ВикипедиЯ

Временная сложность алгоритма

Материал из Википедии — свободной энциклопедии

В информатике **временна́я сложность** алгоритма определяет время работы, используемое алгоритмом, как функции от длины строки, представляющей входные данные [1]. Временная сложность алгоритма обычно выражается с использованием нотации «O» большое, которая исключает коэффициенты и члены меньшего порядка. Если сложность выражена таким способом, говорят об *асимптотическом* описании временной сложности, т.е. при стремлении размера входа к бесконечности. Например, если время, которое нужно алгоритму для выполнения работы, для всех входов длины n не превосходит $5n^3 + 3n$ для некоторого n (большего некоторого n_0), асимптотическая временная сложность равна O (n^3).

Временная сложность зачастую оценивается путём подсчёта числа элементарных операций, осуществляемых алгоритмом, где элементарная операция занимает для выполнения фиксированное время. Тогда полное время выполнения и число элементарных операций, выполненных алгоритмом, отличаются максимум на постоянный множитель.

Поскольку производительность алгоритма может отличаться при входах одного и того же размера, обычно используется временная сложность наихудшего случая поведения алгоритма, которая обозначается как $\boldsymbol{T(n)}$ и которая определяется как максимальное время, которое требуется для любого входа длины n. Реже, и это обычно оговаривается специально, время измеряется как средняя сложность. Сложность по времени классифицируется природой функции $T(n)$. Например, алгоритм с $T(n) = O(n)$ называется алгоритмом *линейного времени*, а об алгоритме с $T(n) = O(M^n)$ и $m^n = O(T(n))$ для некоторого $M \geq m > 1$ говорят как об *алгоритме с экспоненциальным временем*.

Содержание

Таблица сложностей по времени

Следующая таблица суммирует некоторые, обычно рассматриваемые, классы сложности. В таблице poly(x) = $x^{O(1)}$, т.е. многочлен от x.

Название	Класс сложности	Время работы ($T(n)$)	Примеры времени работы	Примеры алгоритмов
постоянное время		$O(1)$	10	Определение чётности целого числа (представленного в двоичном виде)
обратная функция Аккермана от времени		$O(\alpha(n))$		Амортизационный анализ на одну операцию с использованием непересекающихся множеств
повторно логарифмическое время		$O(\log^* n)$		Распределённые раскраски циклов
дважды логарифмическое		$O(\log \log n)$		Время амортизации на одну операцию при использовании ограниченной очереди с приоритетами[2]
логарифмическое время	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Двоичный поиск
полилогарифмическое время		$\text{poly}(\log n)$	$(\log n)^2$	
дробная степень		$O(n^c)$ при $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Поиск в k-мерном дереве
линейное время		$O(n)$	n	Поиск наименьшего или наибольшего элемента в неотсортированном массиве
"n log звёздочка n" время		$O(n \log^* n)$		Алгоритм триангуляции многоугольника Зайделя.
линейно-логарифмическое время		$O(n \log n)$	$n \log n$, $\log n!$	Максимально быстрая сортировка сравнением
квадратичное время		$O(n^2)$	n^2	Сортировка пузырьком, сортировка вставками, прямая свёртка
кубическое время		$O(n^3)$	n^3	Обычное умножение двух $n \times n$ матриц. Вычисление Частичная корреляция.
полиномиальное время	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Алгоритм Кармаркара для линейного программирования, тест простоты числа АКС
квазиполиномиальное время	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Наиболее известный $O(\log^2 n)$-Аппроксимационный алгоритм для ориентированной задачи Штайнера.
подэкспоненциальное время (первое определение)	SUBEXP	$O(2^{n^\varepsilon})$ for all $\varepsilon > 0$	$O(2^{\log n^{\log \log n}})$	Если принять теоретические гипотезы, BPP содержится в SUBEXP.[3]
подэкспоненциальное время (второе определение)		$2^{o(n)}$	$2^{n^{1/3}}$	Наиболее известные алгоритмы разложения на множители целых чисел и изоморфизма графов
экспоненциальное время (с линейной экспонентой)	E	$2^{O(n)}$	1.1^n, 10^n	Решение задачи коммивояжёра с помощью динамического программирования
экспоненциальное время	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	Решение задачи о порядке перемножения матриц с помощью полного перебора
факториальное время		$O(n!)$	$n!$	Решение задачи коммивояжёра полным перебором
дважды	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Проверка верности заданного

экспоненциальное время				утверждения в арифметике Пресбургера

Постоянное время

Говорят, что алгоритм является алгоритмом **постоянного времени** (записывается как время **O(1)**), если значение $T(n)$ ограничено значением, не зависящим от размера входа. Например, получение одного элемента в массиве занимает постоянное время, поскольку выполняется единственная команда для его обнаружения. Однако нахождение минимального значения в несортированном массиве не является операцией с постоянным временем, поскольку мы должны просмотреть каждый элемент массива. Таким образом, эта операция занимает линейное время, O(n). Если число элементов известно заранее и не меняется, о таком алгоритме можно говорить как об алгоритме постоянного времени.

Несмотря на название "постоянное время", время работы не обязательно должно быть независимым от размеров задачи, но верхняя граница времени работы не должна зависеть. Например, задача "обменять значения *a* и *b*, если необходимо, чтобы в результате получили $a \le b$", считается задачей постоянного времени, хотя время работы алгоритма может зависеть от того, выполняется ли уже неравенство $a \le b$ или нет. Однако существует некая константа *t*, для которой время выполнения задачи всегда *не превосходит t*.

Ниже приведены некоторые примеры кода, работающие за постоянное время:

```
int index = 5;
int item = list[index];
if (условие верно) then
   выполнить некоторые операции с постоянным временем работы
else
   выполнить некоторые операции с постоянным временем работы
for i = 1 to 100
   for j = 1 to 200
      выполнить некоторые операции с постоянным временем работы
```

Если $T(n)$ равен O(*некоторое постоянное значение*), это эквивалентно $T(n)$ равно O(1).

Логарифмическое время

Говорят, что алгоритм выполняется за **логарифмическое время**, если $T(n) = \mathbf{O(\log n)}$. Поскольку в компьютерах принята двоичная система счисления, в качестве базы логарифма используется 2 (то есть, $\log_2 n$). Однако при замене базы логарифмы $\log_a n$ и $\log_b n$ отличаются лишь на постоянный множитель, который в записи O-большое отбрасывается. Таким образом, O(log *n*) является стандартной записью для алгоритмов логарифмического времени независимо от базы логарифма.

Алгоритмы, работающие за логарифмическое время, обычно встречаются при операциях с двоичными деревьями или при использовании двоичного поиска.

O(log n) алгоритмы считаются высокоэффективными, поскольку время выполнения операции в пересчёте на один элемент уменьшается с увеличением числа элементов.

Очень простой пример такого алгоритма — деление строки пополам, вторая половина опять делится пополам, и так далее. Это занимает время O(log n) (где n — длина строки, мы здесь полагаем, что *console.log* и *str.substring* занимают постоянное время). Это означает, что для увеличения числа печатей необходимо удвоить длину строки.

```
// Функция для рекурсивной печати правой половины строки
var right = function(str)
{
    var length = str.length;

    // вспомогательная функция
    var help = function(index)
    {
```

```
        // Рекурсия: печатаем правую половину
        if(index < length)
        {

            // Печатаем символы от index до конца строки
            console.log(str.substring(index, length));

            // рекурсивный вызов: вызываем вспомогательную функцию с правой частью
            help(Math.ceil((length + index)/2));
        }
    }
    help(0);
}
```

Полилогарифмическое время

Говорят, что алгоритм выполняется за **полилогарифмическое время**, если $T(n) = O((\log n)^k)$, для некоторого k. Например, задача о порядке перемножения матриц может быть решена за полилогарифмическое время на параллельной PAM-машине[4].

Сублинейное время

Говорят, что алгоритм выполняется за **сублинейное время**, если $T(n) = o(n)$. В частности, сюда включаются алгоритмы с временной сложностью, перечисленные выше, как и другие, например, поиск Гровера со сложностью $O(n^{1/2})$.

Типичные алгоритмы, которые, являясь точными, всё же работают за сублинейное время, используют распараллеливание процессов (как это делают алгоритм NC_1 вычисления определителя матрицы), неклассические вычисления (как в поиске Гровера) или имеют гарантированное предположение о струтуре входа (как работающие за логарифмическое время, алгоритмы двоичного поиска и многие алгоритмы обработки деревьев). Однако формальные конструкции, такие как множество всех строк, имеющие бит 1 в позиции, определяемой первыми log(n) битами строки, могут зависеть от каждого бита входа, но, всё же, оставаться сублинейными по времени.

Термин *алгоритм с сублинейным временем работы* обычно используется для алгоритмов, которые, в отличие от приведённых выше примеров, работают на обычных последовательных моделях машин и не предполагают априорных знаний о структуре входа [5]. Однако для них допускается применение вероятностных методов и даже более того, алгоритмы должны быть вероятностными для большинства тривиальных задач.

Поскольку такой алгоритм обязан давать ответ без полного чтения входных данных, он в очень сильной степени зависит от способов доступа, разрешённых во входном потоке. Обычно для потока, представляющего собой битовую строку $b_1,...,b_k$, предполагается, что алгоритм может за время O(1) запросить значение b_i для любого i.

Алгоритмы сублинейного времени, как правило, вероятностны и дают лишь аппроксимированное решение. Алгоритмы сублинейного времени выполнения возникают естественным образом при исследовании проверки свойств.

Линейное время

Говорят, что алгоритм работает за **линейное время**, или **O(*n*)**, если его сложность равна $O(n)$. Неформально, это означает, что для достаточно большого размера входных данных время работы увеличивается линейно от размера входа. Например, процедура, суммирующая все элементы списка, требует время, пропорциональное длине списка. Это описание не вполне точно, поскольку время работы может существенно отличаться от точной пропорциональности, особенно для малых значений n.

Линейное время часто рассматривается как желательный атрибут алгоритма[6]. Было проведено много исследований для создания алгоритмов с (почти) линейным временем работы или лучшим. Эти исследования включали как программные, так и аппаратные подходы. В случае аппаратного исполнения некоторые алгоритмы, которые, с математической точки зрения, никогда не могут достичь линейного времени исполнения в стандартных моделях вычислений, могут работать за

линейное время. Существуют некоторые аппаратные технологии, которые используют параллельность для достижения такой цели. Примером служит ассоциативная память. Эта концепция линейного времени используется в алгоритмах сравнения строк, таких как алгоритм Бойера — Мура и алгоритм Укконена.

Квазилинейное время

Говорят, что алгоритм работает за квазилинейное время, если $T(n) = \mathbf{O(}n \log^k n\mathbf{)}$ для некоторой константы k. Линейно-логарифмическое время является частным случаем с $k = 1$[7]. При использовании обозначения слабое-O эти алгоритмы являются $\tilde{O}(n)$. Алгоритмы квазилинейного времени являются также $o(n^{1+\varepsilon})$ для любого $\varepsilon > 0$ и работают быстрее любого полинома от n со степенью, строго большей 1.

Алгоритмы, работающие за квазилинейное время, вдобавок к линейно-логарифмическим алгоритмам, упомянутым выше, включают:

- Сортировка слиянием на месте, $O(n \log^2 n)$
- Быстрая сортировка, $O(n \log n)$, в вероятностной версии имеет линейно-логарифмическое время выполнения в худшем случае. Невероятностная версия имеет линейно-логарифмическое время работы только для измерения сложности в среднем.
- Пирамидальная сортировка, $O(n \log n)$, сортировка слиянием, introsort, бинарная сортировка с помощью дерева, плавная сортировка, пасьянсная сортировка, и т.д. в худшем случае
- Быстрые преобразования Фурье, $O(n \log n)$
- Вычисление матриц Монжа, $O(n \log n)$

Линейно-логарифмическое время

Линейно-логарифмическое является частным случаем квазилинейного времени с показателем $k = 1$ на логарифмическом члене.

Линейно-логарифмическая функция — это функция вида $n \cdot \log n$ (т.е. произведение линейного и логарифмического членов). Говорят, что алгоритм работает за **линейно-логарифмическое время**, если $T(n) = \mathbf{O(}n \log n\mathbf{)}$.[8] Таким образом, линейно-логарифмический элемент растёт быстрее, чем линейный член, но медленнее, чем любой многочлен от n со степенью, строго большей 1.

Во многих случаях время работы $n \cdot \log n$ является просто результатом выполнения операции $\Theta(\log n)$ n раз. Например, сортировка с помощью двоичного дерева создаёт двоичное дерево путём вставки каждого элемента в массив размером n один за другим. Поскольку операция вставки в сбалансированное бинарное дерево поиска занимает время $O(\log n)$, общее время выполнения алгоритма будет линейно-логарифмическим.

Сортировки сравнением требуют по меньшей мере линейно-логарифмического числа сравнений для наихудшего случая, поскольку $\log(n!) = \Theta(n \log n)$ по формуле Стирлинга. То же время выполнения зачастую возникает из рекуррентного уравнения $T(n) = 2\,T(n/2) + O(n)$.

Подквадратичное время

Говорят, что алгоритм выполняется за **субквадратичное время**, если $T(n) = o(n^2)$.

Например, простые, основанные на сравнении, алгоритмы сортировки квадратичны (например, сортировка вставками), но можно найти более продвинутые алгоритмы, которые имеют субквадратичное время выполнения (например, сортировка Шелла). Никакие сортировки общего вида не работают за линейное время, но переход от квадратичного к субквадратичному времени имеет большую практическую важность.

Полиномиальное время

Говорят, что алгоритм работает за **полиномиальное время**, если время работы ограничено сверху многочленом от размера входа для алгоритма, то есть $T(n) = O(n^k)$ для некоторой константы k[1][9]. Задачи, для которых алгоритмы с детерминированным полиномиальным временем существуют, принадлежат классу сложности **P**, который является центральным в теории вычислительной сложности. Тезис Кобэма утверждает, что полиномиальное время является синонимом понятий «легко поддающийся обработке», «выполнимый», «эффективный» или «быстрый»[10].

Некоторые примеры алгоритмов полиномиального времени:

- Алгоритм быстрой сортировки n целых чисел делает максимум An^2 операций для некоторой константы A. Таким образом, он работает за $O(n^2)$ и является алгоритмом за полиномиальное время.
- Все базовые арифметические операции (сложение, вычитание, умножение, деление и сравнение) могут быть выполнены за полиномиальное время.
- Максимальные паросочетания в графах можно найти за полиномиальное время.

Строго и слабо полиномиальное время

В некоторых контекстах, особенно в оптимизации, различают алгоритмы со **строгим полиномиальным временем** и **слабо полиномиальным временем**. Эти две концепции относятся только ко входным данным, состоящим из целых чисел.

Строго полиномиальное время определяется в арифметической модели вычислений. В этой модели базовые арифметические операции (сложение, вычитание, умножение, деление и сравнение) берутся за единицы выполнения, независимо от длины операндов. Алгоритм работает в строго полиномиальное время, если[11]

1. число операций в арифметической модели вычислений ограничено многочленом от числа целых во входном потоке, и
2. память, используемая алгоритмом, ограничена многочленом от размеров входа.

Любой алгоритм с этими двумя свойствами можно привести к алгоритму полиномиального времени путём замены арифметических операций на соответствующие алгоритмы выполнения арифметических операций на машине Тьюринга. Если второе из вышеприведённых требований не выполняется, это больше не будет верно. Если задано целое число 2^n (которое занимает память, пропорциональную n в машине Тьюринга), можно вычислить 2^{2^n} с помощью n операций, используя повторное возведение в степень. Однако память, используемая для представления 2^{2^n}, пропорциональна 2^n, и она скорее экспоненциально, чем полиномиально, зависит от памяти, используемой для входа. Отсюда — невозможно выполнить эти вычисления за полиномиальное время на машине Тьюринга, но можно выполнить за полиномиальное число арифметических операций.

Обратно — существуют алгоритмы, которые работают за число шагов машины Тьюринга, ограниченных полиномиальной длиной бинарно закодированного входа, но не работают за число арифметических операций, ограниченное многочленом от количества чисел на входе. Алгоритм Евклида для вычисления наибольшего общего делителя двух целых чисел является одним из примеров. Для двух целых чисел a и b время работы алгоритма ограничено $O((\log a + \log b)^2)$ шагам машины Тьюринга. Это число является многочленом от размера бинарного представления чисел a и b, что грубо можно представить как $\log a + \log b$. В то же самое время число арифметических операций нельзя ограничить числом целых во входе (что в данном случае является константой — имеется только два числа во входе). Ввиду этого замечания алгоритм не работает в строго полиномиальное время. Реальное время работы алгоритма зависит от величин a и b, а не только от числа целых чисел во входе.

Если алгоритм работает за полиномиальное время, но не за строго полиномиальное время, говорят, что он работает за **слабо полиномиальное время**[12]. Хорошо известным примером задачи, для которой известен слабо полиномиальный алгоритм, но не известен строго полиномиальный алгоритм, является линейное программирование. Слабо полиномиальное время не следует путать с псевдополиномиальным временем.

Классы сложности

Концепция полиномиального времени приводит к нескольким классам сложности в теории сложности вычислений. Некоторые важные классы, определяемые с помощью полиномиального времени, приведены ниже.

- **P**: Класс сложности задач разрешимости, которые могут быть решены в детерминированной машине Тьюринга за полиномиальное время.
- **NP**: Класс сложности задач разрешимости, которые могут быть решены в недетерминированной машине Тьюринга за полиномиальное время.
- **ZPP**: Класс сложности задач разрешимости, которые могут быть решены с нулевой ошибкой в вероятностной машине Тьюринга за полиномиальное время.
- **RP**: Класс сложности задач разрешимости, которые могут быть решены с односторонними ошибками в вероятностной машине Тьюринга за полиномиальное время.
- **BPP**: Класс сложности задач разрешимости, которые могут быть решены с двусторонними ошибками в вероятностной машине Тьюринга за полиномиальное время.
- **BQP**: Класс сложности задач разрешимости, которые могут быть решены с двусторонними ошибками в квантовой машине Тьюринга за полиномиальное время.

P является наименьшим классом временной сложности на детерминированной машине, которая является устойчивой в терминах изменения модели машины. (Например, переход от одноленточной машины Тьюринга к мультиленточной может привести к квадратичному ускорению, но любой алгоритм, работающий за полиномиальное время на одной модели, будет работать за полиномиальное время на другой.)

Суперполиномиальное время

Говорят, что алгоритм работает за **суперполиномиальное время**, если $T(n)$ не ограничен сверху полиномом. Это время равно $\omega(n^c)$ для всех констант c, где n — входной параметр, обычно — число бит входа.

Например, алгоритм, осуществляющий 2^n шагов, для входа размера n требует суперполиномиального времени (конкретнее, экспоненциального времени).

Ясно, что алгоритм, использующий экспоненциальные ресурсы, суперполиномиален, но некоторые алгоритмы очень слабо суперполиномиальны. Например, тест простоты Адлемана — Померанса — Румели (https://ru.wikipedia.org/w/index.php?title=%D0%A2%D0%B5%D1%81%D1%82_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE%D1%82%D1%8B_%D0%90%D0%B4%D0%BB%D0%B5%D0%BC%D0%B0%D0%BD%D0%B0_%E2%80%94_%D0%9F%D0%BE%D0%BC%D0%B5%D1%80%D0%B0%D0%BD%D1%81%D0%B0_%E2%80%94_%D0%A0%D1%83%D0%BC%D0%B5%D0%BB%D0%B8&redirect=no) работает за время $n^{O(\log \log n)}$ на n-битном входе. Это растёт быстрее, чем любой полином, для достаточно большого n, но размер входа должен стать очень большим, чтобы он не доминировался полиномом малой степени.

Алгоритм, требующий суперполиномиального времени, лежит вне класса сложности **P**. Тезис Кобэма утверждает, что эти алгоритмы непрактичны, и во многих случаях это так. Поскольку задача равенства классов P и NP не решена, никаких алгоритмов для решения NP-полных задач за полиномиальное время в настоящее время не известно.

Квазиполиномиальное время

Алгоритмы **квазиполиномиального времени** — это алгоритмы, работающие медленнее, чем за полиномиальное время, но не столь медленно, как алгоритмы экспоненциального времени. Время работы в худшем случае для квазиполиномиального алгоритма равно $2^{O((\log n)^c)}$ для некоторого фиксированного c. Хорошо известный классический алгоритм разложения целого числа на множители, общий метод решета числового поля, который работает за время около $2^{\tilde{O}(n^{1/3})}$, *не* является квазиполиномиальным, поскольку время работы нельзя представить как $2^{O((\log n)^c)}$ для некоторого фиксированного c. Если константа "c" в определении алгоритма квазиполиномиального времени равна 1, мы получаем алгоритм полиномиального времени, а если она меньше 1, мы получаем алгоритм сублинейного времени.

Алгоритмы квазиполиномиального времени обычно возникают при сведении NP-трудной задачи к другой задаче. Например, можно взять NP-трудную задачу, скажем, 3SAT, и свести её к другой задаче B, но размер задачи станет равным $2^{O((\log n)^c)}$. В этом случае сведение не доказывает, что задача B NP-трудна, такое сведение лишь показывает, что не существует полиномиального алгоритма для B, если только не существует квазиполиномиального алгоритма для 3SAT (а тогда и для всех NP-задач). Подобным образом — существуют некоторые задачи, для которых мы знаем алгоритмы с квазиполиномиальным временем, но для которых алгоритмы с полиномиальным временем неизвестны. Такие задачи появляются в аппроксимационых алгоритмах. Знаменитый пример — ориентированная задача Штайнера, для которой существует аппроксимационный квазиполиномиальный алгоритм с аппроксимационным коэффициентом $O(\log^3 n)$ (где n — число вершин), но существование алгоритма с полиномиальным временем является открытой проблемой.

Класс сложности **QP** состоит из всех задач, имеющих алгоритмы квазиполиномиального времени. Его можно определить в терминах DTIME следующим образом[13]

$$\mathrm{QP} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}(2^{(\log n)^c})$$

Связь с NP-полными задачами

В теории сложности нерешённая проблема равенства классов P и NP спрашивает, не имеют ли все задачи из класса NP алгоритмы решения за полиномиальное время. Все хорошо известные алгоритмы для NP-полных задач, наподобие 3SAT, имеют экспоненциальное время. Более того, существует гипотеза, что для многих естественных NP-полных задач не существует алгоритмов с субэкспоненциальным временем выполнения. Здесь "субэкспоненциальное время " взято в смысле второго определения, приведённого ниже. (С другой стороны, многие задачи из теории графов, представленные естественным путём матрицами смежности, разрешимы за субэкспоненциальное время просто потому, что размер входа равен квадрату числа вершин.) Эта гипотеза (для задачи k-SAT) известна как гипотеза экспоненциального времени[14]. Поскольку она предполагает, что NP-полные задачи не имеют алгоритмов квазиполиномиального времени, некоторые результаты неаппроксимируемости в области аппроксимационных алгоритмов исходят из того, что NP-полные задачи не имеют алгоритмов квазиполиномиального времени. Например, смотрите известные результаты по неаппроксимируемости задачи о покрытии множества.

Субэкспоненциальное время

Термин **субэкспоненциальное время** используется, чтобы выразить, что время выполнения некоторого алгоритма может расти быстрее любого полиномиального, но остаётся существенно меньше, чем экспоненциальное. В этом смысле задачи, имеющие алгоритмы субэкспоненциального времени, являются более податливыми, чем алгоритмы только с экспотенциальным временем. Точное определение "субэкспоненциальный" пока не является общепринятым[15], и мы приводим ниже два наиболее распространённых определения.

Первое определение

Говорят, что задача решается за субэкспоненциальное время, если она решается алгоритмом, логарифм времени работы которого растёт меньше, чем любой заданный многочлен. Более точно — задача имеет субэкспоненциальное время, если для любого $\varepsilon > 0$ существует алгоритм, который решает задачу за время $O(2^{n^\varepsilon})$. Множество все таких задач составляет класс сложности **SUBEXP**, который в терминах DTIME можно выразить как[3][16][17][18].

$$\mathrm{SUBEXP} = \bigcap_{\varepsilon > 0} \mathrm{DTIME}\left(2^{n^\varepsilon}\right)$$

Заметим, что здесь ε не является частью входных данных и для каждого ε может существовать свой собственный алгоритм решения задачи.

Второе определение

Некоторые авторы определяют субэкспоненциальное время как время работы $2^{o(n)}$[14][19][20]. Это определение допускает большее время работы, чем первое определение. Примером такого алгоритма субэкспоненциального времени служит хорошо известный классический алгоритм разложения целых чисел на множители, общий метод решета числового поля, который работает за время около $2^{\tilde{O}(n^{1/3})}$, где длина входа равна n. Другим примером служит хорошо известный алгоритм для задачи изоморфизма графов, время работы которого равно $2^{O(\sqrt{(n \log n)})}$.

Заметим, что есть разница, является ли алгоритм субэкспоненциальным по числу вершин или числу рёбер. В параметризованной сложности эта разница присутствует явно путём указания пары (L, k), задачи разрешимости и параметра k. **SUBEPT** является классом всех параметризованных задач, которые работают за субэкспоненциальное время по k и за полиномиальное по n[21]:

$$\mathbf{SUBEPT} = \mathbf{DTIME}\left(2^{o(k)} \cdot \mathrm{poly}(n)\right).$$

Точнее, SUBEPT является классом всех параметризованных задач (L, k), для которых существует вычислимая функция $f : \mathbb{N} \to \mathbb{N}$ с $f \in o(k)$ и алгоритм, который решает L за время $2^{f(k)} \cdot \mathrm{poly}(n)$.

Гипотеза об экспоненциальном времени

Гипотеза об экспоненциальном времени ('**ETH**) утверждает, что 3SAT, задача выполнимости булевых формул в конъюнктивной нормальной форме с максимум тремя литералами на предложение и n переменными, не может быть решена за время $2^{o(n)}$. Точнее, гипотеза говорит, что существует некая константа c>0, такая, что 3SAT не может быть решена за время 2^{cn} на любой детерминированной машине Тьюринга. Если через m обозначить число предложений, ETH эквивалентна гипотезе, что k-SAT не может быть решена за время $2^{o(m)}$ для любого целого $k \geq 3$[22]. Из гипотезы об экспоненциальном времени следует, что P ≠ NP.

Экспоненциальное время

Говорят, что алгоритм работает за **экспоненциальное время**, если $T(n)$ ограничено сверху значением $2^{\mathrm{poly}(n)}$, где poly(n) — некий многочлен от n. Более формально, алгоритм работает за экспоненциальное время, если $T(n)$ ограничено $O(2^{n^k})$ с некоторой константой k. Задачи, которые выполняются за экспоненциальное время на детерминированных машинах Тьюринга, образуют класс сложности **EXP**.

$$\mathbf{EXP} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}\left(2^{n^c}\right)$$

Иногда термин экспоненциальное время используется для алгоритмов, для которых $T(n) = 2^{O(n)}$, где показатель является не более чем линейной функцией от n. Это приводит к классу сложности **E**.

$$\mathbf{E} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}\left(2^{cn}\right)$$

Двойное экспоненциальное время

Говорят, что алгоритм выполняется за дважды экспоненциальное время, если $T(n)$ ограничено сверху значением $2^{2^{\mathrm{poly}(n)}}$, где poly(n) — некоторый многочлен от n. Такие алгоритмы принадлежат классу сложности 2-EXPTIME.

$$\mathbf{2\text{-}EXPTIME} = \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}(2^{2^{n^c}})$$

К хорошо известным дважды экспоненциальным алгоритмам принадлежат:

- Процедура вычисления для арифметики Пресбургера
- Вычисление базиса Грёбнера (в худшем случае[23])
- Элиминация кванторов в вещественно замкнутых полях требует как минимум дважды экспоненциальное время выполнения [24] и может быть выполнена за это время[25].

См. также

- L-нотация
- Сложность по памяти

Примечания

1. *Michael Sipser.* Introduction to the Theory of Computation. — Course Technology Inc, 2006. — ISBN 0-619-21764-2.
2. *Kurt Mehlhorn, Stefan Naher.* Bounded ordered dictionaries in O(log log N) time and O(n) space // Information Processing Letters. — 1990. — Т. 35, вып. 4. — С. 183. — DOI:10.1016/0020-0190(90)90022-P (https://dx.doi.org/10.1016%2F0020-0190%2890%2990022-P).
3. *László Babai, Lance Fortnow, N. Nisan, Avi Wigderson.* BPP has subexponential time simulations unless EXPTIME has publishable proofs // Computational Complexity. — Berlin, New York: Springer-Verlag, 1993. — Т. 3, вып. 4. — С. 307–318. — DOI:10.1007/BF01275486 (https://dx.doi.org/10.1007%2FBF01275486).
4. *J. E. Rawlins, Gregory E. Shannon.* Efficient Matrix Chain Ordering in Polylog Time // SIAM Journal on Computing. — Philadelphia: Society for Industrial and Applied Mathematics, 1998. — Т. 27, вып. 2. — С. 466–490. — ISSN 1095-7111 (https://www.worldcat.org/search?fq=x0:jrnl&q=n2:1095-7111). — DOI:10.1137/S0097539794270698 (https://dx.doi.org/10.1137%2FS0097539794270698).
5. *Ravi Kumar, Ronitt Rubinfeld.* Sublinear time algorithms // SIGACT News. — 2003. — Т. 34, вып. 4. — С. 57–67. — DOI:10.1145/954092.954103 (https://dx.doi.org/10.1145%2F954092.954103).
6. *DR K N PRASANNA KUMAR, PROF B S KIRANAGI AND PROF C S BAGEWADI.* A GENERAL THEORY OF THE SYSTEM ‘QUANTUM INFORMATION - QUANTUM ENTANGLEMENT, SUBATOMIC PARTICLE DECAY – ASYMMETRIC SPIN STATES, NON LOCALLY HIDDEN VARIABLES – A CONCATENATED MODEL // International Journal of Scientific and Research Publications. — July 2012. — Т. 2, вып. 7. — ISSN 22503153 (https://www.worldcat.org/search?fq=x0:jrnl&q=n2:22503153).
7. *Ashish V. Naik, Kenneth W. Regan, D. Sivakumar.* On Quasilinear Time Complexity Theory // Theoretical Computer Science. — Т. 148. — С. 325–349.
8. Sedgewick, R. and Wayne K (2011). Algorithms, 4th Ed (http://algs4.cs.princeton.edu/home/). p. 186. Pearson Education, Inc.
9. *Christos H. Papadimitriou.* Computational complexity. — Reading, Mass.: Addison-Wesley, 1994. — ISBN 0-201-53082-1.
10. *Alan Cobham.* Proc. Logic, Methodology, and Philosophy of Science II. — North Holland, 1965. — С. The intrinsic computational difficulty of functions.
11. *Martin Grötschel, László Lovász, Alexander Schrijver.* Geometric Algorithms and Combinatorial Optimization. — Springer, 1988. — С. Complexity, Oracles, and Numerical Computation. — ISBN 0-387-13624-X.
12. *Alexander Schrijver.* Combinatorial Optimization: Polyhedra and Efficiency. — Springer, 2003. — Т. 1. — С. Preliminaries on algorithms and Complexity. — ISBN 3-540-44389-4.
13. *Complexity Zoo* (http://qwiki.stanford.edu/wiki/Complexity_Zoo) Архивная копия (http://web.archive.org/web/20100726082118/http://qwiki.stanford.edu/wiki/Complexity_Zoo) от 26 июля 2010 на Wayback Machine Class QP: Quasipolynomial-Time (https://complexityzoo.uwaterloo.ca/Complexity_Zoo:Q#qp)
14. *R. Impagliazzo, R. Paturi.* On the complexity of k-SAT // Journal of Computer and System Sciences. — Elsevier, 2001. — Т. 62, вып. 2. — С. 367–375. — ISSN 1090-2724 (https://www.worldcat.org/search?fq=x0:jrnl&q=n2:1090-2724). — DOI:10.1006/jcss.2000.1727 (https://dx.doi.org/10.1006%2Fjcss.2000.1727).
15. *Aaronson, Scott.* A not-quite-exponential dilemma. — 5 April 2009.
16. *Complexity Zoo* (http://qwiki.stanford.edu/wiki/Complexity_Zoo) Архивная копия (http://web.archive.org/web/20100726082118/http://qwiki.stanford.edu/wiki/Complexity_Zoo) от 26 июля 2010 на Wayback Machine Class SUBEXP: Deterministic Subexponential-Time (https://complexityzoo.uwaterloo.ca/Complexity_Zoo:S#subexp)
17. *P. Moser.* Baire's Categories on Small Complexity Classes // Lecture Notes in Computer Science. — Berlin, New York: Springer-Verlag, 2003. — С. 333–342. — ISSN 0302-9743 (https://www.worldcat.org/search?fq=x0:jrnl&q=n2:0302-9743).
18. *P.B. Miltersen.* DERANDOMIZING COMPLEXITY CLASSES // Handbook of Randomized Computing. — Kluwer Academic Pub, 2001. — С. 843.

19. *Greg Kuperberg.* A Subexponential-Time Quantum Algorithm for the Dihedral Hidden Subgroup Problem // SIAM Journal on Computing. — Philadelphia: Society for Industrial and Applied Mathematics, 2005. — Т. 35, вып. 1. — С. 188. — ISSN 1095-7111 (https://www.worldcat.org/search?fq=x0:jrnl&q=n2:1095-7111). — DOI:10.1137/s0097539703436345 (https://dx.doi.org/10.1137%2Fs0097539703436345).
20. *Oded Regev.* A Subexponential Time Algorithm for the Dihedral Hidden Subgroup Problem with Polynomial Space. — 2004.
21. *Jörg Flum, Martin Grohe.* Parameterized Complexity Theory. — Springer, 2006. — С. 417. — ISBN 978-3-540-29952-3.
22. *R. Impagliazzo, R. Paturi, F. Zane.* Which problems have strongly exponential complexity? // Journal of Computer and System Sciences. — 2001. — Т. 63, вып. 4. — С. 512–530. — DOI:10.1006/jcss.2001.1774 (https://dx.doi.org/10.1006%2Fjcss.2001.1774).
23. *Mayr,E. & Mayer,A.* The Complexity of the Word Problem for Commutative Semi-groups and Polynomial Ideals // Adv. in Math.. — 1982. — Вып. 46. — С. 305-329.
24. *J.H. Davenport, J. Heintz.* Real Quantifier Elimination is Doubly Exponential // J. Symbolic Comp.. — 1988. — Вып. 5. — С. 29-35..
25. *G.E. Collins.* Proc. 2nd. GI Conference Automata Theory & Formal Languages. — Springer. — Т. 33. — С. 134-183. — (Lecture Notes in Computer Science).
