Лекция 12

Знание, какой алгоритм следует применить при том или ином наборе обстоятельств, может привести к очень большой разнице в производительности вашего программного обеспечения. Пусть эта книга станет вашим учебником, из которого вы узнаете о ряде важных групп алгоритмов, например о таких, как сортировка и поиск. Мы введем ряд общих подходов, применяемых алгоритмами для решения задач, например подход “разделяй и властвуй” или жадная стратегии. Вы сможете применять полученные знания для повышения эффективности разрабатываемого вами программного обеспечения.

Структуры данных жестко связаны с алгоритмами с самого начала развития информатики. Из этой книги вы узнаете о фундаментальных структурах данных, используемых для надлежащего представления информации для ее эффективной обработки.

АЛГОРИТМЫ

Можно дать такое неформальное определение: алгоритм — это последовательность команд для *исполнителя*, обладающая свойствами:

- **полезности**, то есть умения решать поставленную задачу;
- **детерминированности**, то есть строгой определённости каждого шага во всех возможных ситуациях;
- **конечности**, то есть способности завершаться для любого множества входных данных;
- **массовости**, то есть применимости к разнообразным входным данным;
- **корректности**, то есть получения верных результатов для всех допустимых входных данных.

Список свойств алгоритмов можно продолжать и далее, но сейчас мы отметили те свойства, на которые будем обращать в дальнейшем особое внимание.

Каждый алгоритм для своего исполнения (ещё говорят — *вычисления*) требует от исполнителя некоторых *ресурсов*. *Программа* есть запись алгоритма на формальном языке.

Одну и ту же задачу зачастую можно решить несколькими способами, несколькими алгоритмами, которые могут отличаться использованием ресурсов, таких, как *элементарные действия* и *элементарные объекты*. Например, исполнитель алгоритма «компьютер использует устройство *цен- тральный процессор* для исполнения таких элементарных действий, как сложение, умножение, сравнение, переход и других, и устройство *оперативная память* как хранителя элементарных объектов (целых и вещественных чисел). Способность алгоритма использовать ограниченное количество ресурсов называется *эффективностью*.

Сложность алгоритма

Если мы спросим у специалиста по алгоритмам, какая сложность у предложенного им алгоритма, он задаст встречный вопрос: а какую сложность вы имеете в виду?

Если требуется реализовать алгоритм в виде схемы вычислительного устройства, реализующего конкретную функцию, то комбинационная сложность определит минимальное число конструктивных элементов для реализации этого алгоритма. Описательная сложность есть длина описания алгоритма на некотором формальном языке. Один и тот же алгоритм на различных языках может иметь различную описательную сложность. Например, одна строка описания алгоритма на языке Python может быть эквивалентна нескольким десяткам строк алгоритма на языке Pascal. Нас, как составителей алгоритма, больше всего будет интересовать вычислительная сложность, определяющая количество элементарных операций, исполняемых алгоритмом для каких-то входных данных. Для алгоритмов, не содержащих циклов, описательная сложность примерно коррелирует с вычислительной. Если алгоритмы содержат циклы, то прямой корреляции нет, и нас интересует другая корреляция — времени вычисления от входных данных, причём обычно интересна именно асимптотика этой зависимости. Давайте введём понятие главный параметр (мы его будем обычно обозначать буквой N), наиболее сильно влияющий на скорость исполнения алгоритма. Это может быть, например, размер массива при его обработке, количество символов в строке, количество бит в записи числа. Если нам приходится выделять несколько таких параметров, то постараемся создать функцию от них, определяющую один обобщённый параметр. Для определения вычислительной сложности алгоритма (а здесь и далее под термином сложность будет подразумеваться именно вычислительная сложность) введена специальная нотация. Мы опустим здесь строгие математические определения используемых символов O и Θ и дадим их неформально.

В дальнейшем изложении материала мы будем под термином сложность понимать именно вычислительную сложность, если об этом не будет сказано особо.

Определение 1. *Функция* $f(N)$ *имеет порядок сложности* $\Theta(g(N))$*, если существуют постоянные* c_1*,* c_2 *и* N_1*, такие, что для всех* $N > N_1$ $0 \le c_1 g(N) \le f(N) \le c_2 g(N)$.

$\Theta(f(n))$ — *класс функций, примерно пропорциональных* $f(n)$.

На графике это выглядит таким образом: коэффициенты c_1 и c_2, умноженные на функцию $g(n)$, приводят к тому, что график функции $f(n)$ оказывается зажат между графиками функций $c_1 g(n)$ и $c_2 g(n)$.

Например, если мы о каком-то алгоритме сказали, что он имеет сложность $\Theta(N^2)$, где-то при больших N функция сложности будет неотличима от функции cN^2, где c — константа, которую ещё называют *коэффициент амортизации*.

Определение 2. *Функция* $f(N)$ *имеет порядок сложности* $O(g(N))$*, если существуют постоянные* c_1 *и* N_1*, такие, что для всех* $N > N_1$ $f(N) \le c_1 g(N)$

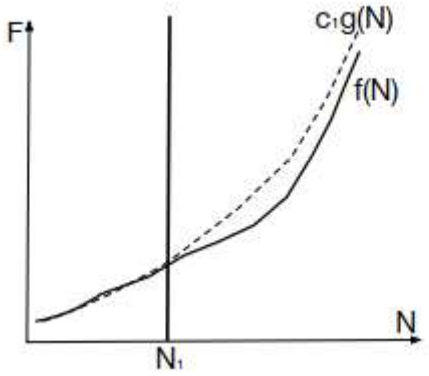


Рис. 1.1. Сложность O(N)

Пусть $f(N)$ — функция сложности алгоритма в зависимости от N: Тогда если существует такая функция $g(N)$ (асимптотическая функция) и константа C, что

$$\lim_{N\to\infty} \frac{f(N)}{g(N)} = C,$$

то сложность алгоритма $f(N)$ определяется функцией $g(N)$ с коэффициентом амортизации C.

Говорят, что символ $\Theta(f(n))$ определяет класс функций, примерно пропорциональных $f(n)$, а символ $O(f(n))$ — класс функций, ограниченных сверху $cf(n)$.

Класс сложности алгоритма определяется по асимптотической зависимости $a(N)$.

• Экспонента с любым коэффициентом превосходит любую степень.
• Степень с любым коэффициентом, большим единицы, превосходит логарифм по любому основанию, большему единицы.
• Логарифм по любому основанию, большему единицы, превосходит 1.

Можно привести несколько примеров:

$F(N) = N3 + 7N2 - 14N = \Theta(N3)$:
$F(N) = 1{:}01N + N10 = \Theta(1{:}01N)$:
$F(N) = N1{:}3 + 10 \log 2\ N = \Theta(N1{:}3)$:

Однако, не стоит делать поспешные выводы о том, что алгоритм $A1$ с асимптотической сложностью $\Theta(N2)$ заведомо хуже алгоритма $A2$ с асимптотической сложностью $\Theta(N \log N)$: Вполне может оказаться так, что для небольших значениях N количество операций, требуемых для исполнения алгоритма $A1$ может оказаться меньше (и существенно), чем для алгоритма $A2$.

Время исполнения алгоритма, исчисляемое в элементарных операциях, может отличаться для различных входных данных. Рассмотрим элементарный пример.

Пример: поиск в массиве

Задача. Пусть имеется массив *A* длиной *N* элементов. Найти номер первого вхождения элемента со значением *P* .

Сколько операций потребуется, чтобы обнаружить искомый номер с помощью алгоритма, заключающегося в последовательном просмотре элементов массива?

Самый первый элемент массива может оказаться P, следовательно, минимальное количество операций поиска будет $K_{min} = 1$. Элемента в массиве может не быть совсем, и тогда для поиска потребуется ровно $K_{max} = N$ операций. А какое среднее значение количества поисков? Предполагая, что количество итераций алгоритма. требуемых для поиска, равномерно распределено по всем числам от 1 до N, получаем:

$$K_{avg} = \frac{\sum_{i=1}^{N} i}{N} = \frac{N \times (N+1)}{2N} = \frac{N+1}{2}$$

Можно ли сказать, что алгоритм имеет сложность порядка Θ(*N*) в общем случае? Нет, в наилучшем случае *f*(*n*) = 1 совсем не зависит от *N*. Мы можем сказать, что в лучшем случае алгоритм имеет сложность Θ(1), в среднем и в худшем — сложность Θ(*N*). Но для данного алгоритма нам проще будет использовать *O*-нотацию: *f*(*N*) = *O*(*N*).

Все ли задачи можно решить за полиномиальное время? Как выясняется, отнюдь не все. Вот очень простая в формулировке и тем не менее очень сложная задача.

Задача о наполнении рюкзака

Задача. Пусть имеется *N* предметов, каждый из которых имеет объём *Vi* и стоимость *Ci*, предметы неделимы. Имеется рюкзак вместимостью *V* . Требуется поместить в рюкзак набор предметов максимальной стоимости, суммарный объём которых не превышает объёма рюкзака.
Решение задачи. Как оказывается, задача не имеет решения с полиномиальной сложностью. Один из простых в реализации неполиномиальных по сложности алгоритмов заключается в следующем:

1. Перенумеруем все предметы.
2. Установим максимум достигнутой стоимости *M* в 0.
3. Составим двоичное число с *N* разрядами, в котором единица в разряде будет означать, что предмет выбран для укладки в рюкзак. Это число однозначно определяет расстановку предметов.
4. Рассмотрим все расстановки, начиная от 000 *: : :* 000 до 111 *: : :* 111. Для каждой из них подсчитаем значение суммарного объёма V_M . (а) Если суммарный объём расстановки *VM* не превосходит объёма рюкзака *V* , то подсчитывается суммарная стоимость W_M и сравнивается с достигнутым ранее максимумом стоимости *M*.

(b)Если вычисленная суммарная стоимость превосходит максимум M, то максимум M устанавливается в вычисленную стоимость WM и запоминается текущая конфигурация.

Алгоритм, как нетрудно убедиться, обладает всеми требуемыми свойствами: он *детерминирован*, так как его поведение зависит исключительно
от входных данных; он *конечен*, так как его исполнение неизбежно прекратится, как только будут исчерпаны все расстановки; он *массовый*, так как он решает все задачи этого класса, и он *полезный*, так как даёт нам решение конкретной задачи.

Его сложность пропорциональна $2N$, так как требуется перебрать все возможные перестановки (мы не рассматриваем тривиальный вариант, когда все N предметов помещаются в рюкзак).

Много ли времени потребуется на решение задачи для $N = 128$? Предположим, на подсчёт одного решения потребуется 10^{-9} секунд, то есть, одна наносекунда. Предположим, задачу будет решать триллион компьютеров (10^{12}). Тогда общее время решения задачи будет составлять

$$\frac{2^{128} \times 10^{-9}}{10^{12}} \text{секунд} \approx 10.8 \times 10^{9} \text{лет}.$$

Это — пример задачи, которая имеет решение не полиномиальной сложности (NP), но до сих пор не найдено решение полиномиальной сложности (P). Мало того, не доказано, что она может иметь решение полиномиальной сложности. Однако проверить, удовлетворяет ли какое-либо предложенное решение условию корректности, можно за полиномиальное время. Имеется понятие *сертификат* решения, который явным образом определяет предложенное решение. Например, в рассмотренной нами задаче с рюкзаком, сертификатом может быть значение последовательности из N двоичных цифр. Точное решение подобных задач (а задача о рюкзаке относится к классу *NP-сложных* задач) требует времени, превышающего все мыслимые значения. Мы должны понимать, что такие задачи существуют, и что для них лучше искать *приближённое* решение, которое может оказаться не таким сложным, пусть и не таким хорошим.

Рекурсия

С XII века известны числа Фибоначчи *f*0*;* 1*;* 1*;* 2*;* 3*;* 5*;* 8*;* 13*;* 21*;* 34*; : : : g* и способ их получения по правилу

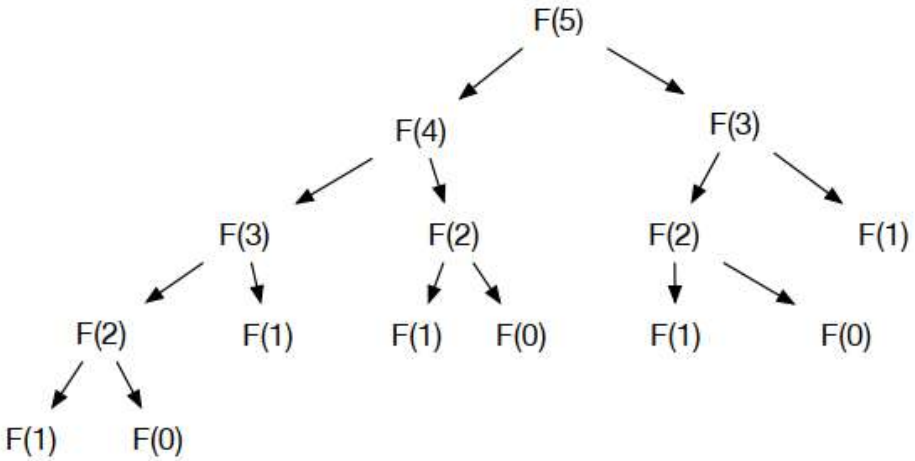
$$F_n = \begin{cases} 0, & \text{если n} = 0, \\ 1, & \text{если n} = 1, \\ F_{n-1} + F_{n-2}, & \text{если } n > 1. \end{cases}$$

Это — рекуррентный способ записи последовательностей. Часто

последовательности задаются только таким способом. Такой алгоритм добывания элементов последовательности Фибоначчи очень легко запрограммировать.

```
def fibonacci(n):
    if n in (1, 2):
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Каково время его работы? Этот вопрос сложнее. Взглянем на *дерево вызовов* этой функции



Попытаемся оценить количество вызовов этой функции. Количество вызовов функции для $n = 0$ и $n = 1$ равно одному. Обозначив количество вызовов через $t(n)$, получаем $t(0) > F(0)$ и $t(1) = F(1)$. Для $n > 1$ $t(n) = t(n - 1) + t(n - 2) > F(n)$.

Следовательно, при рекурсивной реализации алгоритма количество вызовов превосходит число Фибоначчи для соответствующего n. Сами же числа Фибоначчи удовлетворяют отношению

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \Phi, \qquad \Phi = \frac{\sqrt{5}+1}{2},$$

то есть, $Fn \approx C \times \Phi n$. Сложность этого алгоритма есть $\Theta(\Phi N)$

Как же так, алгоритм прост, но почему так медленно исполняется? Проблема в том, что мы много раз повторно вычисляем значение функции от одних и тех же аргументов. Требуемая для исполнения память характеризует *сложность алгоритма по памяти*.

• Каждый вызов функции создаёт новый *контекст функции* или *фрейм вызова*.

• Каждый фрейм вызова содержит все аргументы, локальные переменные и служебную информацию.

• Максимальное количество фреймов, которое создаётся, равно глубине рекурсии.

• Сложность алгоритма по занимаемой памяти равна $O(N)$.

Представление чисел в алгоритмах

Оценим, насколько затратен такой переход от абстрактных чисел к их представлениям.

В реальных программах имеются ограничения на операнды машинных команд. X86,X64 int есть 32 бита, long long есть 64 бита. На 32-битной архитектуре сложение двух 64-разрядных сложение младших разрядов и прибавление бита переноса к сумме старших разрядов. Две или три машинных команды.

X86: сложение: 32-битных ≈ 1 такт; 64-битных ≈ 3 такта. X64: сложение: 32-битных ≈ 1 такт; 64-битных ≈ 1 такт. X86: умножение: 32-битных ≈ 3-4 такта; 64-битных ≈ 15-50 тактов. X64: умножение: 32-битных ≈ 3-4 такта; 64-битных ≈ 4-5 тактов. Команды деления целых чисел и нахождения целочисленного остатка на современных компьютерах весьма долго исполняются. Мы будем их использовать только в тех случаях, когда без этого в алгоритме не обойтись.

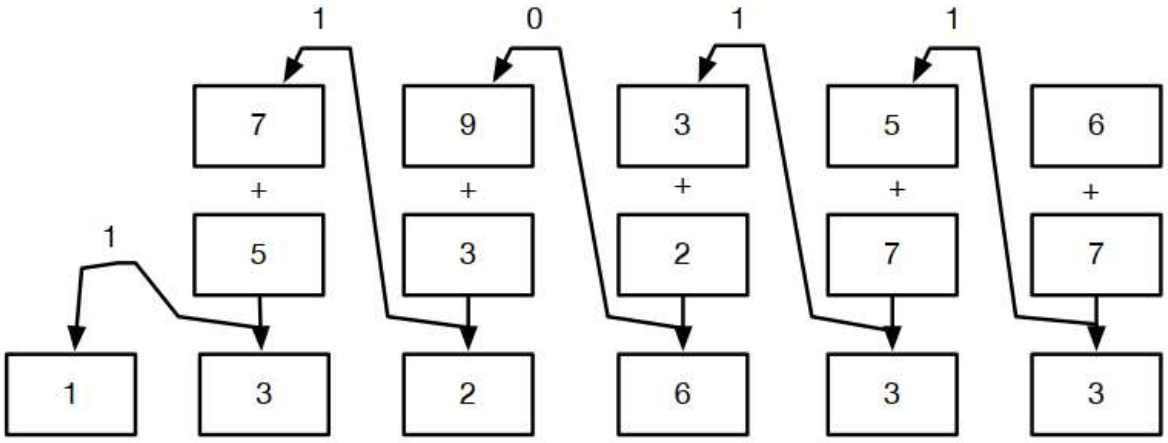
Представление длинных чисел

Представлять такие числа можно многими способами, но, для удобства вычислений, мы воспользуемся привычной нам позиционной системой счисления, правда, с необычным основанием.

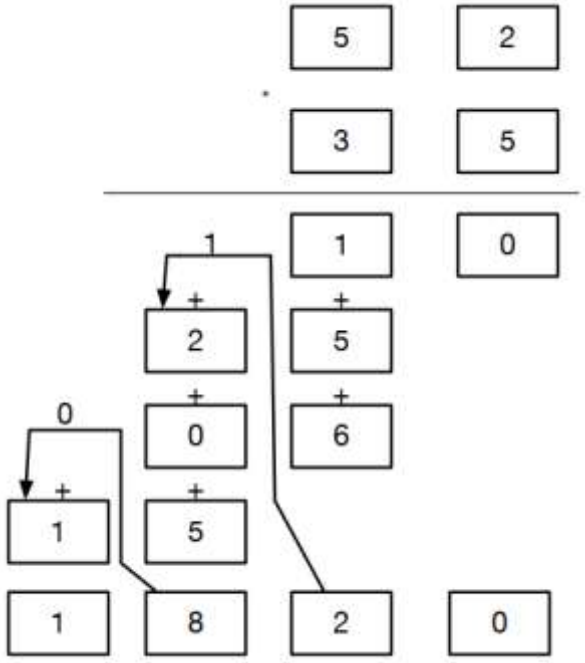
Так как длинные числа имеют представление в виде *цифр*, представляющих удобный для нас тип данных в позиционной системе счисления, то все операции будут производиться в этой системе счисления. Мы привыкли использовать по одному знаку на десятичную цифру, то есть использовать десятичную систему. Аппаратному исполнителю удобнее работать с длинными числами в системе счисления по основаниям, большим 10 (2^8; 2^{16}; 2^{32}; 2^{64}).

Определение. *(n)-числа — те числа, которые требуют не более n элементов элементарных типов (цифр) в своём представлении.*

Например, если за элементарный тип данных для представления на 32-битной архитектуре мы выберем тип int, то long long будут (2)-числами. (n)-числа, где n>2 в исполнительной системе языков Си и C++ уже представления не имеют. Представление длинных чисел требует массивов элементарных типов. Основание системы счисления R для каждой из цифр представления должно быть представимо элементарным типом данных аппаратного исполнителя. Сколько операций потребуется для сложения двух (n)-чисел? Похоже, что оптимальнее школьного алгоритма сложения «в столбик» придумать что-либо трудно.



Каждую цифру первого числа нужно сложить с соответствующей цифрой второго и учесть перенос из соседнего разряда. Сложность алгоритма составляет $O(n)$*:* Школьный алгоритм умножения длинных чисел тоже на первый взгляд кажется оптимальным: мы умножаем первое число на каждую из цифр второго, на что требуется n операций умножения, затем складываем все n промежуточных результатов, что даёт нам $O(n^2)$ операций умножения и $O(n^2)$ операций сложения.



Можно ли умножать быстрее?

Мы будем полагать, что существует операция умножения двух 32-битных чисел, дающая 64-битное число, и это — одна операция. Именно поэтому мы не переходим к 64-разрядным числам как к элементарным единицам, потому что нам тогда потребуется операция умножения двух 64-разрядных чисел с получением 128-разрядного результата. Некоторые компиляторы имеют такую встроенную функцию, некоторые — нет. Будем ориентироваться на более слабые компиляторы и оставим использование 64-битной арифметики как резерв для дальнейшей оптимизации.

Итак, наивный алгоритм умножения длинных чисел (n) имеет сложность $O(N2)$. К счастью, имеется более быстрый алгоритм. Он изобретён в 1960-х годах аспирантом А. Н. Колмогорова Анатолием Карацубой и с тех пор является неизменным участником любых библиотек работы с большими числами. Нас он интересует постольку, поскольку реализует принцип Цезаря — *разделяй и властвуй*. Пусть нам требуется перемножить два (2n)-числа. Введём константу T, на единицу большую максимального числа, представляемого (n)-числом. Тогда любое (2n)-число X можно представить в виде суммы $Tx_u + x_l$. Это разложение имеет сложность $O(n)$, так как оно заключается просто в копировании соответствующих разрядов (n)-чисел.

$$N_1 = Tx_1 + y_1$$
$$N_2 = Tx_2 + y_2$$

При умножении в столбик

$$N_1 \times N_2 = T^2 x_1 x_2 + T((x_1 y_2 + x_2 y_1) + y_1 y_2.$$

Это — четыре операции умножения и три операции сложения. Число T определяет, сколько нулей нужно добавить к концу числа в соответствующей системе счисления, и мы полагаем сложность этой операции равной $O(1)$. Алгоритм Карацубы находит произведение по другой формуле:

$$N_1 \times N_2 = T^2 x_1 x_2 + T((x_1 + y_1)(x_2 + y_2) - x_1 x_2 - y_1 y_2) + y_1 y_2.$$

Рассмотрим алгоритм на примере произведения чисел 56 и 78, приняв T за 10 (мы хотим получить ответ в десятичной системе счисления).

$$x_1 = 5, y_1 = 6$$
$$x_2 = 7, y_2 = 8$$
$$x_1 x_2 = 5 \times 7 = 35$$
$$(x_1 + y_1)(x_2 + y_2) = (5 + 6)(7 + 8) = 11 * 15 = 165$$
$$y_1 y_2 = 6 \times 8 = 48$$
$$N_1 \times N_2 = 35 * 100 + (165 - 35 - 48) * 10 + 48 = 3500 + 920 + 48 = 4368$$

Мы уменьшили число операций умножения за счёт увеличения операций сложения. Принесёт ли это нам выгоду? На этот вопрос нам поможет ответить основная теорема о рекурсии.

Основная теорема о рекурсии

А всё-таки, как определить, какой порядок сложности будет иметь рекурсивная функция, не проводя вычислительных экспериментов? Так как рекурсия есть разбиение задачи на подзадачи с последующей *консолидацией* результата, обозначим количество подзадач, на которые разбивается задача за a. Пусть размер каждой подзадачи уменьшается в b раз и становится [n/b]. Пусть сложность консолидации после решения подзадач есть $O(n^d)$. Тогда

сложность такого алгоритма, выраженная рекуррентно, есть

$$T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d).$$

Епрощённый вариант основной теоремы о рекурсии, которую мы даём здесь без доказательств, звучит так:

Теорема 1. *Пусть* $T(n) = aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d)$ *для некоторых* $a > 0, b > 1, d \geqslant 0$. *Тогда*

$$T(n) = \begin{cases} O(n^d), & \text{если } d > \log_b a, \\ O(n^d \log n), & \text{если } d = \log_b a, \\ O(n^{\log_b a}), & \text{если } d < \log_b a. \end{cases}$$

Рассуждая неформально, можно заметить, что общая сложность алгоритма есть сумма членов геометрической прогрессии с знаменателем $q = a / b^d$ При $q < 1$ она сходится и её сумма оценивается через первый член, при $q > 1$ она расходится и её сумма оценивается через её последний член, а при $q = 1$ все члены (а их $O(\log n)$) равны. Оценим сложность алгоритма Карацубы по этой теореме. Коэффициент a порождения задач здесь равен трём, так как одна первичная операция «большого» умножения требует трёх операций «маленького» умножения. Коэффициент уменьшения размера подзадачи $b = 2$, мы делим число на две примерно равные части. Консолидация решения производится за время $O(n)$, так как она заключается в операциях сложения и вычитания (которые имеют сложность $O(n)$ и их строго определённое количество), следовательно, $d = 1$. Так как $1 < \log_2 3$, то в действие вступает третий случай теоремы и, следовательно, сложность алгоритма есть $O(N^{3/2})$.

Операция умножения чисел (n) при умножении в столбик имеет порядок сложности $O(n^2)$. Наличие большого количества операций сложения и вычитания говорит о том, что для малых (n) алгоритм может исполняться дольше, чем «школьный», и поэтому в реальных программах рекурсию стоит ограничить. Например, в большинстве библиотек длинной арифметики алгоритм Карацубы используется при достаточно больших значениях n.