустыми. При достижении в процессе работы пустой клетки машина Тьюринга останавливается.

Состояние Символ	q_1	q	q_k
a_1					
.....					
a			$b \ r \ D$		
.....					
a_l					

Рис. 1. Табличная форма программы машины Тьюринга

Пример. Рассмотрим программу вычисления функции $S(x) = x + 1$, т. е. увеличение аргумента на 1.

Используя алфавит $A_2 = \{e, 0, 1\}$, x будем кодировать последовательностью нулей и единиц, как при двоичном кодировании целых неотрицательных чисел.

Предположим, что в момент старта головка машины Тьюринга находилась напротив крайней левой ячейки с символом 1.

Первая выполняемая команда $1q_11q_1R$ – не меняет символ в ячейке и состояние и производит сдвиг головки вправо по ленте. Так продвигаемся вправо, пока не встретится символ e . Это означает, что код числа x закончился. После этого начинается процесс прибавления 1.

	q_1	q_2	q_3	q_4
0	$0q_1R$	$1q_3L$	$0q_3L$	
1	$1q_1R$	$0q_2L$	$1q_3L$	
e	eq_2L	$1q_4S$	eq_4R	

Рис. 2. Программа машины Тьюринга для вычисления функции $S(x) = x + 1$

Если младшей цифрой является 0, то достаточно заменить ее на 1 командой $0q_21q_3L$ и начать обратное движение к исходной позиции.

Если младшая цифра 1, то результатом в данной ячейке будет 0 и единица перейдет в старший разряд. Процесс переноса может закончиться где-то внутри числа и тогда нужно будет осуществить «прокрутку» ленты к исходной позиции.

Алан Тьюринг сформулировал тезис, который связывает понятие алгоритма и машины: «*Для всякого неформального алгоритма может быть построен Тьюрингов алгоритм (программа машины Тьюринга), дающий при одинаковых исходных данных тот же результат*».

Нормальные алгорифмы. Российский математик Андрей Марков (1903–1979) предложил свою формализацию понятия алгоритма, которую он назвал нормальным алгорифмом. Он рассматривал алгорифмы, работающие со словами, т.е. цепочками букв в некотором конечном алфавите A . Такие алгоритмы он называл *вербальными алгорифмами*. Процесс работы такого алгорифма в алфавите A состоит в последовательном порождении слов в алфавите A согласно предписанию. Работа начинается с некоторого исходного слова (входные данные) и заканчивается порождением слова-результата. Хотя для некоторых входных слов процесс может и не заканчиваться. Если же для слова W процесс заканчивается некоторым результатом Q , то говорят, что алгоритм α *применим* к слову W и обозначают этот факт так:

$$! \alpha (W).$$

Запись $\alpha: W \Rightarrow Q$ означает то, что алгорифм α перерабатывает слово W в слово Q . Делает это он за несколько шагов и мы собираемся формализовать этот процесс и дать математически точное понятие нормального алгорифма.

Для того чтобы описать алгорифм, нужен язык, который, в свою очередь, основывается на некотором алфавите. У нас уже есть алфавит A для записи слов. Но его недостаточно. Дополним алфавит A тремя метасимволами \rightarrow , $\rightarrow.$ и $|$ (эти символы не должны первоначально входить в A). В расширенном алфавите можно записывать разные слова, но мы особенное внимание уде-

лим словам вида $L \zeta R$, где ζ – один из двух символов: стрелка или стрелка с точкой, а L и R – слова в алфавите A . Такие слова называются *формулами подстановок*: $L \rightarrow R$ – простые формулы подстановок, $L \rightarrow. R$ – заключительные формулы подстановок. L – левая часть формулы подстановки, R – правая часть. Будем говорить, что формула подстановки S *действует на слово W*, если левая часть S входит в W , т.е. слово W можно представить в виде $W = ULV$, где W и V – слова в алфавите A , могут быть пустыми. *Результатом действия S на слово W* будет слово URV , т.е. результат подстановки правой части формулы вместо левой части. Если L входит в W несколько раз, то подстановка R будет осуществляться вместо первого вхождения.

Схемой называется слово в расширенном алфавите, имеющее вид

$$|S_1|S_2|S_3|\dots|S_i|\dots|S_n|,$$

где S_i – формулы подстановок. *Схема Sch действует на слово W*, если в схеме найдется формула подстановки, действующая на это слово. Если в схеме несколько таких формул, то берется первая (с наименьшим номером) и результат ее действия (слово W') считается *результатом действия схемы Sch*. Говорят, что схема Sch *переводит слово W в слово W'* (просто переводит или заключительно переводит в зависимости от типа примененной подстановки).

Нормальный алгорифм задается указанием трех объектов: алфавита A , алфавита $\{\rightarrow, \rightarrow., |\}$ и некоторой схемы Sch . Алфавит A называют алфавитом алгорифма α , а схему Sch – схемой алгорифма. Всякий нормальный алгорифм однозначно определяет пошаговый процесс преобразования входного слова W . На первом шаге берется само слово W , и если схема Sch действует на это слово, то оно переводится в результате одной подстановки в слово W' . На следующем шаге берется слово W' и процесс продолжается. Если же на некотором шаге была применена заключительная формула подстановки, результатом действия которой является слово Z , то процесс обрывается; если схема Sch не дей-

вует на очередное слово T , то процесс также обрывается. При этом результатом работы нормального алгорифма α будет слово Z или T и этот результат обозначается $\alpha(W)$. Если процесс преобразования слова W никогда не обрывается, то результат преобразования W алгорифмом α не определен.

В формулах подстановок слова L или R могут быть пустыми. Если слово R пустое, то результатом действия такой подстановки на слово $W = U L V$ будет слово $U V$. Если пустым является слово L , то считается, что результатом подстановки является слово RW , т.е. правая часть правила приписывается слева к слову W . Конечно, это довольно специальный случай. Например, алгорифм со схемой $Sch = |\Lambda \rightarrow. \Lambda|$, где Λ обозначает пустое слово (пустое слово не содержит ни одной буквы) задает тождественную функцию $f(R) = R$, а алгорифм со схемой $Sch = |\Lambda \rightarrow \Lambda|$ вычисляет нигде не определенную функцию.

Рассмотрим несколько простых примеров нормальных алгорифмов.

1. Имеем произвольный алфавит A (конечно, не включающий метасимволов \rightarrow , $\rightarrow.$ и $|$). Схема $Sch = |\Lambda \rightarrow. Q|$. Для всякого слова W в алфавите A $\alpha: W \Rightarrow QW$, т.е. все, что алгорифм делает, – это приписывает к слову W слева всегда одно и то же слово Q , после чего останавливается.

2. Задан алфавит $A = \{0, 1, 2, 3\}$. Схема $Sch = |0 \rightarrow \Lambda| |1 \rightarrow \Lambda| |2 \rightarrow \Lambda| |3 \rightarrow \Lambda|$. Алгорифм с такой схемой будет *аннулирующим* алгорифмом, так как он переводит любое слово в пустое, $\alpha: W \Rightarrow \Lambda$.

Однотипные правила в схеме удобно записывать в сокращенном виде. Обозначим буквой ξ произвольную букву алфавита A . Тогда все четыре правила в примере 2 могут быть записаны как одно правило, и вся схема будет записана сокращенно как

$$Sch = |\xi \rightarrow \Lambda|, \text{ где } \xi \text{ пробегает алфавит } A.$$

3. Зададим некоторое слово C в алфавите A и сокращенную схему

$Sch = |\xi \rightarrow \Lambda| |\Lambda \rightarrow. C|$, где ξ пробегает алфавит. Алгорифм с такой схемой переводит любое слово в слово C , $\alpha: W \Rightarrow C$.

4.3. ДОКАЗАТЕЛЬСТВА НЕСУЩЕСТВОВАНИЯ АЛГОРИТМОВ

Обсудим теперь проблему разрешимости и неразрешимости с точки зрения машин Тьюринга, т.е. точного понятия алгоритма. Машина Тьюринга вычисляет значение некоторой функции натуральных аргументов, т.е., каждая машина представляет некоторую функцию. Верно ли обратное: каждая функция может быть представлена некоторой машиной? Ответить на этот вопрос очень легко. Поскольку каждая машина однозначно задается своей программой, а программу можно записать как последовательность команд – строку конечной длины в конечном алфавите, то мы имеем счетное количество строк, т.е. множество всех машин Тьюринга счетно. В то же время множество всех функций натуральных аргументов, дающих в качестве результата натуральные числа, – несчетно!

Теорема. Множество всех функций, определенных всюду или частично на множестве натуральных чисел и дающих в качестве результата натуральные числа, несчетно.

Доказательство. Предположим обратное: упомянутое множество функций счетно, т.е. имеется некоторое перечисление функций: f_1, f_2, f_3, \dots , ставящее в соответствие каждой функции индекс – натуральное число. Важно заметить, что *все* возможные функции попали в этот ряд. Зададим функцию $u(n)$ следующими соотношениями:

if $f_n(n)$ не определено **then** $u(n) := 1$ **else** $u(n) := f_n(n) + 1$.

Таким образом, u – всюду определенная на множестве натуральных чисел функция, принимающая натуральные значения. Но она никак *не может содержаться* в нашем списке f_1, f_2, f_3, \dots !

Действительно, если бы для какого-либо номера m выполнялось $u = f_m$, то из определения функции u следовало бы, что

$f_m(m) = u(m) = 1$, если $f_m(m)$ не определено [противоречие!]
или

$f_m(m) = f_m(m) + 1$ [противоречие!].

Следовательно, существуют функции, не вычисляемые никакой машиной Тьюринга. Такие функции называются невычислимими. Если теперь сказать, что вычисление функции это решение задачи, а программа машины Тьюринга – алгоритм, то мы делаем вывод о существовании *алгоритмически неразрешимых задач*.

В теории алгоритмов известно много таких задач. Перечислим наиболее важные из них.

1. *Проблема остановки*. При обсуждении машин Тьюринга мы говорили, что на некоторых исходных данных машина может не останавливаться, т.е. не давать результата. Любая машина Тьюринга может быть представлена некоторым кодом (номером), отличающимся от всех других. Например, каждое состояние мы можем закодировать числом, символы движения закодировать различными числами. Тогда каждая команда представляет собой строку чисел, которую можно интерпретировать как одно большое число, а последовательность всех команд – как еще большее число N . Эта или какая-либо другая процедура, устанавливает однозначное соответствие между множеством натуральных чисел и множеством алгоритмов. Функция $\phi : \text{Natur} \rightarrow \text{Algorithms}$, называется *нумерацией* алгоритмов, а ее аргумент N – номером алгоритма при нумерации ϕ . Функция ϕ по номеру N восстанавливает описание алгоритма, $\phi(N) = \alpha$. Обратная функция по описанию алгоритма определяет его номер. Введение нумераций позволяет работать с алгоритмами как с натуральными числами, что особенно удобно при исследовании алгоритмов над алгоритмами: алгоритм, закодированный числом, может рассматриваться как входные данные другого алгоритма. *Проблема остановки* состоит в построении машины Тьюринга M , которая получая на входе код (номер) N произвольной машины T и входные данные этой машины X , определяет, остановится ли машина T на данных X . Иначе говоря,

$$M(N, X) = 1, \text{ если } T \text{ останавливается на } X,$$

$$M(N, X) = 0, \text{ если } T \text{ не останавливается на } X.$$

Доказано, что машину M построить нельзя, т.е. *проблема остановки алгоритмически неразрешима*.

2. Проблема эквивалентности. Для вычисления одной и той же функции можно построить две различные машины Тьюринга, отличающиеся набором команд, а значит, и последовательностью действий. Тем не менее, для любых исходных данных обе машины будут вычислять одинаковые результаты. Такие машины естественно считать эквивалентными. *Проблема эквивалентности* состоит в построении машины Тьюринга, получающей на входе описания (коды) двух машин T_1 и T_2 и определяющей, эквивалентны ли T_1 и T_2 . С практической точки зрения можно поставить аналогичный вопрос: можно ли написать программу, которая по текстам двух программ на Паскале определяет, закончатся ли они одинаковыми результатами.

Ответ отрицательный – *проблема эквивалентности алгоритмически неразрешима*. Это не означает, что для двух конкретных программ нельзя выяснить, эквивалентны ли они. Просто не существует *общего* метода. Для каждой пары программ придется применять собственные методы, зависящие от существа этих программ.

Понятия вычислимой функции (т.е. той, для которой существует вычисляющая ее машина Тьюринга – алгоритм) и собственно алгоритма не следует смешивать. Различие между вычислимой функцией и алгоритмом – это различие между функцией и способом ее задания. Для одной и той же функции может существовать бесконечное число способов задания – текстов алгоритмов. Тривиальный пример: к тексту алгоритма, вычисляющего функцию, припишем текст тождественного алгоритма (результат которого всегда равен его входным данным); новый алгоритм представляет ту же самую функцию; теперь припишем текст тождественного алгоритма еще раз, затем еще раз; мы получим бесконечную последовательность различающихся алгоритмов, представляющих одну функцию. Подобные факты и порождают множество проблем в теории алгоритмов, связанных с разрешимостью. Существует даже теорема (теорема Райса), утверждающая, что «*Никакое нетривиальное свойство вычислимых функций не является алгоритмически разрешимым*».

Тривиальное свойство означает принадлежность функции либо множеству всех функций, либо пустому множеству. Нетри-

виальное свойство – «функция f принадлежит классу C », где C – такой непустой класс, что существуют функции, не принадлежащие ему. Нумерация всех алгоритмов является одновременно и нумерацией всех вычислимых функций: функции можно присвоить номер одного из вычисляющих ее алгоритмов. Теперь теорему Райса можно сформулировать иначе: *Не существует алгоритма, который по номеру функции f определял бы, принадлежит эта функция заданному классу C или нет.*

Из изложенного видно, что проблема алгоритмической неразрешимости достаточно серьезна. На практике это означает, что если ставится проблема построения алгоритма, имеющего общий (универсальный) характер, например, автоматического синтеза программ по описанию *произвольной* задачи, то скорее всего такая проблема не может быть решена. Но если проблему ограничить, не ориентируясь на синтез программ для произвольных задач, а очеркнуть более узкий круг задач и задать простые правила описания таких задач, то, вероятно, проблема автоматического синтеза может быть решена. Более того, известны примеры успешных реализаций систем автоматического синтеза программ.

Каждый программист, разрабатывая новый алгоритм, должен решать свою локальную проблему остановки – всегда ли его алгоритм будет завершаться. Без доказательства этого факта заказчик не примет работу. Алгоритмическая неразрешимость общей проблемы остановки означает, что программисту для *каждого конкретного* алгоритма придется придумывать *собственный метод* доказательства завершаемости.

Общая проблема эквивалентности на практике касается двух проблем:

1) действительно ли алгоритм выполняет ту работу, для которой он создавался (проблема правильности алгоритма, соответствия алгоритма функции);

2) сохранит ли правильность преобразованная (например, с целью оптимизации или переноса на другую платформу или в иную операционную обстановку) программа, будет ли выполнять те же функции.

Обе эти проблемы для конкретных программ и алгоритмов решить, как правило, можно.

Различные сложности в постановках задач, в теории множеств заставили ученых развивать *основания математики* как самостоятельную область исследования.

Основания математики посвящены изучению формальных методов постановки задач и проведения доказательств (решения задач).

Отдельные содержательные математические теории становятся объектами исследования, о свойствах теорий доказываются *метатеоремы*. Утверждения же (теоремы) частной математической теории рассматриваются как элементарные объекты. Можно провести (очень приблизительную!) аналогию с языком программирования: в языке есть константы-значения и есть переменные. Описывая на языке действия с переменными, мы тем самым указываем машине, что нужно и возможно делать с текущими значениями этих переменных. Также в метатеоремах мы оперируем с переменными-теоремами и указываем, какого рода теоремы могут быть доказаны в частной математической теории, а какие – не могут.

Цель, поставленная в основаниях математики, впечатляет. Поставим в соответствие содержательной теории S (например, арифметике) формальную теорию T (формальную арифметику), включающую правила вывода. Тогда из аксиом, механически (алгоритмически) пользуясь правилами вывода, можно получать новые формулы теории T и соответствующие им теоремы содержательной теории S . Таким образом, вычислительная машина может заменить математика.

Однако довольно быстро выяснилось, что не всякое истинное утверждение содержательной теории может быть доказано (выведено) в соответствующей формальной теории. Иначе говоря, формальная теория может быть не полна. Более того, Гедель (в 1931 г. – австрийский логик, математик и философ математики) доказал, что *любая формальная теория, содержащая формальную арифметику, неполна*.

Далеко не все задачи решаются алгоритмически или, как принято говорить в математике, конструктивно. Многие решения,

особенно в Анализе, формулируются в виде теорем существования, утверждающих, что объект имеется, описывающих его свойства, но не указывающих, как его найти (построить). Некоторые математики, начиная с Брауэра, стали отвергать *актуальную бесконечность* (бесконечные множества, данные нам сразу, в один момент) и заменять ее на *потенциальную осуществимость* (возможность построения бесконечного множества поэтапно, элемент за элементом). Это привело к появлению новых направлений в математической логике, в частности, к *интуиционизму*. Считается, что логический закон исключенного третьего ($P(x) \vee \neg P(x) = \text{true}$) в применении к высказываниям $P(x)$ о бесконечных множествах x не может использоваться. В результате родилась конструктивная математика, очень не похожая на классическую.

По духу конструктивная математика ближе к программированию, хотя результаты классической математики успешно применяются при разработке прикладного программного обеспечения.

Контрольные вопросы и задания

1. Из каких элементов состоит команда машины Тьюринга? Что собой представляет программа машины Тьюринга?
2. Где и в какой форме записываются исходные данные для работы машины Тьюринга?
3. Каковы условия остановки – прекращения работы машины Тьюринга? Где и в какой форме записан результат работы машины Тьюринга?
4. Может ли машина Тьюринга никогда не останавливаться? Как интерпретировать бесконечную работу машины Тьюринга?
5. В какой форме представлены исходные данные и результаты работы нормального алгорифма Маркова?
6. Может ли процесс подстановок при работе алгорифма Маркова никогда не завершаться?
7. Могут ли метасимволы встречаться в словах – исходных данных и результатах работы нормальных алгорифмов Маркова? Какова роль метасимволов в описании алгорифма?

Задачи для самостоятельного решения

1. Напишите программу машины Тьюринга для вычисления функции $f(x) = x + 2$ в алфавите $A = \{e, 0, 1\}$. Код числа x размещен на ленте перед началом вычислений. Незначащих нулей слева нет. Начальное положение головки машины – напротив первой слева значащей цифры 1. Конечное положение – стандартное, напротив первой слева цифры 1. Незначащих нулей результата по окончании программы не образуется.
2. Напишите программу машины Тьюринга для вычисления функции $f(x_1, x_2) = x_1 + x_2$ в алфавите $A = \{e, 1\}$. Коды чисел x_1 и x_2 размещены на ленте перед началом вычислений и разделены одним пустым символом e . Начальное положение головки машины – напротив первой слева цифры числа x_1 .
3. Напишите программу машины Тьюринга для вычисления функции $f(x) = 2 \times x$ в алфавите $A = \{e, 1\}$. Код числа x размещен на ленте перед началом вычислений. Конечное положение – стандартное, напротив первой слева цифры 1.
4. Выведите формулу для сложности программы машины Тьюринга, вычисляющей значение функции $f(x) = x + 1$ в алфавите $A = \{e, 0, 1\}$. За единицу времени выполнения берется время выполнения одной команды, включая изменение состояния, запись символа на ленту и сдвиг головки (если требуется).
5. Напишите семейство программ машины Тьюринга для вычисления функции $f(x) = k \times x$ в алфавите $A = \{e, 1\}$. Здесь k – некоторая константа, не размещаемая на ленте машины. Оцените сложность как функцию двух параметров k и x .
6. Напишите программу машины Тьюринга для вычисления функции $f(x) = 3 \times x$ в алфавите $A = \{e, 0, 1, 2\}$.
7. Разработайте нормальный алгорифм Маркова для вычисления функции $f(x) = x + 1$ в алфавите $A = \{1\}$. Исходное значение x задано словом W , представляющим собой код x в единичной системе счисления.
8. Разработайте нормальный алгорифм Маркова для вычисления функции $f(x_1, x_2) = x_1 + x_2$ в алфавите $A = \{1, +\}$. Исходное слово W представляет собой последовательность кодов x_1 и x_2 в единичной системе счисления, разделенных символом «+». Решение задачи 8 описано в следующем параграфе.

зультат работы должен представлять собой код суммы чисел x_1 и x_2 .

9. Разработайте нормальный алгорифм Маркова, инвертирующий слово W , т.е. записывающий символы, входящие в слово W , в обратном порядке. При необходимости введите дополнительные (вспомогательные) символы в алфавит A .

10. Оцените сложность нормальных алгорифмов Маркова из задач 8 и 9.

11. Разработайте программу машины Тьюринга для инвертирования слова, записанного на ленте. Проведите сравнение с алгорифмом Маркова.

ЗАКЛЮЧЕНИЕ

Вся история вычислительной техники – это история борьбы за производительность. В ней участвуют ученые многих специальностей. Физики стараются создавать быстродействующую элементную базу, математики создают новые методы решения задач. Ученые в области компьютерных наук разрабатывают алгоритмы решения задач и архитектуры вычислительных машин и процессов. Каждый из них стремится к лучшему результату, но в каждом классе решений есть свои ограничения, свои пределы. Здесь много противоречий. Например, для физиков одним из ограничений при передаче сигналов является скорость света. Увеличение частоты переключения элементов в вычислительной машине приводит к такому росту количества выделяемого тепла, которое способно расплавить процессор.

Вполне понятно стремление к универсальности: это экономит время и силы. И наряду с универсальными ЭВМ появляются универсальные языки программирования, ведутся разработки алгоритмов, которые должны решать широкий круг задач. Но универсальные языки (Алгол-68, PL/1, Ада) не приживаются, а если сформулировать математическую задачу слишком общо, то, как мы видели, не найдется алгоритма, ее решающего. Необходим разумный компромисс между общим и частным.

Для решения некоторых задач требуется так много элементарных шагов, что если они выполняются последовательно, результат не будет получен в приемлемые сроки. Казалось бы большое время работы компьютера (большая сложность алгоритмов) – отрицательный факт. Но оказалось, что такие алгоритмы очень нужны в криптографии, они даже составили целую область исследований и практического применения – асимметричную криптографию. Конечно, в других случаях прилагаются большие усилия для разработки быстрых алгоритмов, решающих важные вычислительные задачи – быстрая сортировка, быстрое перемножение матриц, быстрое преобразование Фурье и т.д.

В предлагаемом учебном пособии рассмотрены наиболее важные проблемы вычислимости и сложности алгоритмов. В

значительной мере известные классические теории результаты относятся к случаю одного исполнителя. Насколько может возрасти производительность, если использовать несколько параллельно работающих исполнителей, объединенных в многопроцессорную вычислительную машину или в распределенную компьютерную систему? Отдельные ответы известны и были приведены ранее, но теория сложности алгоритмов для распределенных вычислителей еще мало разработана и требует дальнейших исследований.

ГЛОССАРИЙ

Алгоритм (*algorithm*) – описание последовательности действий исполнителя по решению некоторой задачи. Различают интуитивное и формальное понятие алгоритма.

Алгоритмическая разрешимость (*algorithmic solvability*) – свойство задачи (проблемы) быть решаемой с помощью некоторого алгоритма.

Алгоритмически неразрешимая задача (*algorithmic insoluble problem*) – общая задача, в описание которой входят исходные данные (аргументы функции), для которой не существует универсального алгоритма решения; этот факт должен быть доказан математическими средствами; при этом могут существовать алгоритмы для решения частных случаев этой задачи, при некоторых исходных данных, но они различаются для различных частных случаев.

Архитектура вычислительной системы (*computer system architecture*) – представление (описание) вычислительной системы в виде блоков различного назначения и связей между этими блоками; связи показывают направления и способы передачи информации между блоками.

Верхняя оценка сложности алгоритма (*upper bound*) – числовая функция от сложности исходных данных, значения которой не могут быть меньше значения сложности алгоритма при таких же величинах сложности исходных данных; этот факт должен быть доказан математическими средствами.

Временная сложность алгоритма (*time complexity*) – функция, аргументом которой является параметр сложности исходных данных алгоритма, а результатом – количество элементарных шагов, которые алгоритм выполнит до остановки при этих исходных данных.

Выполнимость формулы (*Boolean satisfiability*) – свойство логической формулы иметь значение «истина» хотя бы на одном наборе значений переменных, входящих в эту формулу.

Вычислимая функция (*computable function*) – функция (обычно, целочисленная функция целочисленных аргументов), для которой существует алгоритм вычисления ее значения; существование алгоритма должно быть доказано математически, не обязательно конструктивно (т.е. сам алгоритм может быть не описан).

Вычислимость (*computability*) – свойство общей (зависящей от параметров – исходных данных) задачи, заключающееся в том, что существует алгоритм, решающий задачу. Алгоритм получает на входе значения параметров и через конечное число шагов выдает результат. Текст алгоритма не зависит от значений входных параметров.

Гиперкуб (*hyper-cube*) – класс связных регулярных графов, с количеством вершин, равным степени двойки; класс порождается из простейшего гиперкуба – двух смежных вершин с помощью операции произведения графов.

Диаметр графа (*graph diameter*) – длина наибольшего из кратчайших путей между парами вершин графа.

Дизъюнкт (*disjunction of literals*) – литерал или дизъюнкция нескольких литералов.

Емкостная сложность алгоритма (*space complexity*) – функция, определяющая зависимость объема памяти вычислителя, требуемого алгоритмом, от параметра сложности исходных данных алгоритма.

Задача о выполнимости (*satisfiability problem*) – для заданной формулы логики высказываний требуется определить, существует ли какой-либо набор значений переменных, при котором эта формула принимает значение «истина».

Задача распознавания (*recognition problem, decision problem*) – задача определения того, принадлежит ли конкретный объект заданному собственному подмножеству из предметной области.

Изоморфизм графов (*graph isomorphism*) – взаимнооднозначное отображение множеств вершин двух графов, сохраняющее смежность вершин.

Индивидуальная задача (*individual problem*) – то же, что «частная задача».

Итерационный процесс (*iteration process*) – многошаговый процесс вычисления объекта, удовлетворяющего определенному условию, начинающийся с объекта, возможно, не удовлетворяющего этому условию; в результате выполнения каждого шага получается новый объект, все более близкий к искомому; предполагается, что на множестве объектов существует метрика.

Класс сложности (*complexity class*) – класс алгоритмов (или задач) имеющих одинаковые (с точностью до постоянных множителей и малых слагаемых) функции сложности (временной или емкостной).

Кодирование данных (*coding*) – преобразование объектов предметной области в объекты, распознаваемые техническим устройством (компьютером, системой передачи данных); как правило, требуется, чтобы преобразование было взаимнооднозначным.

Команда (*instruction*) – элементарная инструкция для исполнителя, содержащая небольшое количество исходных данных, заканчивающаяся выработкой результата, и требующая ограниченного константой времени для исполнения (не зависящего от значений исходных данных).

Компьютерная сеть (*computer network*) – множество компьютеров, соединенных между собой линиями связи, вместе с протоколами взаимодействия.

Литерал (*literal*) – переменная в формуле логики высказываний или ее отрицание

Локальный алгоритм (*stand-alone algorithm*) – алгоритм, спроектированный для реализации одним исполнителем – для выполнения на однопроцессорной ЭВМ.

Массовая задача (*mass problem*) – то же, что и «общая задача».

Масштабируемость (*scalability*) – свойство сети или системы, заключающееся в том, что количество ее пользователей или

узлов может быть значительно увеличено без изменения ее архитектуры.

Машина Тьюринга (*Turing machine*) – один из способов формализации понятия алгоритма, предложенный математиком Алланом Тьюрингом.

Недетерминированный алгоритм (*nondeterministic algorithm*) – алгоритм, в котором после окончания выполнения некоторого шага может выполняться не один строго определенный шаг (свойство детерминированности), а несколько шагов – либо параллельно, либо в произвольном порядке, или произвольно выбираться один из шагов.

Несчетное множество (*non-enumerable set*) – бесконечное множество, неравнозначное множеству натуральных чисел.

Нижняя оценка сложности алгоритма (*lower bound*) – числовая функция от сложности исходных данных, значения которой не могут превышать значение сложности алгоритма при таких же величинах сложности исходных данных; этот факт должен быть доказан математическими средствами.

Общая задача (*generic problem*) – задача, сформулированная таким образом, что ее решение зависит от одного или нескольких заранее не фиксированных параметров (переменных), значения которых могут выбираться из некоторой (описанной или понимаемой по умолчанию) предметной области.

Основное дерево (*spanning tree*) – связный ациклический подграф графа, содержащий все его вершины и, может быть, не все ребра.

Параллельный алгоритм (*parallel algorithm*) – алгоритм, сформулированный таким образом, что некоторые его шаги могут выполнятся одновременно (параллельно во времени) несколькими независимыми исполнителями (процессорами).

Параметризованная сложность (*parameterized complexity*) – запись функции сложности алгоритма в виде формулы, зависящей от двух или более аргументов.

Переборная задача (*brute-force search, brute-force problem*) – задача отыскания объекта в конечной предметной области, удовлетворяющего сформулированным условиям, для которой известны только решения, заключающиеся в проверке этого условия для всех объектов предметной области (в худшем случае).

Последовательный алгоритм (*sequential algorithm*) – алгоритм, который предполагает, что исполнение инструкций должно происходить в строгой последовательности (во времени), в соответствии с тем, как они записаны в алгоритме.

Полиномиальный алгоритм (*polynomial algorithm*) – алгоритм, верхняя граница сложности которого может быть задана полиномом от переменной – параметра сложности исходных данных.

Правильность алгоритма (*correctness of algorithm*) – точное соответствие алгоритма решаемой задаче: при любых значениях исходных данных из заданной области выполнение алгоритма заканчивается получением результата, в точности соответствующего решению задачи.

Предикат (*predicate*) – функция одного или нескольких аргументов, отображающая объекты из предметной области в одно из двух значений, «истина» и «ложь».

Проблема остановки алгоритма (*halting problem*) – проблема определения по описанию (тексту) алгоритма и по значениям исходных данных для алгоритма, остановится ли выполнение этого алгоритма через конечное число шагов.

Проблема эквивалентности алгоритмов (*equivalence problem*) – проблема определения по описаниям (текстам) двух алгоритмов, являются ли эти алгоритмы эквивалентными.

Псевдограф (*pseudograph*) – граф, отличающийся от обычного графа тем, что две вершины могут быть соединены более чем одной дугой, а также вершина может быть смежна сама себе (дуга представляет собой петлю).

Распределенная система (*distributed system*) – система, функционирование которой существенно зависит от удаленности ее элементов друг от друга.

Распределённые вычисления (*distributed computing, grid computing, volunteer computing*) – способ решения трудоёмких вычислительных задач с привлечением большого числа исполнителей, работающих одновременно над разными частями задачи.

Распределенный алгоритм (*distributed algorithm*) – описание взаимодействия локальных алгоритмов, выполняющихся на узлах сети и обменивающихся друг с другом сообщениями.

Расстояние между сайтами в P2P сети (*distance in the P2P topology*) – количество переходов (прыжков – hops), которые нужно сделать между узлами физической сети для передачи информации.

Рекуррентное уравнение (*graph isomorphism*) – уравнение, связывающее между собой значения различных элементов последовательности, часто соседних, и позволяющее вычислить любой элемент последовательности по ее известному началу.

Рекурсия (*recursion*) – механизм сведения решения сложной задачи к решению одной или нескольких более простых задач, подобных исходной; этот механизм используется многократно до тех пор, пока полученные простые задачи не будут тривиальными, после чего из тривиальных решений получается решение исходной задачи.

Сводимость алгоритмическая (*algorithmic reducedness*) – возможность использования для решения задачи алгоритма, разработанного для решения другой задачи путем преобразования исходных данных первой задачи в исходные данные второй задачи, и преобразования результата решения второй задачи в результат решения первой задачи.

Сети ad hoc (*ad hoc networks*) – компьютерные сети, создаваемые «по случаю»; создаются, как правило, в отсутствие инфраструктуры на непродолжительное время для решения некоторой задачи.

Система реального времени (*real-time system*) – система, существующая в некотором физическом окружении, время реакции которой на входные воздействия существенно влияет на качество ее работы; при большом времени реакции качество работы снижается или система теряет работоспособность.

Сложность алгоритма (*algorithm complexity*) – функция, определяющая зависимость количества ресурса вычислителя, требуемого алгоритмом, от параметра сложности исходных данных алгоритма.

Сложность данных (*data complexity*) – количественная оценка объема исходных данных для алгоритма, от которого зависит число шагов при выполнении алгоритма или объем памяти исполнителя, требуемый алгоритмом.

Сложность задачи (*problem complexity*) – функция сложности алгоритма, наиболее быстро решающего задачу; при этом сам наилучший алгоритм может быть неизвестен.

Сосредоточенная система (*stand-alone system*) – в противоположность распределенной системе – система, функционирование которой не зависит от расстояния между элементами.

Средняя оценка сложности алгоритма (*average case complexity*) – математическое ожидание количества элементарных шагов алгоритма до его завершения в предположении о том, что исходные данные для алгоритма выбираются из предметной области случайным образом (как правило, равновероятно).

Счетное множество (*countable set*) – множество, равнозначное множеству натуральных чисел.

Топология компьютерной сети (*computer network topology*) – описание связей между узлами компьютерной сети и, возможно, указание на местоположение узлов (в пространстве).

Торoidalная топология (*scalability*) – топология компьютера или сети, которая может быть изображена на торе в виде специальной системы окружностей.

Формальная арифметика (*formal arithmetic*) – описание арифметики в виде некоторой формальной системы.

Формат команды (*instruction format*) – разбиение записи (кода) команды на элементы (поля) определенного назначения.

Формальная система (*formal system*) – набор аксиом для некоторых математических объектов и правил вывода, позволяющих получать из аксиом новые утверждения.

Цикл (*iteration*) – повторение действий.

Частная задача (*individual problem*) – задача поиска (вычисления) одного или нескольких объектов из предметной области; может быть получена из общей задачи путем фиксации параметров.

Эквивалентность алгоритмов (*equivalency of algorithms*) – два алгоритма эквивалентны, если при любых значениях исходных данных из заданной области их выполнение заканчивается одинаковыми результатами.

Элементарный шаг (*step*) – действие, суть которого очевидна и не требует дополнительного описания (представления) с помощью еще более простых действий.

Ячейка (*cell*) – простейший элемент памяти реальной или абстрактной вычислительной машины.

NP-полная проблема (*NP-complete problem*) – **NP**-трудная задача, принадлежащая классу **NP**.

NP-трудная проблема (*NP-hard problem*) – проблема, к которой другие проблемы из класса **NP** сводятся за полиномиальное время.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

Основная

Королев Л.Н., Миков А.И. Информатика. Введение в компьютерные науки: учеб. для вузов. М.: ООО «Абрис», 2012.

Кузнецов О.П. Дискретная математика для инженера. СПб.: Лань, 2009.

Тель Ж. Введение в распределенные алгоритмы. М.: Изд-во МЦНМО, 2009.

Миков А.И. Распределенные компьютерные системы и алгоритмы: учеб. пособие. Краснодар: Кубан. гос. ун-т, 2009.

Дополнительная

Ахо А.В., Хопкрофт Дж.Э., Ульман Д.Д. Структуры данных и алгоритмы. М.: Вильямс, 2000.

Кнут Д. Искусство программирования. Т.1: Основные алгоритмы. 3-е изд. М.: Вильямс, 2000.

Кнут Д. Искусство программирования. Т.3: Сортировка и поиск. 2-е изд. М.: Вильямс, 2000.

Вирт Н. Алгоритмы и структуры данных: пер. с англ. СПб.: Невский диалект, 2001.

Сигал И.Х., Иванов А.П. Введение в прикладное дискретное программирование: модели и вычислительные алгоритмы. М.: ФИЗМАТЛИТ, 2002.

Макконнелл Дж. Анализ алгоритмов. Активный обучающий подход. М.: Техносфера, 2004.

Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.

Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. М.: Вильямс, 2003.

Миков А.И., Замятина Е.Б. Распределенные системы и алгоритмы: учеб. курс. URL://www.intuit.ru, 2008.

Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: Вильямс, 2005.

Шень А. Программирование: теоремы и задачи. М.: МЦНМО, 1995.

Макконелл Дж. Основы современных алгоритмов. 2-е доп. изд-е. М.: Техносфера, 2004.

Василенко О.Н. Теоретико-числовые алгоритмы в криптографии. М.: МЦНМО, 2003.

Хаггард Г., Шлипф Дж., Уайтсайдс С. Дискретная математика для программистов. М.: БИНОМ. Лаборатория знаний, 2010.

Грэхем Р., Кнут Д., Паташник О. Конкретная математика. Основание информатики. М.: Мир, 1998.

Новиков Ф.А. Дискретная математика для программистов: учеб. пособие для студ. вузов. 2-е изд. СПб.: ПИТЕР, 2004.

Математическая логика и теория множеств
Учебное пособие для вузов
Под ред. А.Н. Склярова

Московский областной технический университет
имени Н.Э. Баумана
Под общим редактором профессора
Ю.А. Борисова

ОГЛАВЛЕНИЕ

Введение	3
1. Сложность алгоритмов с одним исполнителем.....	4
1.1. Понятие сложности алгоритма	4
1.2. Функция сложности циклических алгоритмов	7
1.3. Функция сложности рекурсивных алгоритмов	11
Контрольные вопросы и задания.....	18
Задачи для самостоятельного решения	19
2. Сложность алгоритмов с p исполнителями	20
2.1. Сложность параллельных вычислений	20
2.2. Сложность распределенных вычислений	22
Контрольные вопросы и задания.....	30
Задачи для самостоятельного решения	31
3. Сложность задач	33
3.1. Понятие сложности задачи	33
3.2. Классы сложности	38
3.3. Параметризованная сложность.....	44
Контрольные вопросы и задания.....	46
Задачи для самостоятельного решения	47
4. Существование алгоритмов	49
4.1. Проблема существования алгоритма.....	49
4.2. Машины Тьюринга и алгорифмы Маркова.....	51
4.3. Доказательства несуществования алгоритмов.....	58
Контрольные вопросы и задания.....	63
Задачи для самостоятельного решения	64
Заключение	66
Глоссарий	68
Рекомендуемая литература	76

Учебное издание

М и к о в Александр Иванович
Л а п и н а Ольга Николаевна

**ВЫЧИСЛИМОСТЬ
И СЛОЖНОСТЬ АЛГОРИТМОВ**

Учебное пособие

Дизайн обложки – В.В. Храмцова

Подписано в печать 17.09.13. Печать цифровая.

Формат 60×84 1/16. Уч.-изд. л. 5,5.

Тираж 200 экз. Заказ № 1576.

Кубанский государственный университет
350040, г. Краснодар, ул. Ставропольская, 149.

Издательско-полиграфический центр
Кубанского государственного университета
350040, г. Краснодар, ул. Ставропольская, 149.