1.3. Определение времени работы алгоритмов

Время работы алгоритма – это число единиц времени, необходимого для обработки входа размера n. Если известно время выполнения каждой операции алгоритма, то для определения времени работы алгоритма в целом достаточно определить число выполняемых им операций.

В качестве примера рассмотрим несколько алгоритмов подсчета числа единиц в двоичном наборе $B = b_n, b_{n-1}, \ldots, b_2, b_1$ длины n (задача и алгоритмы заимствованы из [21]).

Первый способ представлен алгоритмом 1.1. Для более простой записи алгоритма можно было бы использовать оператор цикла **for**, но в данном случае специально применен оператор цикла **while**, чтобы показать скрытые в **for** операции. Для каждой строки алгоритма указаны время t_i выполнения (некоторая фиксированная величина, не зависящая от n, т. е. $t_i = O(1)$) и число ее повторений (x – неизвестное число повторений, зависящее от входной последовательности). Ясно, что размером входа является длина n двоичного набора.

	Время t_i	Число повторений
$count \leftarrow 0$	t_1	1
$k \leftarrow 1$	t_2	1
while $k \le n$ **do**	t_3	$n + 1$
if $b_k = 1$	t_4	n
then $count \leftarrow count + 1$	t_5	x
$k \leftarrow k + 1$	t_6	n

Алгоритм 1.1. Первый способ подсчета числа единиц в двоичном наборе

Сложив время t выполнения операций с учетом числа их повторений, можно получить общее время работы алгоритма

$$T(n) = t_1 + t_2 + t_3 + n(t_3 + t_4 + t_6) + x t_5,$$

где x равно числу единиц в исходном наборе, $0 \le x \le n$.

Подобный детальный анализ алгоритмов трудно применить на практике. Величина t_i показывает только то, что время выполнения i-й операции постоянно и равно t_i, но конкретное ее значение, зависящее от многих факторов (используемый компьютер и набор его машинных команд, качество компиляции и т. п.), обычно определить невозможно. Поэтому используют менее точные оценки, ограничиваясь определением констант пропорциональности, или более общие оценки эффективности, использующие асимптотические обозначения и игнорирующие константы пропорциональности.

Если для алгоритма 1.1 принять время выполнения каждой операции равным единице, то время его работы $T(n) = 3n + x + 3$. Поскольку $0 \le x \le n$, т. е. $x = O(n)$, асимптотическая оценка $T(n) = 3n + O(n) + 3 = O(n)$.

В случаях, когда определение констант пропорциональности вызывает проблемы или не ставится задача их определения, асимптотическую оценку можно получить, используя правила сложения и умножения асимптотик. Рассмотрим такой анализ на примере алгоритма 1.1. Первые два оператора присваивания имеют некоторое постоянное время выполнения, не зависящее от размера входа, т. е. имеют время выполнения порядка $O(1)$. Время выполнения операторов в теле цикла также имеет порядок $O(1)$. Поскольку цикл выполняется n раз, время его выполнения по правилу умножения асимптотик имеет порядок $O(1n) = O(n)$. В результате временная сложность алгоритма есть $O(1) + O(n) = O(n)$.

Время работы алгоритмов во многих случаях зависит не только от размера входа n, но и от самого входа. Поэтому для таких алгоритмов рассматривают время работы *в худшем случае*, *в среднем* и *в лучшем случае*. При анализе алгоритмов наибольший интерес представляют время работы в худшем случае (определяет максимальное время работы для входов данного размера) и в среднем. Определение среднего времени работы алгоритма часто является математически сложной задачей. Кроме того, эта величина зависит от выбранного распределения вероятностей входов (обычно используется равномерное распределение), которое может отличаться от реального. Время работы в лучшем случае интересно с точки зрения определения характеристик входов, наиболее благоприятных для работы алгоритма.

В алгоритме 1.1 величина x имеет минимальное и максимальное значения, равные соответственно 0 (если входной набор содержит только нули) и n (если входной набор состоит только из единиц). Среднее значение величины x по всем возможным входным наборам при равномерном распределении их вероятностей равно $n/2$. Поэтому время работы алгоритма составляет $T_{\max}(n) = 4n + 3$ в худшем случае, $T_{\text{ave}}(n) = 3{,}5n + 3$ в среднем и $T_{\min}(n) = 3n + 3$ в лучшем случае. Таким образом, временная сложность алгоритма во всех случаях есть $O(n)$.

Второй способ подсчета числа единиц основан на том, что значение разряда в двоичном наборе содержит информацию о числе единиц в этом разряде, и представлен алгоритмом 1.2. С целью сравнения с алгоритмом 1.1 для идентичных операций использованы те же обозначения t_i времени выполнения. Время работы данного алгоритма составляет

$$T(n) = t_1 + t_2 + t_3 + n(t_3 + t_6 + t_7) = 3n + 3 = O(n)$$

и не зависит от входа (зависит только от размера входа), т. е. любые входные наборы обрабатываются за одно и то же время.

	Время t_i	Число повторений
$count \leftarrow 0$	t_1	1
$k \leftarrow 1$	t_2	1
while $k \le n$ **do**	t_3	$n + 1$
$\begin{cases} count \leftarrow count + b_k \\ k \leftarrow k + 1 \end{cases}$	t_7 t_6	n n

Алгоритм 1.2. Второй способ подсчета числа единиц в двоичном наборе

Алгоритмы 1.1 и 1.2 имеют одну и ту же сложность $O(n)$. Если принять во внимание константы пропорциональности, то алгоритм 1.2 более эффективен. Константы пропорциональности были определены при условии, что время выполнения каждой операции равно единице. Для близких по времени работы алгоритмов такое упрощение может оказаться достаточно грубым. Для иллюстрации сказанного определим, при каких условиях алгоритм 1.1 в среднем эффективнее алгоритма 1.2. Пусть $a_1 = t_3 + t_4 + t_5/2 + t_6$, $a_2 = t_3 + t_6 + t_7$, $b = t_1 + t_2 + t_3$. Тогда время работы алгоритма 1.1 в среднем $T_1(n) = a_1 n + b$, а алгоритма 1.2 – $T_2(n) = a_2 n + b$. Из неравенства $T_1(n) \le T_2(n)$ следует, что $a_1 \le a_2$, т. е. $t_4 + t_5/2 \le t_7$. Время t_5 выполнения оператора присваивания $count \leftarrow count + 1$ и время t_7 выполнения оператора присваивания $count \leftarrow count + b_k$ можно связать соотношением $t_7 = t_5 + c$, где c – некоторая константа. Тогда алгоритм 1.1 в среднем эффективнее алгоритма 1.2, если $t_4 \le t_5/2 + c$. Более того, он эффективнее даже в худшем случае, если $t_4 \le c$. Другой вопрос, существует ли такая константа. Это зависит от результатов трансляции, архитектуры процессора и т.д. Вполне может оказаться, что $c = 0$ и, следовательно, алгоритм 1.1 не будет эффективнее алгоритма 1.2.

Третий способ подсчета числа единиц в двоичном наборе представлен алгоритмом 1.3. Основой алгоритма является операция $B \wedge (B-1)$, которая заменяет в наборе B самую правую единицу нулем (здесь B интерпретируется как двоичное представление числа при выполнении операции вычитания и как двоичный код при выполнении операции конъюнкции). Поэтому цикл повторяется до тех пор, пока B не станет равным нулю, т. е. будет состоять только из нулей. Число итераций и будет являться результатом работы алгоритма.

$$count \leftarrow 0$$
$$\textbf{while } B \neq 0 \textbf{ do } \begin{cases} count \leftarrow count + 1 \\ B \leftarrow B \wedge (B-1) \end{cases}$$

Алгоритм 1.3. Третий способ подсчета числа единиц в двоичном наборе

Если предположить, что оператор $B \leftarrow B \wedge (B-1)$ соответствует двум операторам присваивания, т. е. требует две единицы времени, а величина x есть число единиц в наборе B, то время выполнения алгоритма $T(n) = 4x + 2$. Поскольку $0 \le x \le n$, время работы алгоритма составит $T_{max}(n) = 4n + 2$ в худшем случае, $T_{ave}(n) = 2n + 2$ в среднем и $T_{min}(n) = 2$ в лучшем случае. Таким образом, временная сложность алгоритма в худшем случае и в среднем есть $O(n)$, в лучшем случае – $O(1)$. Очевидно, что данный алгоритм в среднем эффективнее алгоритмов 1.1 и 1.2, особенно на наборах с малым числом единиц.

Четвертый способ. Проще рассмотреть на конкретном примере.

	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
B	1	1	1	0	1	0	1	0

1. Сначала выделяются разряды с нечетными номерами и слева от каждого из них приписывают нули (эти приписанные нули выделены мелким шрифтом, чтобы отличить от нулей, входящих в набор). Полученный набор обозначим $B_{\text{неч}}$

		b_7		b_5		b_3		b_1
$B_{\text{неч}}$	o	1	o	0	o	0	o	0

Затем выделяются четные разряды и сдвигаются вправо на один разряд. К каждому из разрядов приписывается слева нуль. Полученный набор обозначим $B_{\text{чет}}$.

		b_8		b_6		b_4		b_2
$B_{\text{чет}}$	o	1	o	1	o	1	o	1

После этого производится сложение двух двоичных чисел, представленных наборами $B_{\text{неч}}$ и $B_{\text{чет}}$. Через B' обозначен результат суммирования.

	b'_8	b'_7	b'_6	b'_5	b'_4	b'_3	b'_2	b'_1
$B_{\text{неч}}$	0	1	0	0	0	0	0	0
$B_{\text{чет}}$	0	1	0	1	0	1	0	1
B'	1	0	0	1	0	1	0	1

2. Из полученного набора выделяются нечетные пары разрядов (b'_6, b'_5) и (b'_2, b'_1) и слева от каждой пары приписываются по два нуля. Полученный набор обозначим $B'_{\text{неч}}$.

			b'_6	b'_5			b'_2	b'_1
$B'_{\text{неч}}$	o	o	0	1	o	o	0	1

Затем выделяются и сдвигаются на два разряда вправо четные пары (b'_8, b'_7) и (b'_4, b'_3). Слева от каждой пары приписываются по два нуля. Получим набор $B'_{\text{чет}}$.

			b'_8	b'_7			b'_4	b'_3
$B'_{\text{чет}}$	o	o	1	0	o	o	0	1

Складываем два числа $B'_{\text{неч}}$ и $B'_{\text{чет}}$. Результат обозначим B''.

	b''_8	b''_7	b''_6	b''_5	b''_4	b''_3	b''_2	b''_1
$B'_{\text{неч}}$	0	0	0	1	0	0	0	1
$B'_{\text{чет}}$	0	0	1	0	0	0	0	1
B''	0	0	1	1	0	0	1	0

3. Выделяется нечетная четверка и приписывается слева четыре нуля, получается набор $B''_{\text{неч}}$.

					b''_4	b''_3	b''_2	b''_1
$B''_{\text{неч}}$	o	o	o	o	0	0	1	0

Затем выделяется четная четверка, сдвигается вправо на четыре разряда и приписывается к ним слева четыре нуля, получается набор $B''_{\text{чет}}$.

					b''_8	b''_7	b''_6	b''_5
$B''_{\text{чет}}$	o	o	o	o	0	0	1	1

Выполняется сложение двух чисел $B''_{\text{неч}}$ и $B''_{\text{чет}}$, получается набор $B''' = (00000101)_2 = 5_{10}$, т. е. набор является двоичной записью суммы разрядов исходного набора B = (11101010), в примере – пять.

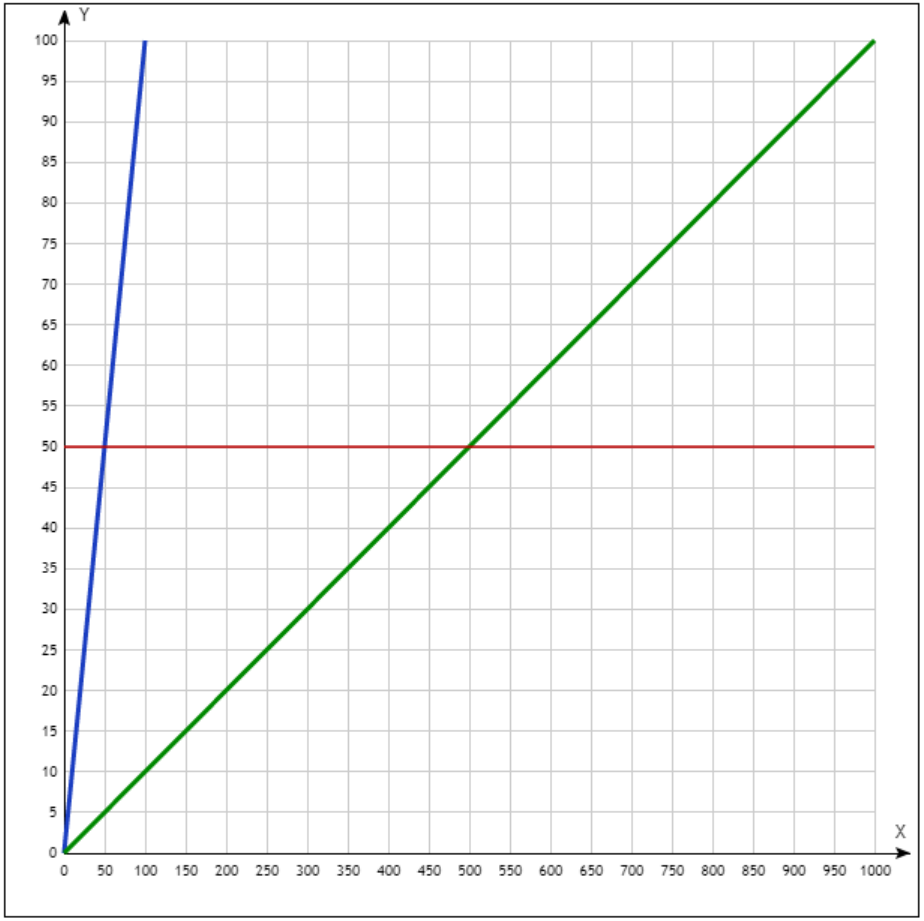
Следует заметить, что если n – не является степенью двойки, то к B предварительно следует приписать слева нули так, чтобы размер стал степенью двойки, ближайшей сверху к n.

Время работы алгоритма $O(\log n)$.

Пятый способ основан на идее, что можно предварительно решить задачу для всех возможных наборов фиксированной длины, хранить в памяти полученные результаты и затем искать среди них нужный набор (*B* будет адресом ячейки памяти, содержащей сумму единиц). Ответ получается за одно обращение к памяти, т. е. $O(1)$. Однако требует памяти $O(2^n)$.

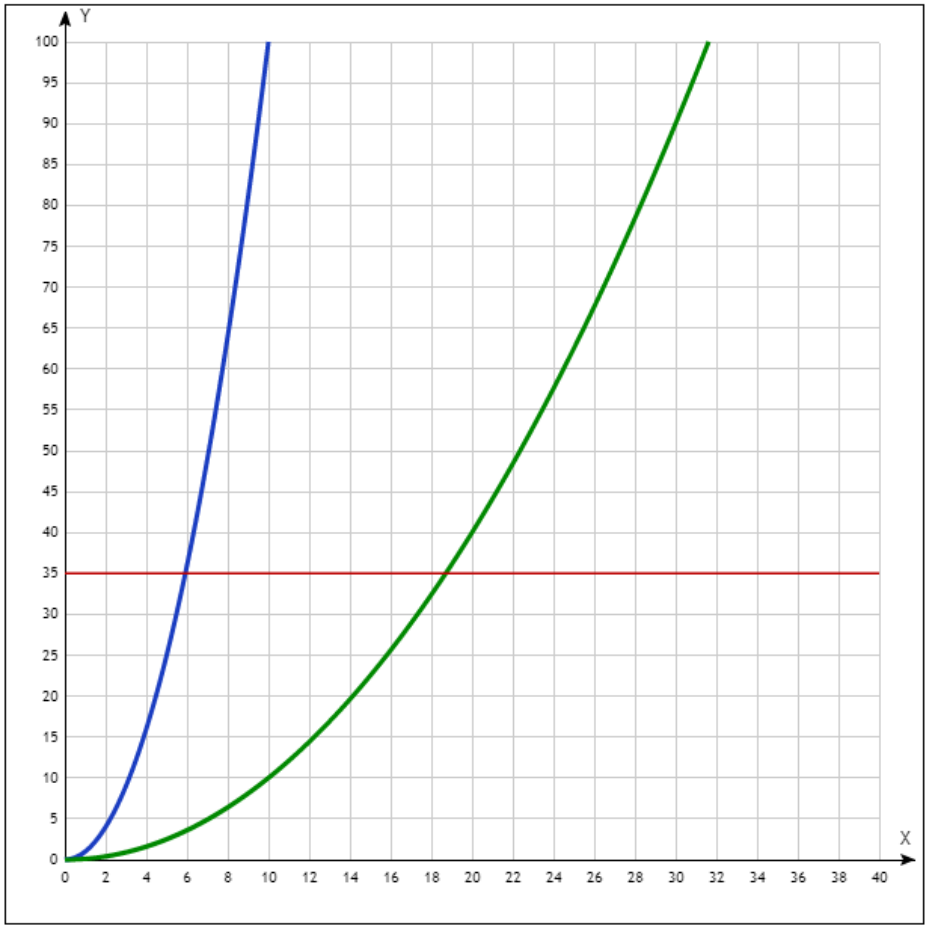
B	Число единиц
000	0
001	1
010	1
011	2
100	1
101	2
110	2
111	3

Различные аспекты использования временной сложности алгоритмов можно проиллюстрировать на следующих примерах. Пусть имеются алгоритмы A_1, A_2 и A_3 с временем выполнения n, n^2 и 2^n соответственно. Для определенности за единицу времени примем одну миллисекунду. Тогда алгоритм A_1 может обработать за одну секунду вход размера 1000, A_2 – вход размера 31, а A_3 – вход размера не более 9. Пусть эти алгоритмы выполняются на компьютере с более высокой производительностью, например, в 10 раз. Тогда для алгоритма A_1 десятикратное увеличение скорости увеличивает размер задачи, которую можно решить, в 10 раз, для A_2 – более чем утраивается, а для A_3 – только на три. Это показывает, насколько важен вопрос выбора наиболее эффективного алгоритма и что асимптотическая сложность является важной мерой эффективности алгоритмов.



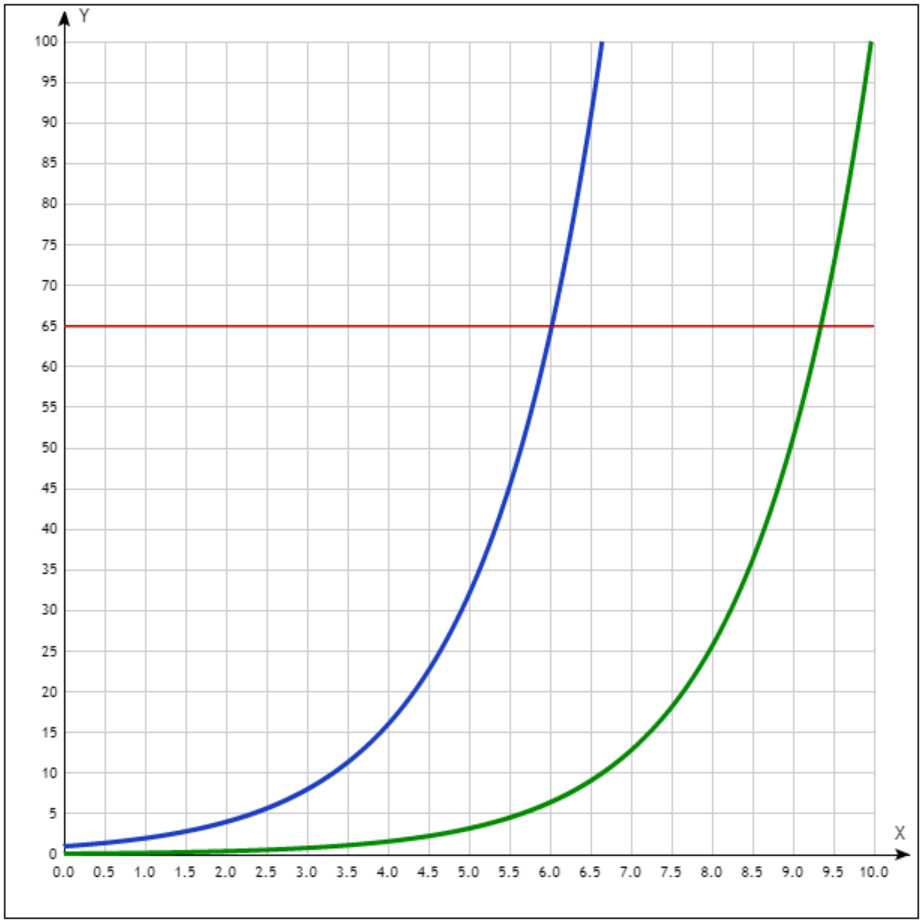
$y(x) = x$

$y(x) = \frac{x}{10}$



$y(x) = x^2$

$y(x) = \frac{x^2}{10}$

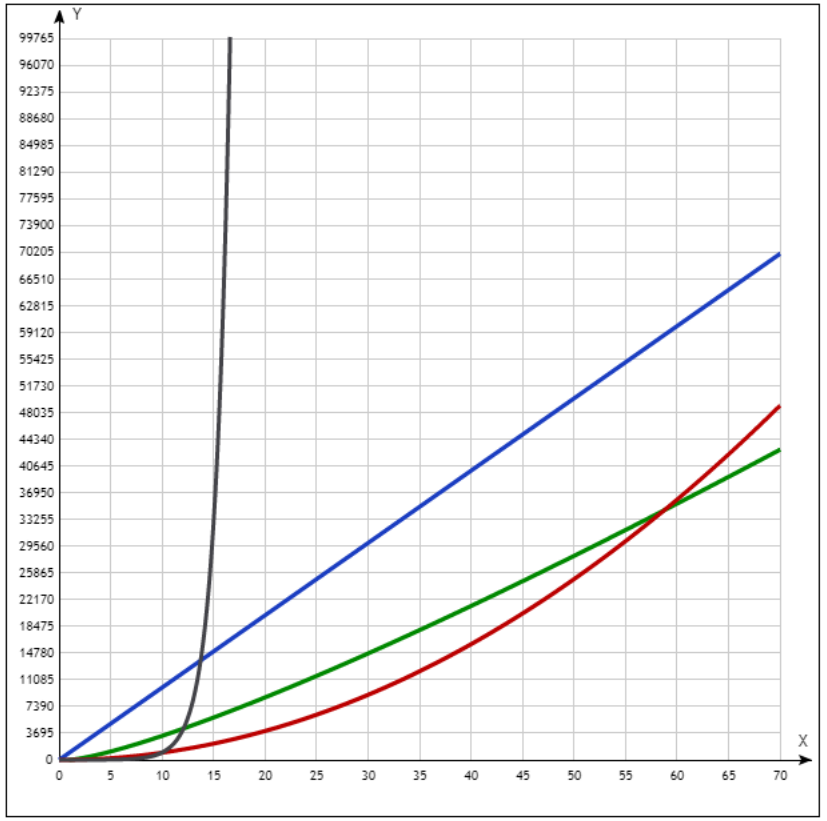


$y(x) = 2^x$

$y(x) = \frac{2^x}{10}$

Таким образом, для сравнения двух алгоритмов необходимо сначала сравнить их асимптотические сложности. Затем, если они окажутся одинаковыми, перейти к определению констант пропорциональности и другим тонкостям. При этом необходимо иметь в виду, что большой порядок роста времени выполнения может иметь меньшую константу пропорциональности, чем малый порядок роста. В таком случае алгоритм с быстро растущей функцией времени выполнения может оказаться предпочтительнее для задач с малым размером.

Пусть, например, имеются алгоритмы A_1, A_2, A_3 и A_4 с временем выполнения соответственно $1000n$, $100n \log n$, $10n^2$ и 2^n. Тогда A_4 будет наилучшим для задач размера $2 \le n \le 9$, A_3 – для задач размера $10 \le n \le 58$, A_2 – при $59 \le n \le 1024$, а A_1 – при $n > 1024$.



$y(x) = 1000x$

$y(x) = 100x \log_2 x$

$y(x) = 10x^2$

$y(x) = 2^x$