ВВЕДЕНИЕ

Настоящий конспект лекций предназначен для студентов 1-2 курсов, изучающих курс «Базовые алгоритмы обработки информации» по специальности 2202. При подготовке конспекта лекций были использованы следующие материалы:

1. А. Ахо, Дж. Хопкрафт, Дж. Ульман Построение и анализ вычислительных алгоритмов., Мир, М. 1979.
2. Э. Рейнгольд, Ю. Нивергельт, Н. Део. Комбинаторные алгоритмы: Теория и практика. Мир, М. 1980
3. С. Гудман, С. Хидетниеми. Введение в разработку и анализ алгоритмов. Мир, М. 1981
4. Е. А. Крук, С. В. Семенов, Б.К. Трояновский, С.В. Федоренко Рекуррентные уравнения, Методические указания, С-Пб., 1997
5. А. Ахо, Дж. Хопкрафт, Дж. Ульман. Структуры данных и алгоритмы. Издат. Дом «Вильямс», М.–С.-Пб. –Киев, 2000.
6. Т. Кормен, Ч. Лейзер, Р. Ривест. Алгоритмы: Построение и анализ. МНЦО, М., 2000.

ПОНЯТИЕ АЛГОРИТМА И СЛОЖНОСТЬ АЛГОРИТМОВ

Алгоритм представляет собой строгую систему правил, определяющую последовательность действий над некоторыми объектами. В частности алгоритмом является последовательность действий, выполняемых ЭВМ. Таким образом, подготовка и решение задачи на ЭВМ сводится к *разработке, описанию и выполнению алгоритма.*

Долгое время задачи нахождения “хороших” алгоритмов не привлекали к себе внимания исследователей, так как в большинстве случаев для решения использовался естественный алгоритм (обычно типа перебора). Поиск более хороших алгоритмов не представлял интереса ни для задач малой размерности, ни для задач большой размерности. Этот на первый взгляд парадоксальный факт имеет простое объяснение. В случае малой размерности задачи лучшие алгоритмы ненамного лучше переборного алгоритма, а отсутствие мощных вычислительных машин приводило к тому, что и в случае задач большой размерности эти алгоритмы, так же как и естественный алгоритм, не приводили к решению из-за большого объема вычислений.

Повсеместное применение ЭВМ и быстрый рост их вычислительной мощности существенно изменили ситуацию, так как стало возможным решать задачи большой размерности. Более того, как оказалось, для таких задач различные усовершенствования естественных алгоритмов могут давать существенный выигрыш во времени работы или требуемой памяти.

Таким образом, мы попытаемся найти ответы на следующие два вопроса:

- Если дана задача, то как найти для ее решения эффективный алгоритм?
- А если алгоритм найден, то как сравнить его с другими алгоритмами, решающими ту же задачу?

Для оценки сложности алгоритмов существует много критериев. Чаще всего нас будет интересовать порядок роста необходимых для решения задачи времени и емкости памяти при увеличении объема входных данных. Прежде всего, нам хотелось бы связать с каждой конкретной задачей некоторую величину, называемую ее размером, которая выражала бы меру количества входных данных.

Рассмотрим в качестве примера задачу сортировки. В качестве входных данных выступает список элементов, подлежащих сортировке, а в качестве выходного результата – те же самые элементы, отсортированные в порядке возрастания или

убывания. Например, входной список 2,1,3,1,5,8 будет преобразован в выходной список 1,1,2,3,5,8 (отсортирован в порядке возрастания). Мерой объема входной информации или размером задачи будет число элементов n, подлежащих сортировке или длина списка. Для нашего примера $n = 6$.

Размером задачи умножения матриц может быть наибольший размер матриц-сомножителей, размер задачи нахождения наибольшего общего делителя двух чисел – длина двоичного представления наибольшего из чисел и т. д.

Время, затрачиваемое алгоритмом, как функция размера задачи, называется *временной сложностью алгоритма.* Аналогично можно определить *емкостную сложность* (емкость памяти, требуемую для решения задачи как функцию размера задачи).

Говорят, что время выполнения программы (временная сложность) имеет порядок $T(n)$ от входных данных размера n. Например, временная сложность некоторой программы $T(n) = cn^2$, где c – некоторая константа. Отметим, что размер n – это всегда целое положительное число или ноль. Однако для многих программ время выполнения действительно является функцией входных данных, а не их размера. Например, пусть имеется база данных размера n, содержащая сведения о продающихся квартирах. В данном случае под размером задачи мы понимаем число записей в базе. Если запрос пользователя – типичный, то соответствующих записей в базе данных содержится много, если же запрос – достаточно редкий, то поиск не займет много времени, особенно если начать его с редко встречающегося атрибута. В этом случае мы определяем $T(n)$ как *время выполнения в наихудшем случае*, т.е. как максимум времени решения задачи по всем входным данным размера n. Мы также будем рассматривать $T_{ср}(n)$ как среднее (в статистическом смысле) время выполнения по всем входным данным размера n.

Поведение временной (емкостной) сложности в пределе при увеличении размерности задачи называется *асимптотической временной (емкостной) сложностью*. Именно асимптотическая сложность алгоритма определяет в итоге размер задачи, которую можно решить этим алгоритмом. Для описания асимптотической скорости роста функций используется O-символика. Если говорят, что время выполнения $T(n)$ некоторой программы имеет порядок $O(n^2)$ (читается “о большое от n в квадрате”), то подразумевается, что существует положительная константа c и размер n_0 такие, что при всех $n \geq n_0$ выполняется неравенство $T(n) \leq cn^2$.

Пример. Пусть $T(0) = 1$, $T(1) = 4$ и в общем случае $T(n) = (n+1)^2$. Покажем, что $T(n)$ имеет порядок $O(n^2)$. Положим $n_0 = 1$ и $c = 4$, легко проверить, что при $n \geq 1$ имеет место неравенство $(n+1)^2 \leq 4n^2$.

Будем говорить, что $T(n)$ имеет порядок $O(f(n))$, если существуют константа c и размер n_0 такие, что при всех $n \geq n_0$ выполняется неравенство $T(n) \leq cf(n)$.

Пример. Пусть $T(n) = 3n^3 + 2n^2$. Тогда, выбирая $n_0 = 0$ и $c = 5$, получаем $T(n) = 3n^3 + 2n^2 \leq 5n^3$ при $n \geq 0$, т.е. $T(n)$ имеет порядок $O(n^3)$. Можно показать, что $T(n)$ имеет порядок $O(n^4)$, но это более слабое утверждение, чем $O(n^3)$.

Пример. Покажем, что функция 3^n не может иметь порядок $O(2^n)$. Пусть существуют константы c и n_0 такие, что имеет место неравенство $3^n \leq c2^n$. Тогда получаем, что $c \geq (3/2)^n$ при всех $n \geq n_0$. Функция $(3/2)^n$ принимает любое, как угодно большое значение при достаточно большом n, поэтому не существует константы, которая могла бы ограничивать $(3/2)^n$ при всех n.

Когда говорят, что $T(n)$ имеет скорость роста $O(f(n))$, то подразумевается, что $f(n)$ является верхней границей роста $T(n)$. Чтобы указать нижнюю границу скорости роста $T(n)$ используется обозначение $\Omega(g(n))$ (читается “омега большое” от $g(n)$). При этом подразумевает существование такой константы c, что при всех n выполняется неравенство $T(n) \geq cg(n)$.

Пример. Пусть $T(n) = n^3 + 2n^2$. Покажем, что эта функция имеет нижнюю границу $\Omega(n^3)$. Положим $c = 1$, тогда $T(n) \geq cn^3$ при всех n.

Может показаться, что рост скорости вычислений на ЭВМ делает задачу нахождения эффективных алгоритмов бессмысленной. На деле оказывается, что происходит в точности противоположное и значение эффективных алгоритмов становится все больше. Так как ЭВМ работают все быстрее, и мы можем решать задачи все большей размерности, то именно сложность алгоритма определяет то увеличение размера задачи, которое можно достичь с увеличением скорости работы ЭВМ.

Пример. Пусть у нас есть 5 алгоритмов $A_1 \ldots A_5$ со следующими временными сложностями:

Алгоритм	Временная сложность
A_1	n
A_2	$n \log_2 n$
A_3	n^2
A_4	n^3
A_5	2^n

Предположим, что следующее поколение ЭВМ будет в 100 раз быстрее нынешнего. Временная сложность – это число единиц времени, требуемого для обработки входа размера n. Посмотрим, как возрастут размерности задач, которые мы сможем решить благодаря этому увеличению скорости при использовании каждого из приведенных выше алгоритмов.

Алгоритм	Временная сложность	Макс. размер задачи до ускорения	Макс. размер задачи после ускорения
A_1	n	n_1	$100n_1$
A_2	$n \log_2 n$	n_2	$\approx 100n_2$ для больших n_2
A_3	n^2	n_3	$10n_3$
A_4	n^3	n_4	$4.65n_4$
A_5	2^n	n_5	$n_5 + 6.64$

Из приведенной таблицы следует, что чем меньше степень роста времени выполнения, тем больше возрастает размер задачи, которая может быть решена данным алгоритмом при увеличении производительности компьютера. Наибольшее увеличение размерности задачи мы получаем для алгоритмов A_1 и A_2, для остальных размер задачи увеличивается незначительно. Для алгоритма A_5 при больших n_5 размерность задачи практически не изменяется.

Теперь вместо эффекта увеличения скорости рассмотрим эффект от применения более действенного алгоритма. Оценим, как изменяется размер задачи, которую можно решить алгоритмами $A_1 \dots A_5$ за 1 минуту и 1 час.

Алгоритм	Временная сложность	Размер задачи за 1 минуту	Размер задачи за 1 час
A_1	n	n_1	$60n_1$
A_2	$n\log_2 n$	$\approx n_1 / \log_2 n_1$	$\approx 60n_1 / (\log_2 n_1 + 5.9)$
A_3	n^2	$\sqrt{n_1}$	$7.74\sqrt{n_1}$
A_4	n^3	$\sqrt[3]{n_1}$	$3.9\sqrt[3]{n_1}$
A_5	2^n	$\log_2 n_1$	$\log_2 n_1 + 5.9$

Из таблицы следует, что различие между алгоритмами с различной асимптотической сложностью становится еще более существенным при увеличении времени вычислений.

Мы получили, что если пренебречь константами в формулах для временной сложности, то алгоритм с временной сложностью порядка $O(n^2)$ всегда лучше, чем алгоритм с временной сложностью $O(n^3)$. Попробуем теперь сравнить два алгоритма, один из которых имеет временную сложность $100n^2$, а второй – $5n^3$. Какой из этих алгоритмов лучше? При размере входных данных $n < 20$ алгоритм с $T(n) = 5n^3$ лучше, чем алгоритм с $T(n) = 100n^2$. Таким образом, для данных небольшого объема следует отдать предпочтение алгоритму с $T(n) = 5n^3$, однако при возрастании n отношение времени $5n^3 / 100n^2 = n/20$ тоже растет, и при больших n следует отдать предпочтение алгоритму с $T(n) = 100n^2$. В следующей таблице сравнение алгоритмов делается с учетом констант в формулах для временной сложности. В последней графе приведены размеры задач, для которых каждый из алгоритмов оказывается наилучшим.

Алгоритм	$f(n)$	$cf(n)$	Размер задачи за 1 час
A_1	n	$1000n$	$n > 1024$
A_2	$n\log_2 n$	$100n\log_2 n$	$59 < n \le 1024$
A_3	n^2	$10n^2$	$10 \le n \le 58$
A_4	n^3	n^3	
A_5	2^n	2^n	$2 \le n \le 9$

Большой порядок роста сложности алгоритма может иметь меньшую мультипликативную составляющую, чем малый порядок роста сложности другого алгоритма. В таком случае алгоритм с быстро растущей сложностью может оказаться предпочтительнее для задач с малым размером. Мультипликативные составляющие зависят от многих факторов, таких как особенности самого алгоритма, особенности компилятора и компьютера и т.д. Еще одно обстоятельство, которое следует принимать во внимание при разработке алгоритмов – это размер машинного слова. Если допустить, что машинное слово имеет фиксированную длину, то возникают трудности даже просто запоминания произвольно больших чисел. Теория алгоритмов не рассматривает влияние этих многочисленных факторов, так как учесть их влияние чрезвычайно сложно, и многие из них существенным образом связаны с развитием технологической базы

вычислительной техники. Основное внимание уделяется разработке и исследованию алгоритмов, имеющих малый порядок роста сложности. Однако разработчик программ при выборе того или иного алгоритма несомненно должен принимать во внимание все сопутствующие факторы.

АЛГОРИТМЫ ПОДСЧЕТА ЧИСЛА ЕДИНИЦ В ДВОИЧНОМ НАБОРЕ

Двоичные последовательности, т.е. последовательности из нулей и единиц, являются носителями информации фактически во всех современных вычислительных устройствах. Целые десятичные числа представлены в компьютере в двоичной системе счисления, т.е. в виде последовательностей нулей и единиц, а арифметические и логические операции над ними представляют собой операции над двоичными последовательностями. Рассмотрим задачу подсчета числа единиц в двоичной последовательности $B = b_n,\ b_{n-1},\ \ldots, b_2,\ b_1$ длины n. Простейший алгоритм решения этой задачи состоит в последовательном просмотре каждого элемента последовательности и, если он равен единице, то счетчику дается соответствующее приращение. Блок-схема простейшего алгоритма приведена на Рис. 1.

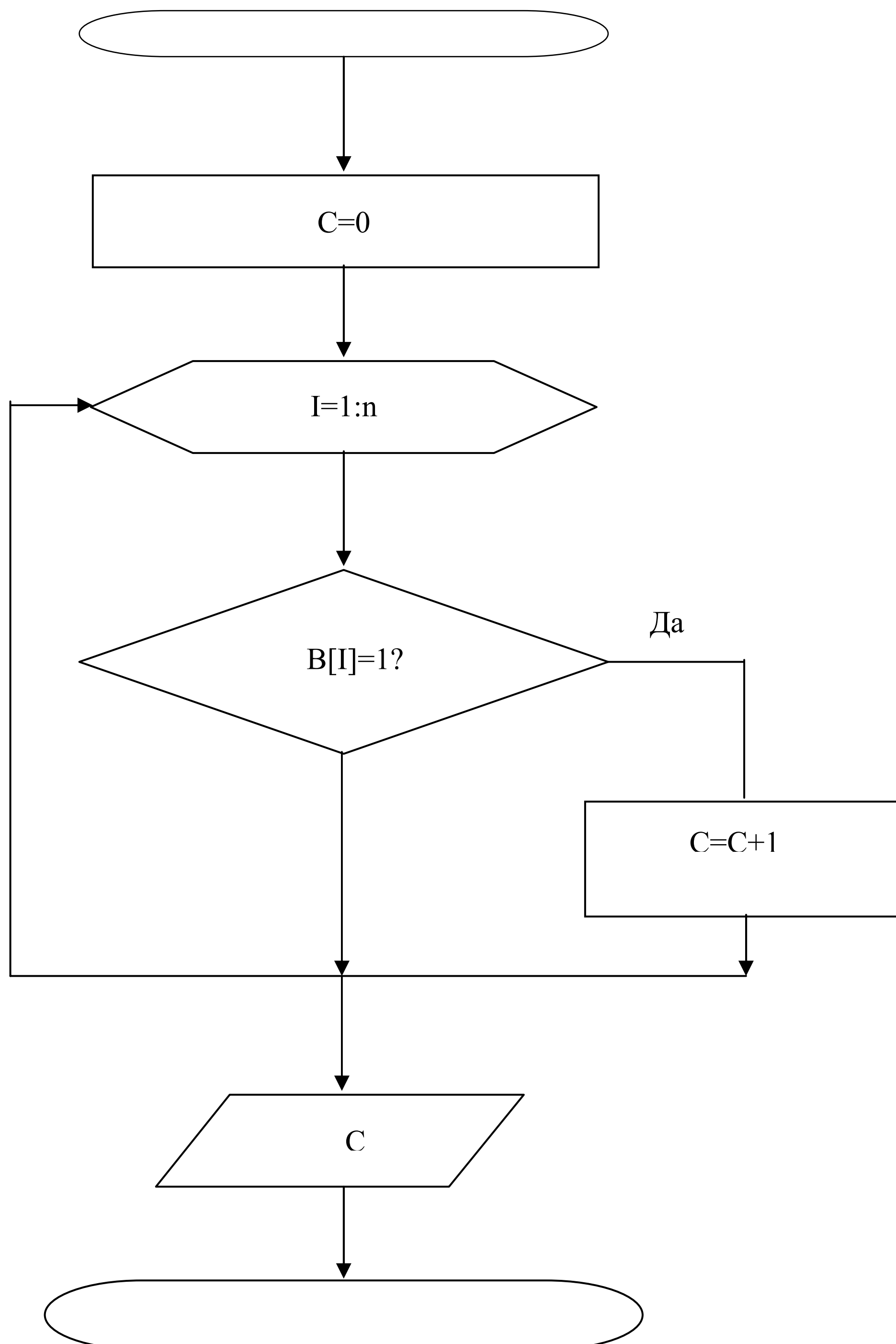

Рис.1 Блок-схема алгоритма подсчета числа единиц в двоичной последовательности

Временная сложность алгоритма, блок-схема которого приведена на Рис. 1 имеет порядок n, т.е. $T(n) = cn$. Рассмотрим более “хитрый” алгоритм подсчета числа единиц в двоичной последовательности. Блок-схема “хитрого” алгоритма подсчета числа единиц в двоичной последовательности приведена на Рис.2.

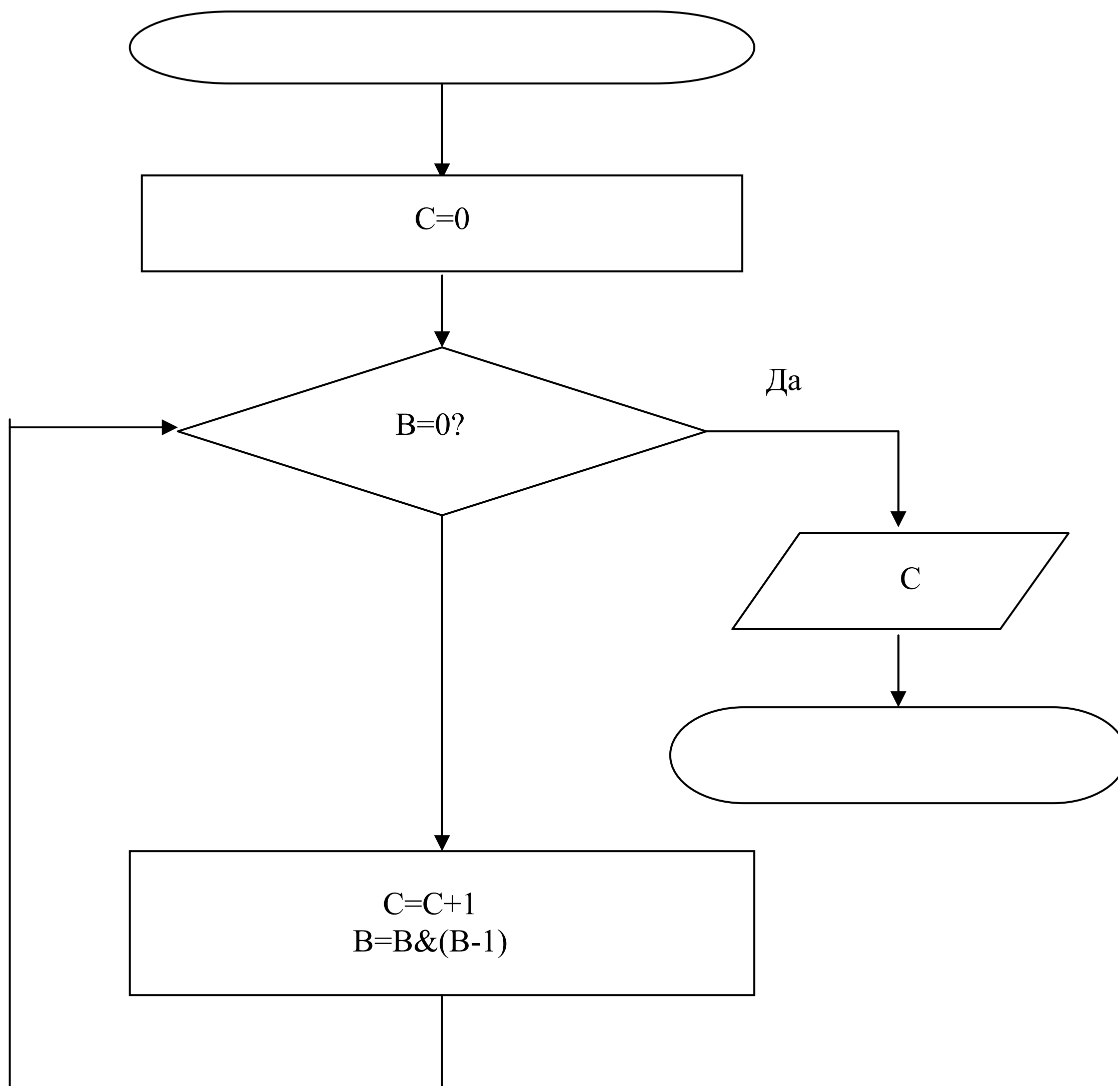

Рис.2. “Хитрый” способ подсчета числа единиц в двоичной последовательности

Мы приняли во внимание, что память компьютера состоит из ячеек, которые могут хранить двоичные слова длины n и что компьютер может выполнять логические и булевы операции параллельно над каждым разрядом слова. Мы также предположили, что компьютер может производить арифметические операции, которые интерпретируют эти слова как беззнаковые целые неотрицательные числа, записанные в двоичной системе счисления.

Пусть $B = 100010101$. Алгоритм, приведенный на Рис.2 рассматривает двоичную последовательность как число, т.е. $B = 277$. Так как $B \neq 0$ мы наращиваем счетчик $C = 1$ и формируем новое значение $B = 100010101 \& 100010100 = 100010100$, т.е. $B = 276$. Отметим, что логическая операция & оперирует параллельно с каждыми двумя соответствующими разрядами пары аргументов B и $B-1$. Операция “-“ – это арифметическая операция вычитания над двоичными целыми числами. Так как $B \neq 0$ опять наращиваем счетчик $C = 2$ и формируем новое значение

$B = 100010100 \& 100010011 = 100010000$, т.е. $B = 272$. Продолжая, получаем $C = 3$, $B = 100010000 \& 100001111 = 100000000$, $B = 256$ и $C = 4$, $B = 100000000 \& 011111111$, $B = 0$, что приводит к завершению алгоритма. Таким образом, мы получили, что число единиц в двоичной последовательности равно 4.

На примере хорошо видно, что операция $B \& (B-1)$ заменяет самую правую единицу в B нулем. Цикл в “хитром” алгоритме повторяется до тех пор пока B не станет равным нулю, т.е. не будет состоять из одних нулей. В то время как цикл в первом алгоритме повторяется n раз, в “хитром” алгоритме он повторяется столько раз, сколько единиц в двоичной последовательности. Очевидно, что этот алгоритм эффективен, когда он применяется к “разреженным” последовательностям, т.е. последовательностям с малым числом единиц. В худшем случае его временная сложность имеет порядок $O(n)$, но средняя временная сложность меньше. Алгоритм зависит от способа представления чисел в ЭВМ.

Следующий алгоритм имеет сходство с простейшим алгоритмом (Рис.1) в том смысле, что цикл повторяется фиксированное число раз, зависящее от n и не зависящее от B. В то же время, в отличие от алгоритма 1, где цикл повторяется n раз, в этом алгоритме цикл повторяется только $\lceil \log_2 n \rceil$ раз. Другими словами временная сложность этого алгоритма $T(n) = c\lceil \log_2 n \rceil$. Для типичного слова длины 32 (или 64) цикл повторяется 5 (соответственно 6) раз, т.е. алгоритм 3 является существенно более быстрым, чем алгоритм 1. Теперь мы предполагаем, что в нашем вычислительном устройстве существует способ быстро сдвигать слово на k разрядов. Блок-схема алгоритма 3 представлена на Рис.3.

Рассмотрим пример. Пусть $B = 11000101$. Сначала выделим все разряды с нечетными номерами, т.е. b_7, b_5, b_3, b_1 и припишем слева от каждого разряда 0. Полученное слово имеет вид

$$B_1 = 0\,\mathbf{1}\,0\,\mathbf{0}\;\;0\,\mathbf{1}\,0\,\mathbf{1}.$$

Затем выделим четные разряды b_8, b_6, b_4, b_2 , сдвинем их вправо на один разряд на места разрядов b_7, b_5, b_3, b_1 , к каждому из разрядов припишем слева ноль. Полученная таким образом последовательность имеет вид

$$B_2 = 0\,\mathbf{1}\;\;0\;\;\mathbf{0}\;\;0\,\mathbf{0}\;\;0\,\mathbf{0}.$$

Складываем два числа, двоичными представлениями которых являются B_1 и B_2, получаем

$$B_1 = B_1 + B_2 = 10000101.$$

Из полученного слова B_1 выделяем пары разрядов $b_6^{(1)}, b_5^{(1)}$ и $b_2^{(1)}, b_1^{(1)}$ и слева от каждой пары приписываем по два нуля. Полученная таким образом последовательность имеет вид

$$B_{21} = 00\,\mathbf{00}\;\;00\,\mathbf{01}.$$

Затем выделим и сдвинем вправо на два разряда другие пары $b_8^{(1)}, b_7^{(1)}$ и $b_4^{(1)}, b_3^{(1)}$. Слева от каждой пары приписываем по два нуля. Полученная последовательность имеет вид

$$B_{22} = 00\,\mathbf{10}\;\;00\,\mathbf{01}.$$

Складываем два числа двоичными представлениями, которых являются B_{21} и B_{22}, получаем

$$B_{21} = B_{21} + B_{22} = 00100010.$$

Возьмем разряды $b_4^{(2)}, b_3^{(2)}$, $b_2^{(2)}$, $b_1^{(2)}$ из полученного числа и припишем к ним слева 4 нуля. Полученную последовательность обозначим через B_{31}

$$B_{31} = 0000\,\mathbf{0010}.$$

Точно также берем разряды $b_8^{(2)}, b_7^{(2)}, b_6^{(2)}, b_5^{(2)}$, сдвигаем их вправо на 4 разряда на места $b_4^{(2)}, b_3^{(2)}, b_2^{(2)}, b_1^{(2)}$ соответственно, приписываем к ним слева 4 нуля и получаем последовательность

$$B_{32} = 0000\ \mathbf{0010.}$$

Теперь складываем два числа, двоичными представлениями которых являются B_{31} и B_{32}, получаем

$$B_{31} = B_{31} + B_{32} = 00000100\,.$$

Нетрудно видеть, что последовательность B_{31}, которая представляет собой двоичную запись результата суммирования, одновременно является двоичной записью суммы разрядов слова B и дает нам требуемый ответ 4.

Отметим, что если длина слова n не является степенью двойки, то к слову слева приписывают нули, чтобы длина стала ближайшей степенью двойки.

Фактически последний из рассмотренных алгоритмов представляет собой одну реализацию общего подхода к решению комбинаторных задач, называемого *принципом декомпозиции.* Принцип декомпозиции приводит нас к решению задачи о сложном объекте путем разложения ее на какое-то число меньших, решению той же задачи для меньших объектов, а потом композиции найденных решений. Для того, чтобы обеспечить окончание процедуры, достаточно для простых объектов прекратить процесс декомпозиции и решить задачу непосредственно для них.

Рассмотрим, как был применен принцип декомпозиции к решению нашей задачи. Пусть $B = b_n, b_{n-1}, ..., b_1$ – двоичный набор длины n, и пусть $S(B)$ – сумма единиц в нем. Зададимся некоторым h и разделим B на две части $B_r = b_n, b_{n-1}, ..., b_{h+1}$ и $B_l = b_h, ..., b_2, b_1$. Число $S(B)$ удовлетворяет следующему рекуррентному уравнению

$$S(B) = S(B_l) + S(B_r)\,.$$

Это означает, что $S(B)$ можно получить за одну операцию сложения, если знать число единиц в наборах меньшей длины. Повторяя такие же рассуждения для последовательностей B_l и B_r, а затем для подпоследовательностей, которые они порождают и т.д., мы получим процесс вычисления $S(B)$ при условии, что мы можем подсчитать число единиц в коротких наборах. Так как $S(B) = B$ при $n = 1$, то очевидно, что, начиная с $n = 1$, мы можем последовательно вычислить $S(B)$ и что сложность соответствующего алгоритма подсчета числа единиц в последовательности зависит от способа разбиения B на подпоследовательности.

Рассмотрим две стратегии разбиения. В первой выберем $h = n-1$, тогда $B_l = b_n$, а $B_r = b_{n-1}, b_{n-2}, ..., b_1$. В этом случае последовательно слева направо просматриваются все разряды. При такой стратегии требуется $n-1$ сложение, фактически мы получили алгоритм 1. Во втором случае положим $h = \lceil n/2 \rceil$. Тогда $B_l = b_n, b_{n-1}, ..., b_{\lceil n/2 \rceil + 1}$ и $B_r = b_{\lceil n/2 \rceil}, ..., b_2, b_1$. Этот способ разбиения, при котором одинаково обрабатываются обе части последовательности и B всегда разбивается примерно пополам, приводит к алгоритму 3. Однако приведенная стратегия разбиения не избавляет нас от необходимости выполнить $n-1$ сложение. Для того, чтобы получить логарифмическую временную сложность алгоритма необходимо, чтобы несколько операций сложения в коротких подпоследовательностях производились путем применения единственной операции в длинных. В алгоритме 3 $S(B)$ – не только сумма единиц в последовательности B, но и слово, представляющее собой двоичную запись числа единиц в B, дополненное слева нулями до соответствующей длины. Если мы имеем операции (такие, как сложение, сдвиг и т.д.), которые обрабатывают двоичные

последовательности длины n, то одна такая операция может интерпретироваться как выполнение многих одинаковых операций над более короткими последовательностями, на которые разбита последовательность B. Действительно, сложение B_1 и B_2 соответствует одновременному вычислению числа единиц в последовательностях длины 2, $B_{21} + B_{22}$ соответствует определению числа единиц в последовательностях длины 4 и т. д.

Существует ли еще более быстрый алгоритм вычисления числа единиц в двоичной последовательности? Существует ли оптимальный алгоритм этого типа? Для того, чтобы ответить на вопрос об оптимальности алгоритма необходимо четко определить класс допустимых алгоритмов и критерий оптимальности. В нашем случае достаточно трудно четко определить класс допустимых алгоритмов, так как мы должны принимать во внимание особенности работы вычислительного устройства. Однако приведенный далее алгоритм несомненно является самым быстрым. Он использует таблицу и получает результат за одну операцию. Платой за такую скорость является расточительное использование памяти, алгоритм требует 2^n ячеек для последовательности длины n. Фактически в этом случае происходит уменьшение временной сложности за счет увеличения емкостной.

Алгоритм 4 основан на идее, что можно предварительно решить задачу для всех последовательностей длины n, хранить в памяти полученные результаты и затем искать среди них. В случае $n = 3$ мы должны хранить таблицу следующего вида

Слово	Число единиц
000	0
001	1
010	1
011	2
100	1
101	2
110	2
111	3

При поиске будем считать B адресом ячейки, содержащей сумму единиц в B.

Таким образом, сравнивая приведенные алгоритмы подсчета числа единиц в двоичной последовательности можно сделать следующие выводы. Алгоритмы 1 и 4 дают решение задачи в “лоб”. При этом алгоритм 1 имеет большую временную сложность, а алгоритм 4 требует большого объема памяти. Алгоритм 3 является изящным компромиссом между ними.

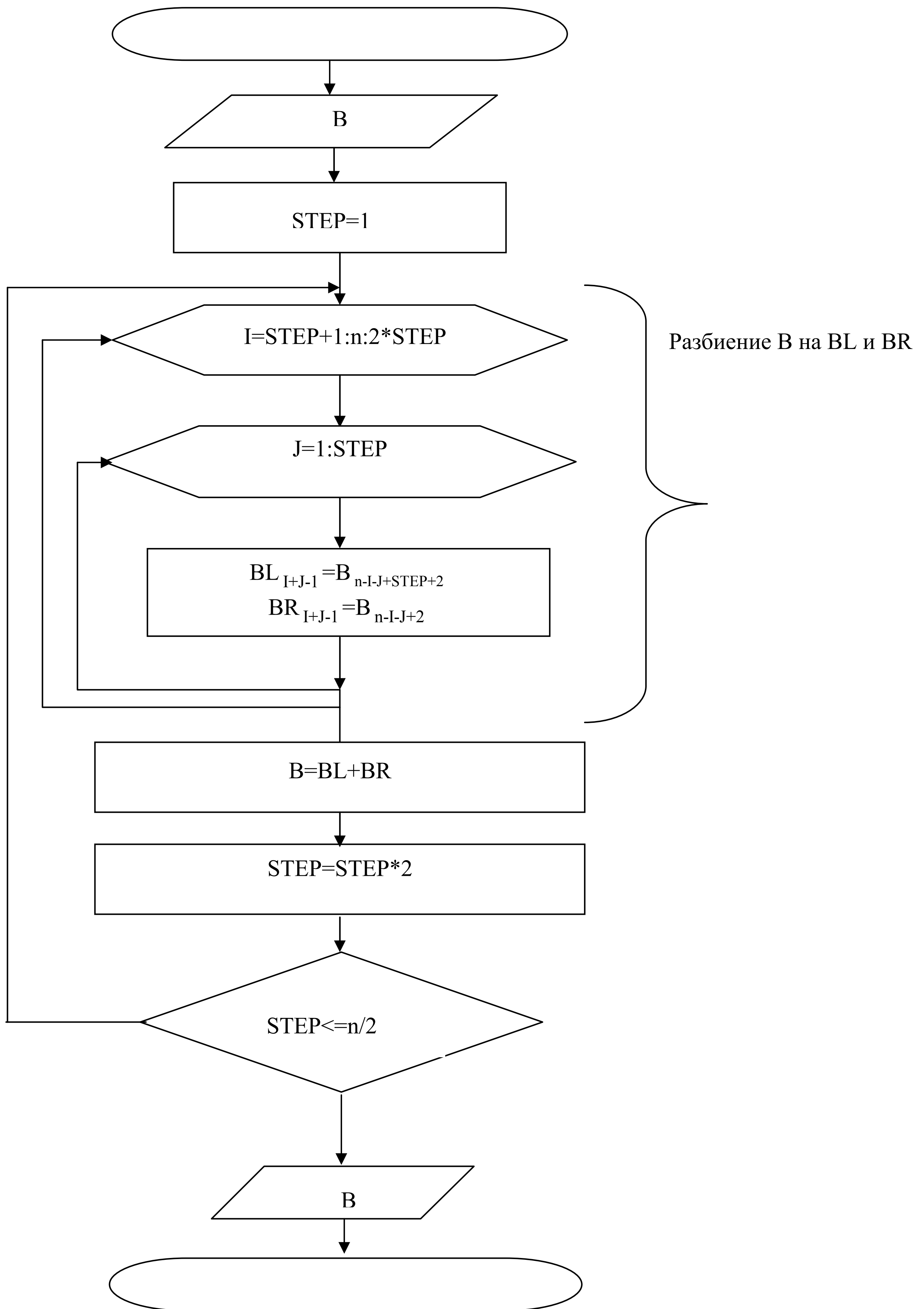

Рис. 3. Алгоритм 3