Алгоритм Евклида для нахождения наибольшего общего делителя

Целое число a делит без остатка другое целое число b, если, и только если

$b = k * a$

для некоторого целого числа k.

В этом случае a называют *делителем числа* b или *множителем в разложении* числа b на множители.

Пусть a – целое число, большее 1. Тогда

a является *простым числом*,

если его единственными положительными делителями будут 1 и само a,

в противном случае a называется *составным* .

Любое целое n >1 может быть представлено единственным образом с точностью до порядка сомножителей как произведение простых .

Существенный с точки зрения криптографии факт состоит в том, что не известно никакого эффективного алгоритма разложения чисел на множители;
не было получено и никакой нетривиальной нижней оценки временной сложности разложения.
Никаких эффективных методов не известно даже в таком простом случае, когда необходимо восстановить два простых числа p и q из их произведения:
$n = p * q$.

Наибольший общий делитель чисел a и b, обозначаемый как НОД (a,b) или просто (a,b), – это наибольшее целое, делящее одновременно числа a и b.

Если НОД (a,b)=1, то целые a и b – *взаимно простые*.

Наибольший общий делитель может быть вычислен с помощью *алгоритма Евклида*. Евклид описал этот алгоритм в своей книге "Начала", написанной около 300 лет до н.э. Он не изобрел его. Историки полагают, что этот алгоритм, возможно, старше еще на 200 лет. Это древнейший нетривиальный алгоритм, который просуществовал до настоящего времени и все еще хорош и сегодня.

Опишем алгоритм Евклида для нахождения НОД (a,b).

Введем обозначения:

q_i – частное; r_i – остаток.

Тогда алгоритм можно представить в виде следующей цепочки равенств:

$$(a,b)=(b,r)=(r,r_1) = \ldots=(r_{n-1},r_n)=r_n$$

Остановка гарантируется, поскольку остатки r_i от делений образуют строго убывающую последовательность натуральных чисел.

Таким образом, r_n = НОД (a, b) или r_n = (a, b).

Как следствие из алгоритма Евклида, можно получить утверждение, что
наибольший делитель целых чисел *a* и *b*
может быть представлен в виде линейной комбинации этих чисел, т.е. существуют целые числа *u* и *v* такие, что справедливо равенство

$$a \cdot u + b \cdot v = r_n$$

Алгоритм Евклида для вычисления наибольшего общего делителя

```
begin
    g[0]: = b;
    g[1]: = a;
      i : =1;
while  g[i] ! = 0   do
      begin
g[i+1] : = g[i–1] mod(g[i]);
        i : = i +1;
      end
gcd : = g[i–1];      /*gcd – результат*/
end
```

Вычисление обратного элемента по заданному модулю

Если целые числа *a* и *n* взаимно просты, то существует число a', удовлетворяющее сравнению $a \cdot a' = 1 \pmod{n}$.
Число a' называют *обратным* к *a* по модулю *n* и используют обозначение : $a^{-1} \pmod{n}$.

Вычислить *a'* можно, например, воспользовавшись представлением наибольшего общего делителя чисел *a* и *n в* виде их линейной комбинации: $a*u + n*v = 1$.

Взяв наименьшие неотрицательные вычеты обеих частей этого равенства по модулю *n,* получим, что искомое значение *a'* удовлетворяет сравнению

$$a' = u \pmod{n}$$

Расширенный алгоритм Евклида

При заданных неотрицательных целых числах a и b этот алгоритм определяет вектор
(u1, u2, u3),
такой, что
a * u1 + b * u2 = u3 = НОД (a, b).

В процессе вычисления используются вспомогательные векторы (v1, v2, v3), (t1, t2, t3). Действия с векторами производятся таким образом, что в течение всего процесса вычисления выполняются соотношения

a * t1 + b * t2 = t3, a * u1 + b * u2 = u3,

a * v1 + b * v2 = v3.

Шаги алгоритма:

1. Начальная установка.

Установить $(u_1, u_2, u_3) := (0, 1, n)$,

$(v_1, v_2, v_3) := (1, 0, a)$.

Пример. Заданы модуль n = 23 и число a = 5. Найти обратное число $a^{-1} \pmod{23}$, т.е. $x = 5^{-1} \pmod{23}$.

Используя расширенный алгоритм Евклида, выполним вычисления, записывая результаты отдельных шагов

q	u_1	u_2	u_3	v_1	v_2	v_3
–	0	1	n = 23	1	0	a = 5
–						

$q := [u_3/v_3]$.

$(u_1, u_2, u_3) := (v_1, v_2, v_3)$,

$(t_1, t_2, t_3) := (u_1, u_2, u_3) – (v_1, v_2, v_3) * q$,

$(v_1, v_2, v_3) := (t_1, t_2, t_3)$.

q	u_1	u_2	u_3	v_1	v_2	v_3
–	0	1	n = 23	1	0	a = 5
4	1	0	5	–4	1	3
1	–4	1	3	5	–1	2
1	5	–1	2	–9	2	1
–	–9	2	1			

При $u_3 = 1$, $u_1 = -9$, $u_2 = 2$

$$(a * u_1 + n * u_2) \bmod n = (5 * (-9) + 23 * 2) \bmod 23 =$$

$$= 5 * (-9) \bmod 23 \equiv 1,$$

$$a^{-1} \pmod n = 5^{-1} \pmod{23} = (-9) \bmod 23 = (-9 + 23) \bmod 23 = 14.$$

Итак, $x = 5^{-1} \pmod{23} \equiv 14 \pmod{23} = 14$.

Малая теорема Ферма

Если m – простое число, и a не кратно m ,то малая теорема Ферма утверждает:

$$a^{m-1} \equiv 1 \left(\mathrm{mod}\, m \right)$$

Функция Эйлера

Приведенной системой вычетов по модулю n называют подмножество полной системы вычетов, члены которого взаимно просты с n.

Например, приведенную систему вычетов по модулю 12 составляют числа {1, 5, 7, 11}.

Если n - простое число, в приведенную систему вычетов по модулю n входит всё множество чисел от 1 до n— 1.

Для любого n, не равного 1, число 0 никогда не входит в приведенную систему вычетов.

Функция Эйлера, которую иногда называют функцией «фи» Эйлера и записывают как $\varphi(n)$, указывает число элементов в приведенной системе вычетов по модулю n.

Иными словами, $\varphi(n)$ - это количество положительных целых чисел, меньших n и взаимно простых с n (для любого n, большего 1).

Если n — простое число, то $\varphi(n) = n-1$.

Если $n = pq$, где p и q - простые числа, то $\varphi(n) = (p-1)(q-1)$.

В соответствии с **обобщением Эйлера малой теоремы Ферма,** если *НОД(a,n)* = 1,то:

$$a^{\varphi(n)} \bmod n = 1$$

Теперь нетрудно вычислить $a^{-1} \bmod n$:

$$a^{-1} = a^{\varphi(n)-1} \bmod n$$

Основные способы нахождения обратных величин

$a^{-1} \pmod n$.

1. Проверить поочередно значения 1, 2, ..., n – 1, пока не будет найден $a^{-1} \pmod n$, такой, что $a*a^{-1} \pmod n \equiv 1$.

n=Prime[number];

a= Mod[b, n]

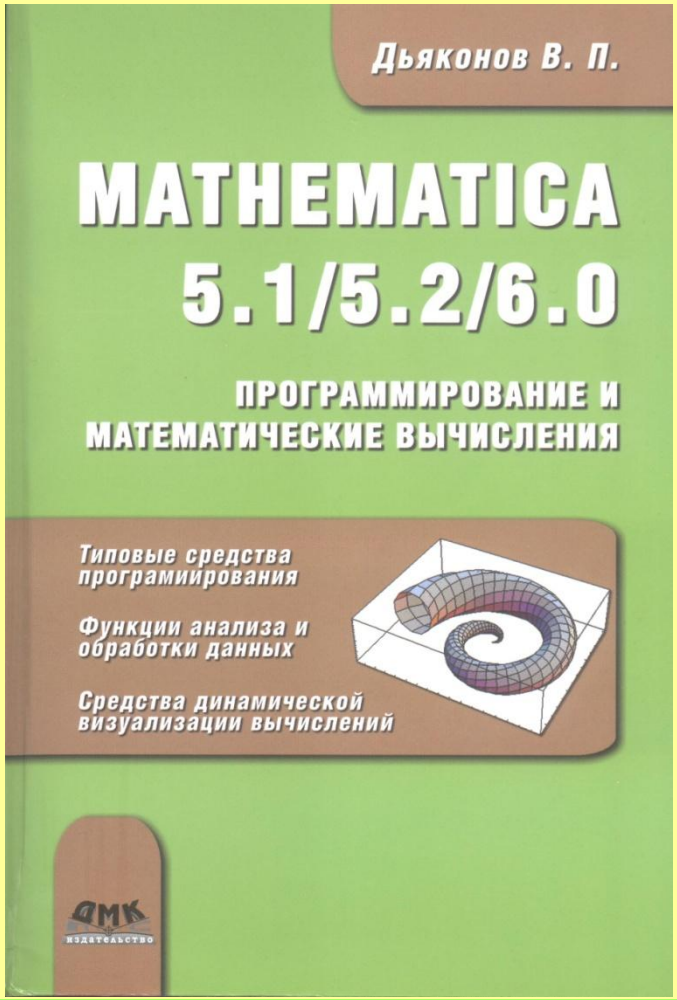
Do[,]

While[,]

For[,]

If[,]

Break[]



2. Если известна функция Эйлера $\phi(n)$, то можно вычислить

$$a^{-1} \pmod n \equiv a^{\phi(n)-1} \pmod n,$$

используя алгоритм быстрого возведения в степень.

EulerPhi[n];

PowerMod[a,b,n] = $a^b \bmod n$.

3. Если функция Эйлера $\phi(n)$ не известна, можно использовать расширенный алгоритм Евклида.

n=23 a=5

ExtendedGCD[n,a]

{1,{2,-9}}

$u_3 = 1,\ u_2 = 2,\ u_1 = -9$

$5^{-1} \pmod{23} = (-9) \bmod 23 = (-9 + 23) \bmod 23 = 14.$

Mod[-9,23]

14

Для решения более сложных сравнений

$$a * x \equiv b \pmod{n}, \text{ т.е. } b \neq 1, \ x = ?$$

используется следующий прием. Сначала решают сравнение

$$a * y \equiv 1 \pmod{n},$$

т.е. определяют

$$y = a^{-1} \pmod{n},$$

а затем находят

$$x = a^{-1} b \pmod{n} = y * b \pmod{n}.$$

Теория сложности

Теория сложности обеспечивает методологию анализа **вычислительной сложности** различных криптографических методов и алгоритмов.

Она сравнивает криптографические методы и алгоритмы и определяет их безопасность.

Теория информации сообщает нам о том, что все криптографические алгоритмы (кроме одноразовых блокнотов) могут быть взломаны.

Теория сложности сообщает, могут ли они быть взломаны до тепловой смерти вселенной.

Большие числа

Физический эквивалент	Число

Большие числа

Большие числа

Большие числа

Большие числа

Физический эквивалент	Число

Проблемы начнутся, когда мы учтём квантовые явления. Главный результат применения квантовой теории к чёрной дыре состоит в том, что она постепенно испаряется благодаря излучению ХокингаПроблемы начнутся, когда мы учтём квантовые явления. Главный результат применения квантовой теории к чёрной дыре состоит в том, что она постепенно испаряется благодаря излучению Хокинга. Это значит, что можно дождаться такого момента, когда масса чёрной дыры снова уменьшится до первоначального значения (перед бросанием в неё тела). Таким образом, мы получаем в результате, что чёрная дыра превратила исходное тело в поток разнообразных излучений, но сама при этом не изменилась (поскольку она вернулась к исходной массе!). Испущенное излучение при этом совершенно не зависит от того, что за тело было брошено в чёрную дыру. То есть чёрная дыра уничтожила попавшую в неё информацию.

В рамках классической (неквантовой) теории гравитации чёрная дыра — объект неуничтожимый. Она может только расти, но не может ни уменьшиться, ни исчезнуть совсем. Это значит, что в принципе возможна ситуация, что попавшая в чёрную дыру информация на самом деле не исчезла, она продолжает находиться внутри чёрной дыры, но просто не наблюдаема снаружи. Иная разновидность этой же мысли: если чёрная дыра служит мостом между нашей вселенной и какой-нибудь другой Вселенной, то информация, возможно, просто перебросилась в другую вселенную.

Рисунок художника: аккреционный диск горячей постоянной *плазмы*, вращающийся вокруг чёрной дыры

Сложность алгоритмов

Сложность алгоритма определяется вычислительными мощностями, необходимыми для его выполнения.

Вычислительная сложность алгоритма часто измеряется двумя параметрами:

T **- временная сложность ;**

S **- пространственная сложность,** или требования к памяти.

T и ***S*** представляются в виде функций от ***n***, где ***n*** - это размер входных данных.

(Существуют и другие способы измерения сложности: количество случайных бит, ширина канала связи, объем данных и т.п.)

Вычислительная сложность алгоритма выражается с помощью нотации "***O*** большого", т.е описывается порядком величины вычислительной сложности.

Это просто член разложения функции сложности, быстрее всего растущий с ростом ***n***, все члены низшего порядка игнорируются.

Например, если временная сложность данного алгоритма равна $4n^2+7n+12$, то вычислительная сложность порядка n^2, записываемая как $\boldsymbol{O}(n^2)$.

Временная сложность измеренная таким образом не зависит от реализации. Не нужно знать ни точное время выполнения различных инструкций, ни число битов, используемых для представления различных переменных, ни даже скорость процессора.

При работе с сложными алгоритмами , всем прочим можно пренебречь (с точностью до постоянного множителя) в сравнении со сложностью по порядку величины.

Алгоритмы классифицируются в соответствии с их временной или пространственной сложностью.

Алгоритм называют **постоянным,** если его сложность не зависит от *n:* ***O***(1).

Алгоритм является **линейным,** если его временная сложность ***O*** (n).

Алгоритмы могут быть **квадратичными, кубическими** и т.д.

Все эти алгоритмы - **полиномиальны,** их сложность - ***O***(n^m), где *m* - константа.

Алгоритмы с полиномиальной временной сложностью называются алгоритмами **с полиномиальным временем.**

Алгоритмы, сложность которых равна $O(t^{f(n)})$,
где t - константа, большая, чем 1,
а $f(n)$ - некоторая полиномиальная функция от n,
называются **экспоненциальными.**

Подмножество экспоненциальных алгоритмов, сложность которых равна $O(c^{f(n)})$,
где где c - константа,
а $f(n)$ возрастает быстрее, чем постоянная, но медленнее, чем линейная функция,
называется **суперполиномиальным**

В идеале, криптограф хотел бы утверждать, что алгоритм, лучший для взлома спроектированного алгоритма шифрования, обладает **экспоненциальной** временной сложностью.

На практике, самые сильные утверждения, которые могут быть сделаны при текущем состоянии теории вычислительной сложности, имеют форму:
"все известные алгоритмы вскрытия данной криптосистемы обладают суперполиномиальной временной сложностью".

То есть, известные алгоритмы вскрытия обладают суперполиномиальной временной сложностью,
но пока невозможно доказать, что не может быть открыт алгоритм вскрытия с полиномиальной временной сложностью.

С ростом ***n*** временная сложность алгоритмов может стать настолько огромной, что это повлияет на практическую реализуемость алгоритма.

Время выполнения для различных классов алгоритмов

Класс	Сложность	Количество операций для $n=10^6$	Время при 10^6 операций в секунду
Постоянные	$O(1)$	1	1 мкс
Линейные	$O(n)$	10^6	1 с
Квадратичные	$O(n^2)$	10^{12}	11.6 дня
Кубические	$O(n^3)$	10^{18}	32000 лет
Экспоненциальные	$O(2^n)$	10^{301030}	В 10^{301006} раз больше, чем время существования вселенной

При условии, что единицей времени для нашего компьютера является микросекунда, компьютер может выполнить постоянный алгоритм за микросекунду, линейный - за секунду, а квадратичный - за 11.6 дня.

Выполнение кубического алгоритма потребует 32 тысяч лет, что в принципе реализуемо, компьютер, конструкция которого позволила бы ему противостоять следующему ледниковому периоду, в конце концов получил бы решение.

Взглянем на проблему вскрытия алгоритма шифрования грубой силой.

Временная сложность такого вскрытия пропорциональна количеству возможных ключей, которое экспоненциально зависит от длины ключа.

Если n - длина ключа, то сложность вскрытия грубой силой равна $\boldsymbol{O}(2^n)$.

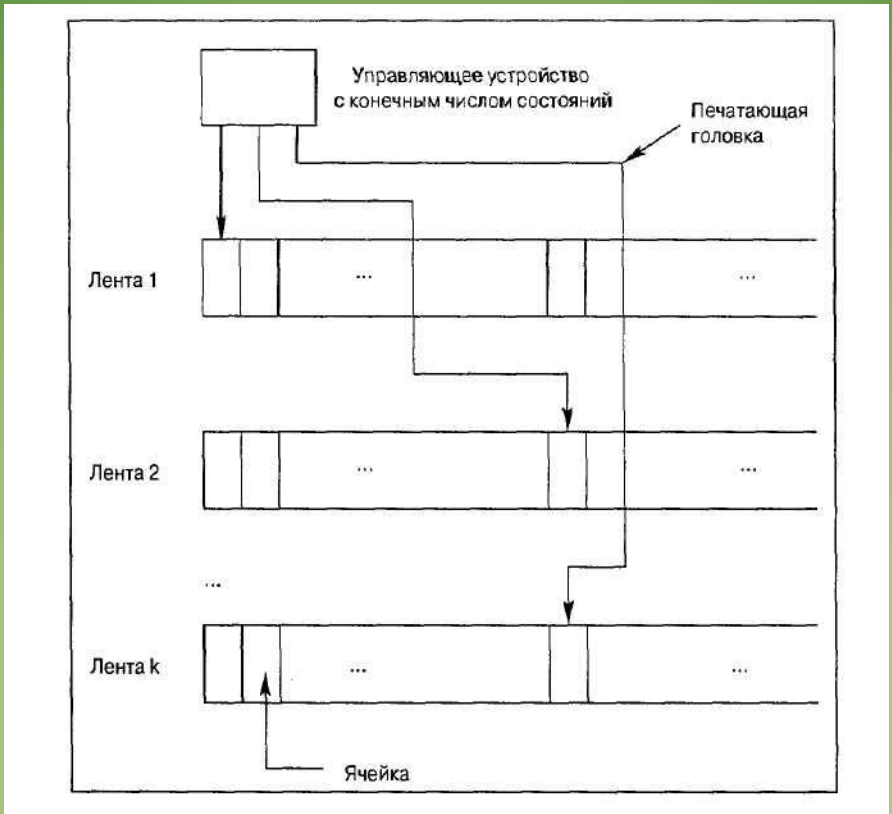
Сложность вскрытия грубой силой алгоритма DES при 56-битовом ключе составляет 2^{56}, а при 112-битовом ключе - 2^{112}.
В первом случае вскрытие возможно, а во втором - нет.

Сложность проблем

Теория сложности также классифицирует и сложность самих проблем, а не только сложность конкретных алгоритмов решения проблемы.

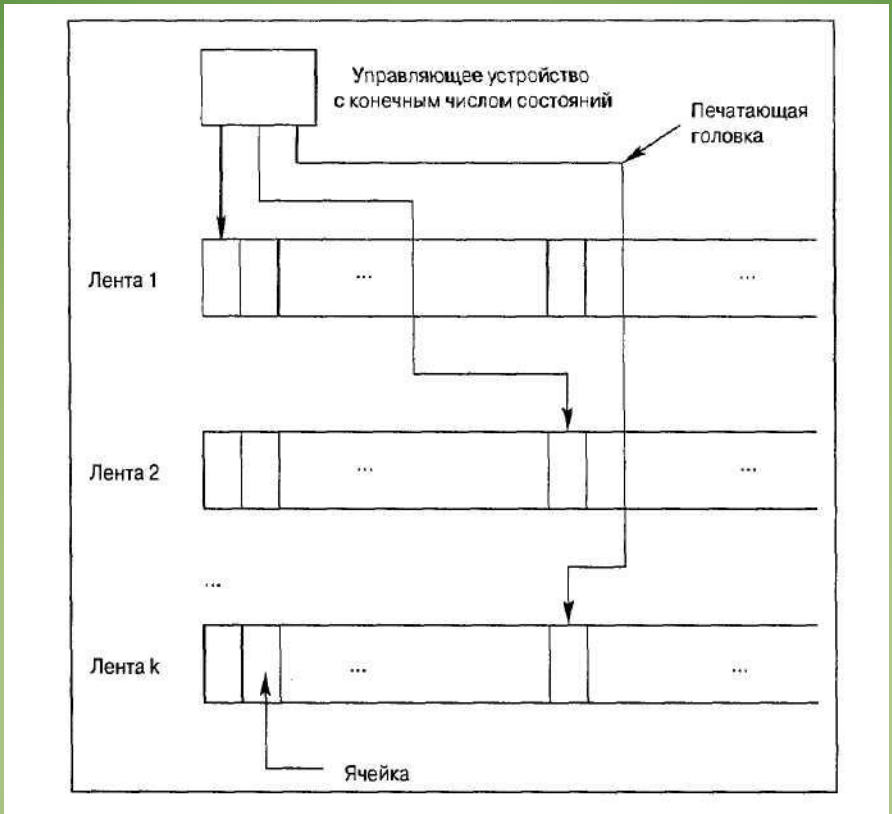
Теория рассматривает минимальное время и объем памяти, необходимые для решения самого трудного варианта проблемы на теоретическом компьютере, известном как **машина Тьюринга.**

Машина Тьюринга представляет собой конечный автомат с бесконечной лентой памяти для чтения-записи и является реалистичной моделью вычислений.



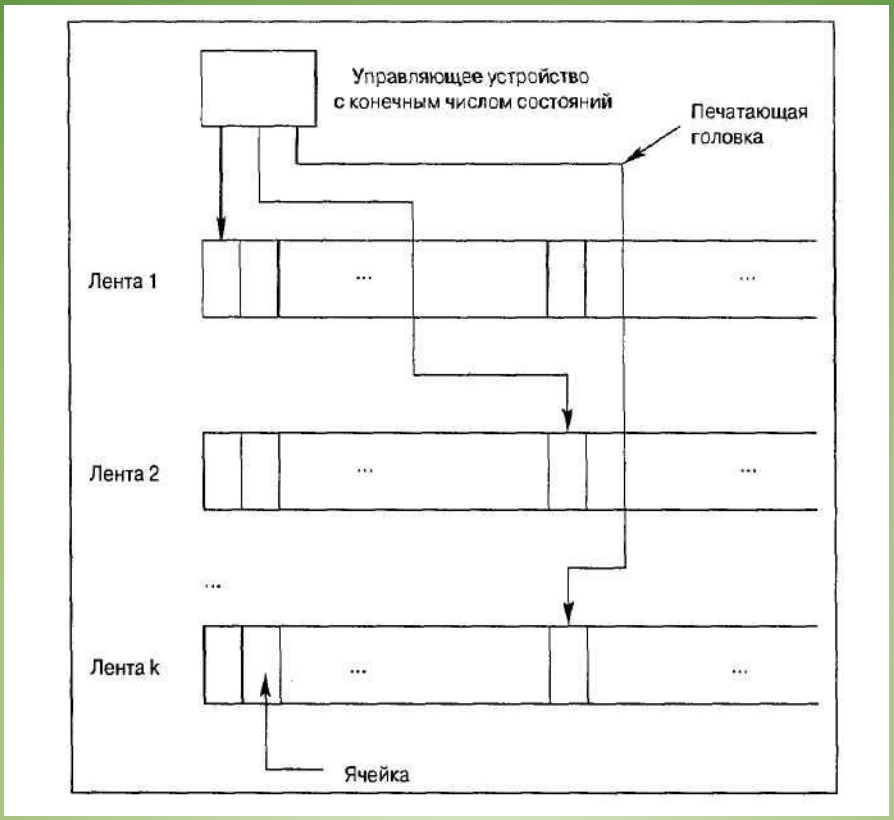
В нашем варианте машина Тьюринга состоит из управляющего устройства с конечным числам состояний (finite-state control unit), k≥1 лент (tapes) и такого же количества *головок* (tapeheads).

Управляющее устройство контролирует операции, выполняемые головками, которые считывают информацию с лент или записывают ее на ленту.



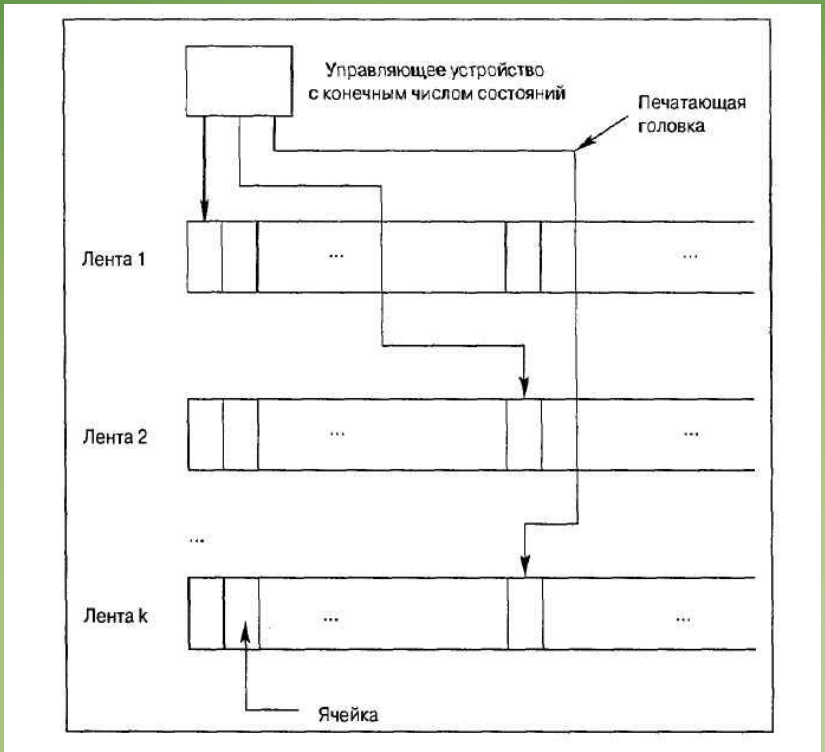
Каждая головка имеет доступ к *своей* ленте и может перемещаться вдоль нее влево и вправо.
Каждая лента разделена на бесконечное количество *ячеек* (cells).

Машина решает задачу, перемещая головку вдоль строки, состоящей из конечного количества символов, расположенных последовательно, начиная с крайней левой ячейки.



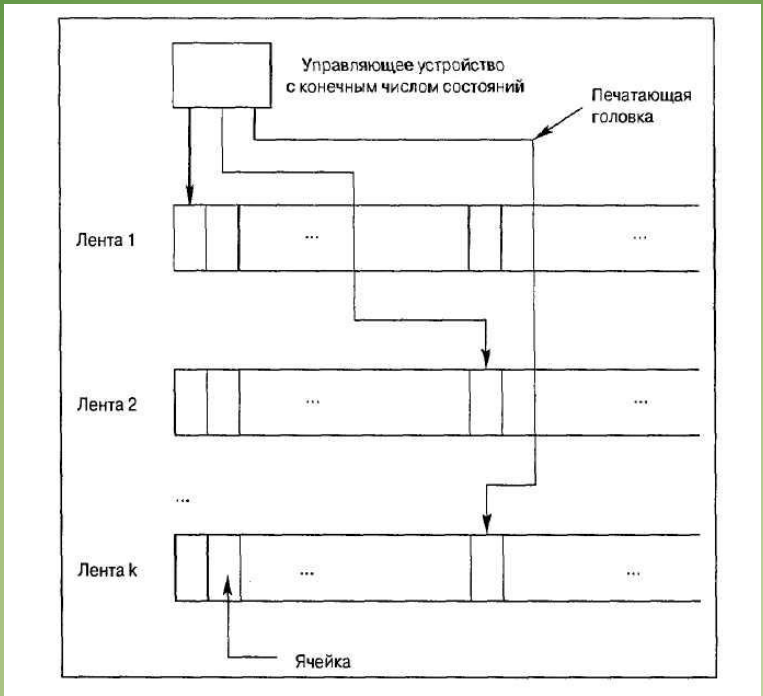
Каждый символ занимает одну ячейку, а оставшиеся ячейки ленты, расположенные справа, остаются *пустыми* (blank). Описанная выше строка называется *исходными данными задачи* (input).

Сканирование начинается с крайней слева ячейки ленты, содержащей исходные данные, когда машина находится в предписанном *начальном состоянии* (initial state).



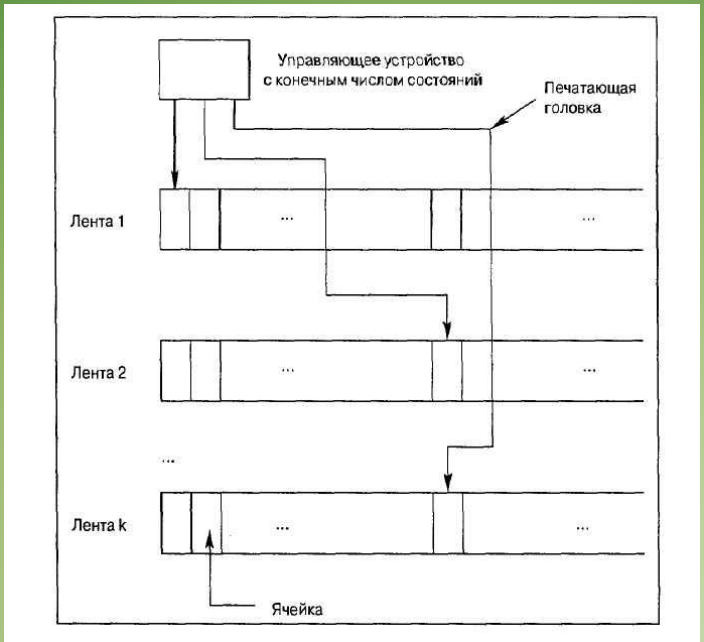
В каждый момент времени только одна из головок имеет доступ к своей ленте.

Операция доступа головки к своей ленте называется *тактом* (legal move).



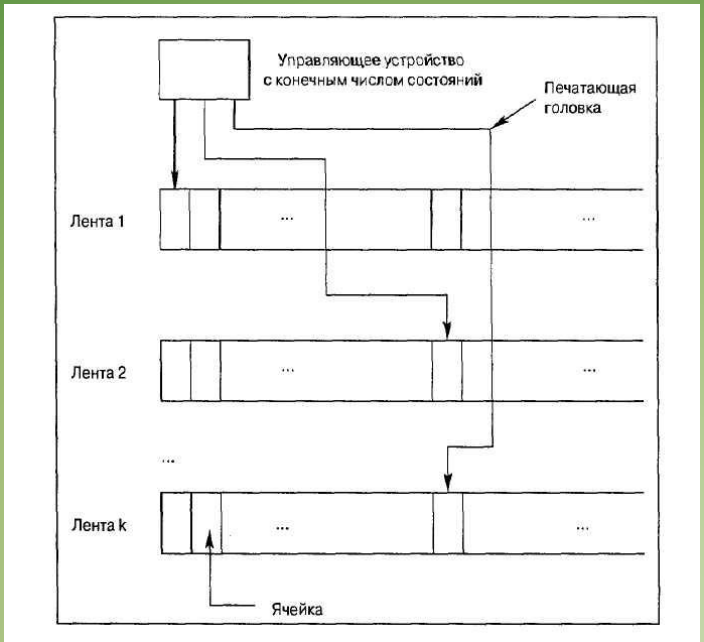
Если машина начинает работу с начального состояния, последовательно выполняет такты, сканирует исходную строку и завершает работу, достигая *заключительного состояния* (termination condition), говорят, что машина *распознает* (recognize) исходные данные.

В противном случае в некоторый момент машина не может выполнить очередной такт и останавливается, не распознав исходные данные.



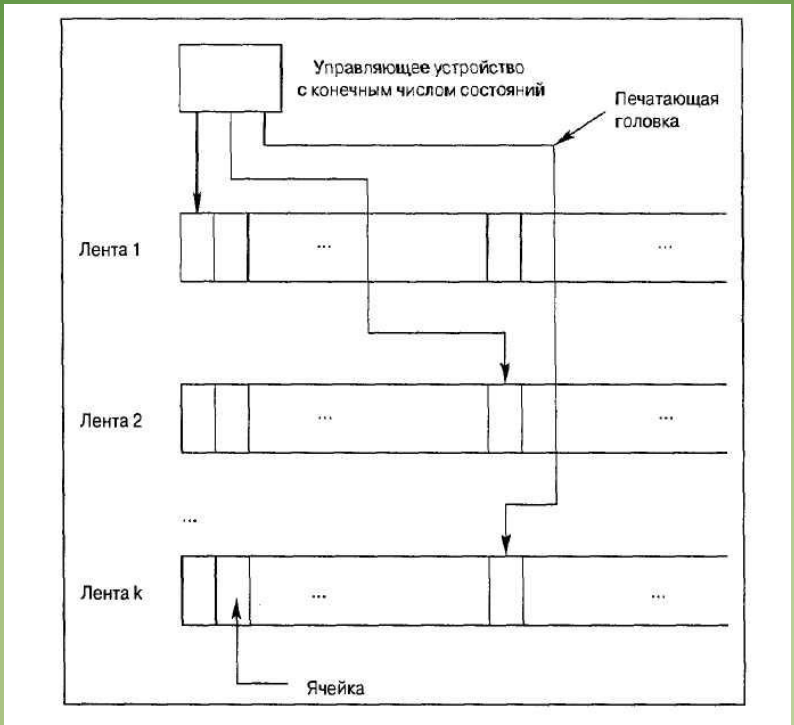
Исходные данные, распознаваемые машиной Тьюринга, называются *предложением* (instance) распознаваемого *языка* (language).
Для каждой конкретной задачи машину Тьюринга можно полностью определить с помощью функции, описывающей работу управляющего устройства с конечным числом состояний.

Такая функция может иметь вид таблицы, в которой для каждого состояния указана операция, выполняемая на следующем такте.



Количество тактов $T_м$, которые машина Тьюринга *M* должна выполнить при распознавании исходной строки, называется продолжительностью работы или *временной сложностью* (time complexity) машины *M*.

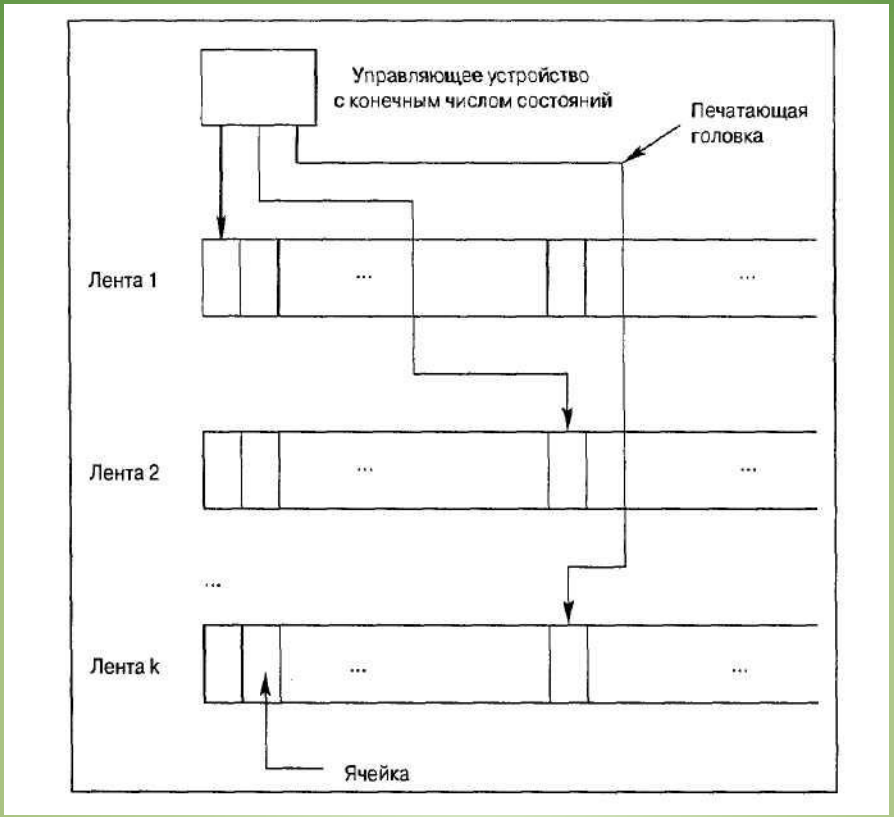
Очевидно, что величину $T_м$ можно представить в виде функции $T_м(n) : \mathrm{N} \rightarrow \mathrm{N}$, где *n* — *длина* (length), или *размер* (size), исходного предложения, т.е. количество символов, из которых состоит исходная строка, когда машина *M* пребывает в начальном состоянии.



Легко видеть, что $T_м(n) \geq n$.

Кроме временных ограничений, машина M имеет ограничения памяти S_M, представляющие собой количество ячеек, которые доступны для записи.

Величину S_M также можно представить в виде функции $S_м(n) : N \rightarrow N$, которая называется *пространственной сложностью* (space complexity) машины M.



Если машина начинает работу с начального состояния, последовательно выполняет такты, сканирует исходную строку и завершает работу, достигая *заключительного состояния* (termination condition), говорят, что машина *распознает* (recognize) исходные данные.

Детерминированное полиномиальное время

Функция *p(n)* является полиномиальной по целому аргументу *n,* если она имеет вид:

$$p(n) = c_k n^k + c_{k-1} n^{k-1} + \cdots + c_1 n + c_0,$$

где числа *к* и c_i ($i = 0,1,2,\ldots ,к$) — целые константы, причем $c_i \neq 0$.

Число $к \geq 0$ называется степенью (degree) полинома и обозначается как deg(p(n)),
а числа c_i называются коэффициентами (coefficients) полинома *p(n)*.

Определение : **Класс** □.

Символом □ обозначается класс языков, имеющих следующие характеристики.

Язык L принадлежит классу □, если существует машина Тьюринга M и полином p(n), такие что машина M распознает любое предложение $I \in L$ за время $T_м(n)$,

где $T_м(n) \leq p(n)$ для всех неотрицательных целых чисел n, а n — параметр, задающий длину предложения I.

В этом случае говорят, что язык L распознается за полиномиальное время.

Языки, распознаваемые за полиномиальное время, считаются *"всегда* простыми", а полиномиальные машины Тьюринга — *"всегда* эффективными".

Рассмотрим смысл слова *"всегда"*. Все машины Тьюринга, распознающие язык *L* из класса □, называются детерминированными (deterministic).

Детерминированная машина Тьюринга порождает результат, который полностью предопределен исходными данными и начальным состоянием машины.

Иначе говоря, повторный запуск машины Тьюринга с теми же исходными данными и начальным состоянием приводит к идентичным результатам.

В работах, посвященных теории вычислительной сложности, задача считается решенной, только если любой экземпляр данной задачи можно решить с помощью одной и той же машины Тьюринга (т.е. с помощью одного и того же метода).

Только в этом случае алгоритм считается достаточно общим и может называться методом.

Пример - язык DIV3

Пусть DIV3 — множество неотрицательных целых чисел, кратных трем.
Покажем, что DIV3 $\in P$.

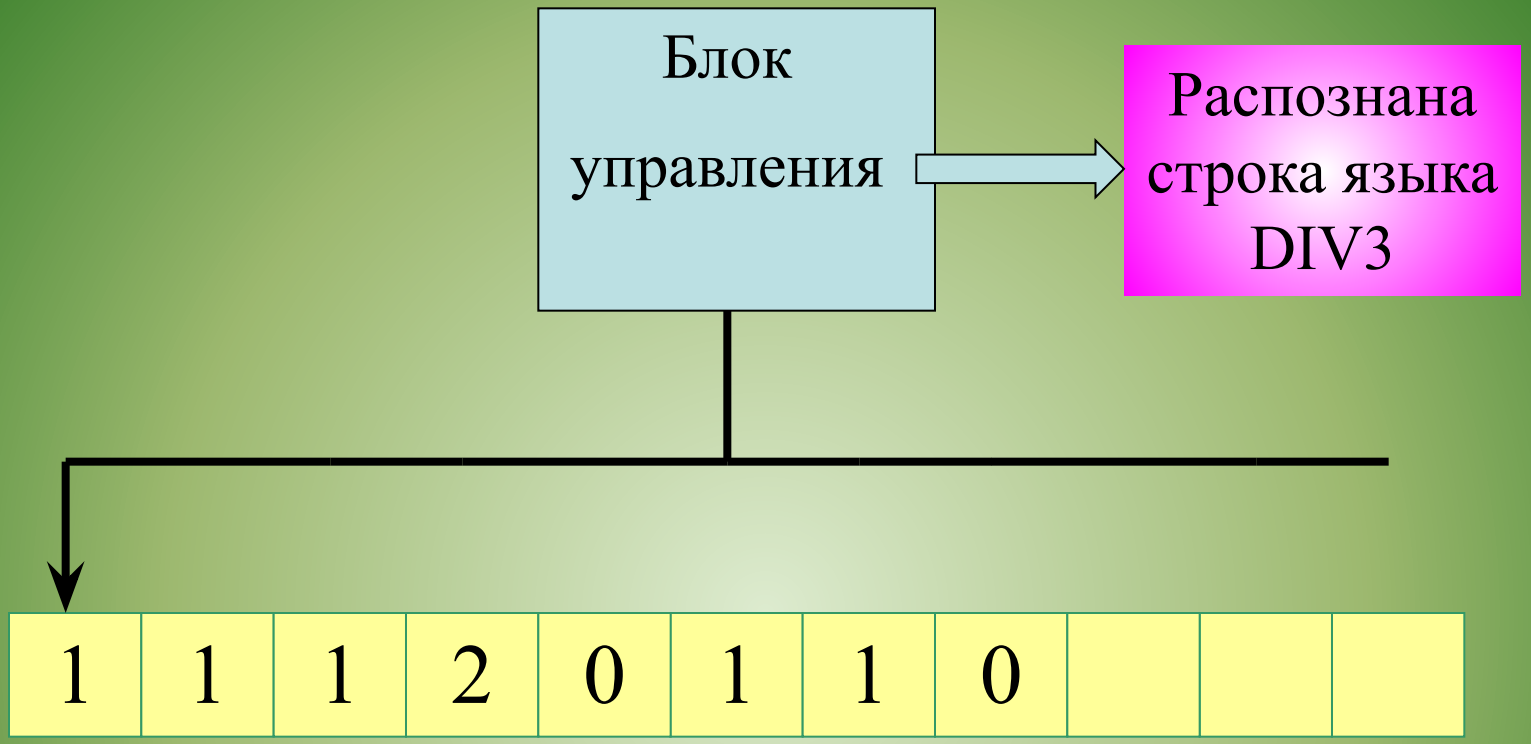
BaseForm[3333,3]

11120110_3

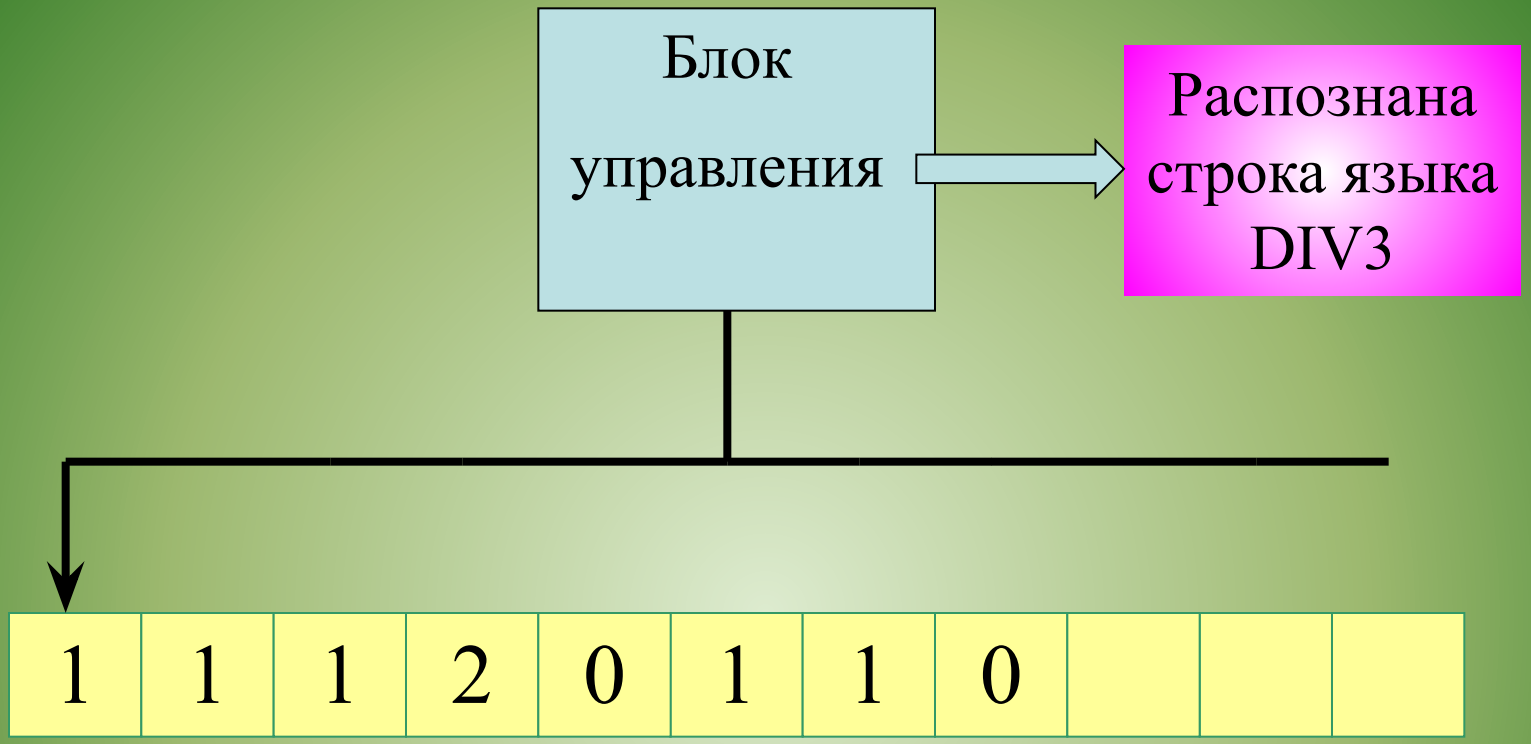
Для того чтобы распознать язык DIV3 за полиномиальное время, построим машину Тьюринга с одной лентой.

Если записать исходные данные в виде троичных чисел (в позиционной системе счисления по основанию 3), т.е. в виде строки символов из множества {0,1,2}, то задача распознавания становится тривиальной:

Исходная строка *x* принадлежит языку DIV3 тогда и только тогда, когда последняя цифра в строке *x* равна нулю.



Следовательно, создаваемая машина должна просто перемещать головку вправо, пока не обнаружит пустой символ.
Машина должна остановиться и выдать ответ "ДА", если и только если последний непустой символ был равен нулю.



Очевидно, что данная машина может распознавать любое предложение, состоящее из цифр, причем **количество шагов алгоритма равно длине исходной строки.**

Следовательно, DIV3 $\in P$.

$T_{DIV3}(n) = n$. Машина распознает язык DIV3 за полиномиальное время.

Полиномиальные вычислительные задачи

По определению класс является классом языков, распознаваемых за полиномиальное время.

Задача распознавания языка является задачей **принятия решений** (decisional problem).

При любых исходных данных результатом решения такой задачи является ответ "ДА" или "НЕТ".

Однако класс является более широким и содержит **полиномиальные вычислительные задачи** (polynomial-time computational problems).

При любых исходных данных результатом решения таких задач является более общий ответ, чем "ДА" и "НЕТ".

Поскольку машина Тьюринга может записывать символы на ленту, она позволяет решать такие задачи.

Вычислительное устройство, имеющее неймановскую архитектуру (иначе говоря, всем известную современную компьютерную архитектуру),
состоит из счетчика, памяти и центрального процессора (central processor unit — CPU),
поочередно выполняющего элементарные команды, называемые *микрокомандами.*

Load (Загрузить)
Store (Сохранить)
Add (Сложить)
Comp (Дополнение)
Jump (Перейти)
JumpZ (Условный переход)
Stop (Остановиться)

Хорошо известно ,что перечисленных выше микрокоманд достаточно для создания алгоритмов, решающих любые арифметические задачи на неймановском компьютере.

Можно доказать ,что каждую микрокоманду из указанного выше набора, можно имитировать на машине Тьюринга за полиномиальное время.

Следовательно, задачу, которую можно решить за полиномиальное время на неймановском компьютере (т.е. количество микроинструкций, используемых в алгоритме, представляет собой значение полинома, зависящего от размера исходных данных),
можно решить за полиномиальное время и с помощью машины Тьюринга.

Это возможно благодаря тому, что для любых полиномов $p(n)$ и $q(n)$ произвольные арифметические комбинации $p(n)$, $q(n)$, $p(q(n))$ и $q(p(n))$ также являются полиномами, зависящими от аргумента n.

□-символика
(order notation)

Символом □(*f*(*n*)) *обозначается функция* *g*(*n*), *для которой существует константа* c > 0 *и натуральное число N,*
такие что |*g*(*n*)| ≤ c| *f*(*n*)| *для всех* n ≥ *N.*

Используя O – символику можно отобразить временну*ю* сложность алгоритмов, рассмотрим следующую теорему:

Теорема.
Наибольший общий делитель $\gcd(a, b)$ *можно вычислить с помощью не более чем* $2\max(|a|, |b|)$ *операций модулярной арифметики.*

Эту теорему впервые доказал Ж. Ламе (1795-1870) (G. Lame). Ее можно считать первой теоремой в теории вычислительной сложности.

$\lceil x \rceil$ - минимальное целое число, большее или равное числу x;

функция **Ceiling[x]** *в пакете Mathematica*

$\lfloor x \rfloor$ - максимальное целое число, не превосходящее число x;

функция **Floor[x]** *в пакете Mathematica*

Обозначение $|x|$ - *длина целого числа* x, равная $1+\lfloor \log_2 x \rfloor$ для $x \geq 1$, или модуль числа x.

Таким образом, временная сложность алгоритма вычисления наибольшего общего делителя (при условии a > b) может быть определена как: □(log a).
Не указывая явно основание логарифма.

□-символика для поразрядных вычислений

Для оценки сложности **поразрядных** (bitwise) арифметических операций используется модифицированная □-символика.
В поразрядных вычислениях все переменные принимают значения нуль или единица, а операции носят не арифметический, а логический характер:

∧ (для операции AND),
∨ (для операции OR),
⊕ (для операции XOR, т.е. "исключающего или") и
¬ (для операции NOT).

В рамках модели поразрядных вычислений на сложение и вычитание двух целых чисел i и j затрачиваются $\max(|i|, |j|)$ побитовых операций, т.е. порядок временной сложности равен $\Theta(\max(|i|, |j|))$.

На умножение и деление двух целых чисел i и j затрачиваются $|i| \cdot |j|$ побитовых операций, т.е. порядок их временной сложности равен $\Theta(\log i \times \log j)$.

Поразрядные оценки сложности основных операций в модулярной арифметике

Операция над $a, b \in_U [1, n)$	Сложность
$a \pm b \pmod n$	$O_B(\log n)$
$a \cdot b \pmod n$	$O_B((\log n)^2)$
$b^{-1} \pmod n$	$O_B((\log n)^2)$
$a/b \pmod n$	$O_B((\log n)^2)$
$a^b \pmod n$	$O_B((\log n)^3)$

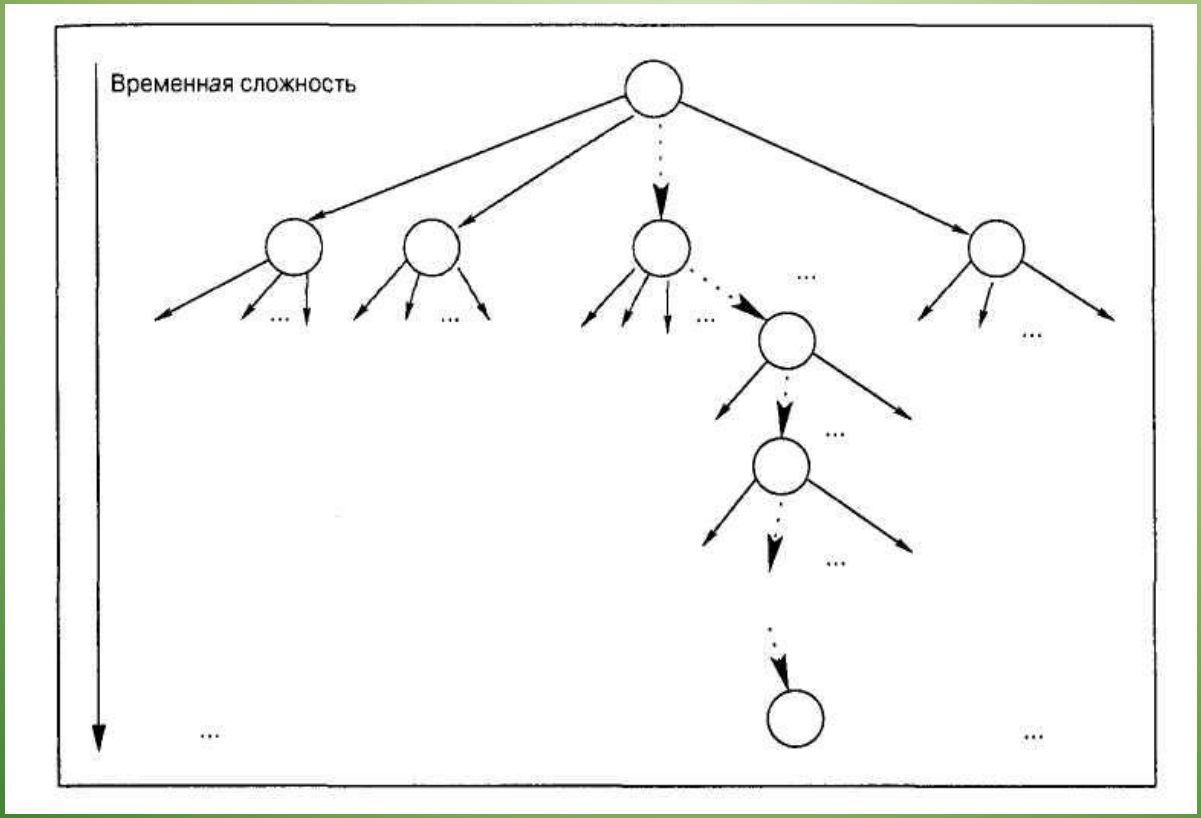
Недетерминированное полиномиальное время

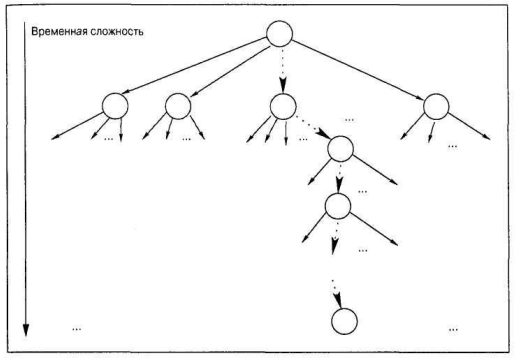
Недетерминированной машиной Тьюринга (non-deterministic Turing machine) называется устройство, на каждом такте работы которого существует конечное количество вариантов следующего такта.

Входная строка считается распознанной, если существует хотя бы одна последовательность разрешенных тактов, начинающаяся считыванием первого символа строки и завершающаяся считыванием последнего символа.

Такая последовательность тактов называется *последовательностью распознавания* (recognition sequence).

Работу недетерминированной машины Тьюринга можно представить в виде серии догадок.
В этом случае последовательность распознавания представляет собой серию правильных догадок.
Таким образом, все возможные такты образуют дерево, называемое **вычислительным деревом** (computational tree) недетерминированной машины Тьюринга.





Размер (количество узлов) этого дерева экспоненциально зависит от размера входа.

Однако, поскольку количество тактов в последовательности распознавания входной строки равно глубине дерева *d,* получаем, что *d* = □ (log (количество узлов дерева)) и количество тактов в последовательности распознавания **полиномиально** зависит от размера входной строки.

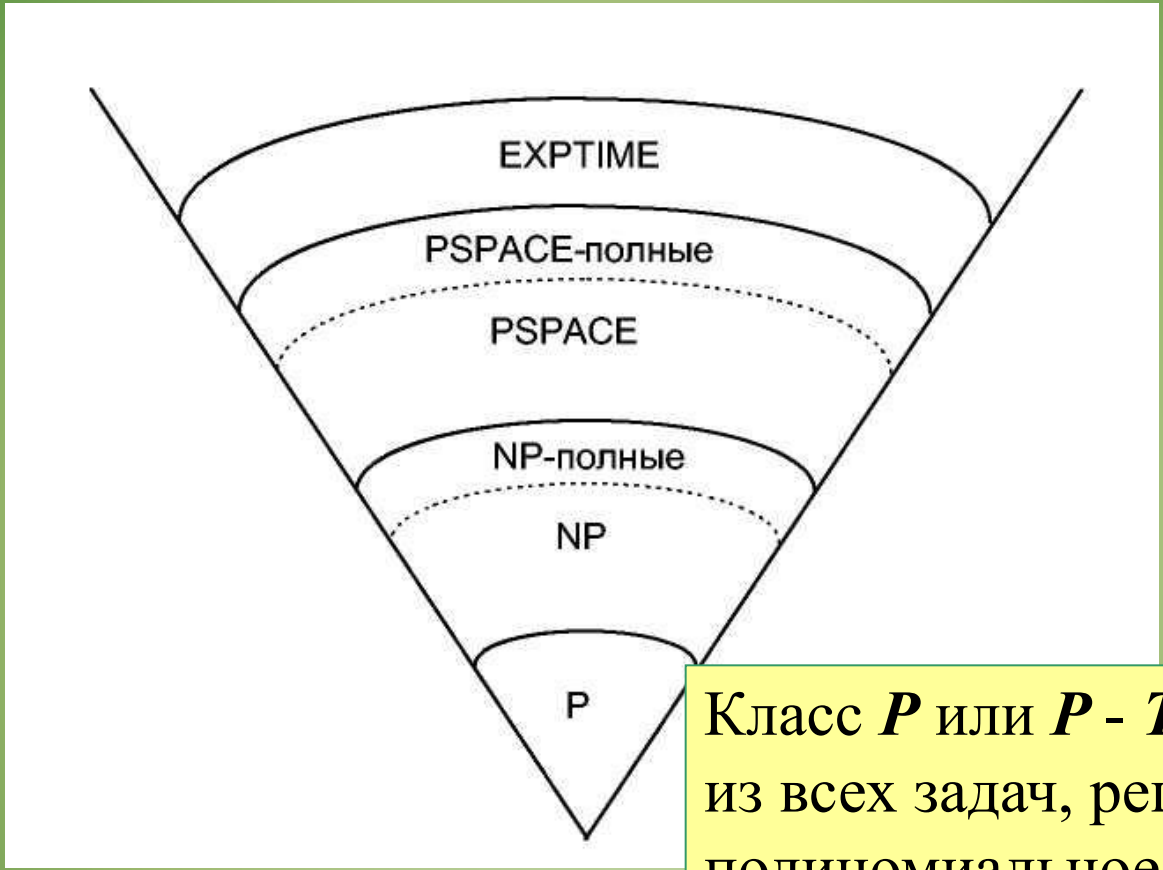
Итак, временная сложность распознавания строки с помощью серии правильных догадок полиномиально зависит от размера исходных данных.

Язык принадлежит классу □□, если он распознается недетерминированной машиной Тьюринга за полиномиальное время.

Итак, временная сложность распознавания строки с помощью серии правильных догадок **полиномиально** зависит от размера исходных данных.

Определение : ***Язык принадлежит классу □□, если он распознается недетерминированной машиной Тьюринга за полиномиальное время.***

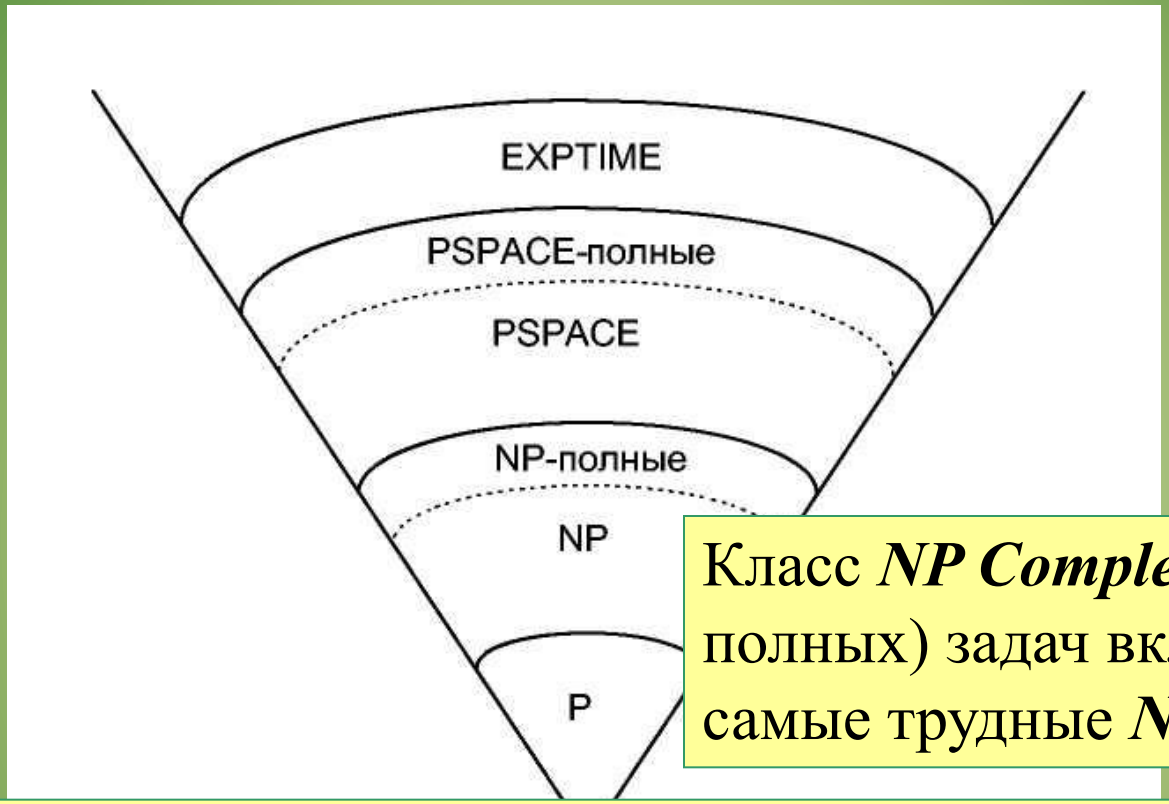
Классы сложности



Класс ***P*** или ***P - TIME*** состоит из всех задач, решаемых за полиномиальное время

Задачи, которые решаются за полиномиальное время называются ***решаемыми***, так как они обычно могут быть решены для задач достаточно большой размерности ***n***.

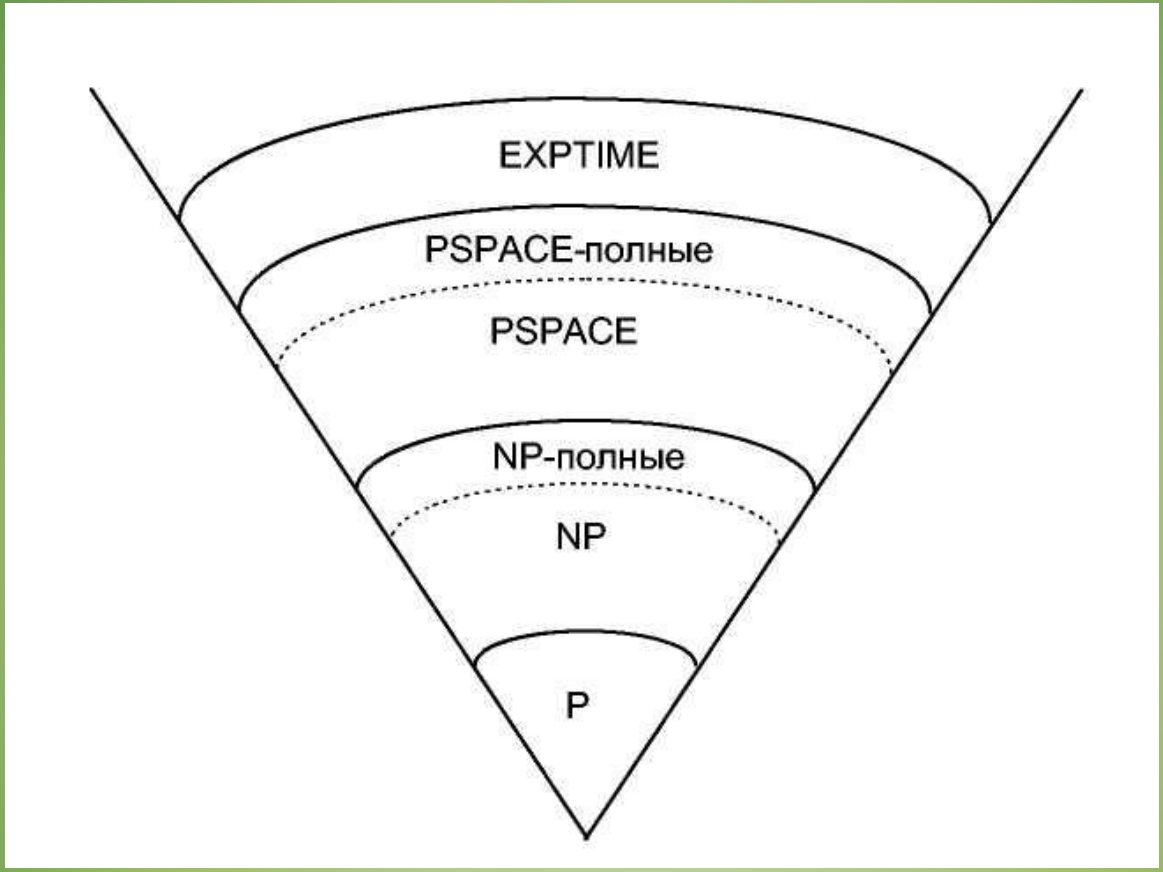
Классы сложности



Класс ***NP Complete*** (***NP*** - полных) задач включает все самые трудные ***NP*** задачи.

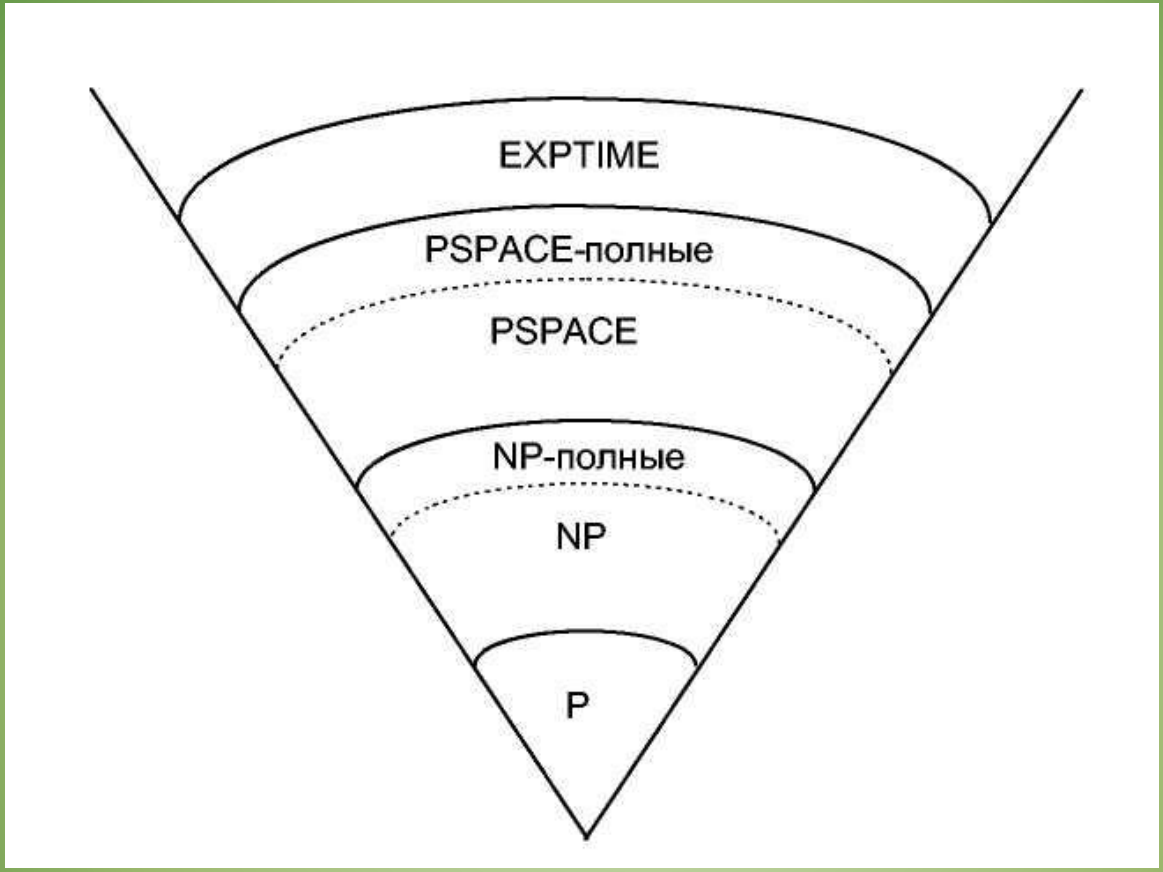
Класс ***NP*** или ***NP - TIME***, состоит из
всех задач решаемых за полиномиальное время на недетерминированной машине Тьюринга, способной параллельно выполнять
неограниченное количество независимых вычислений.

Классы сложности



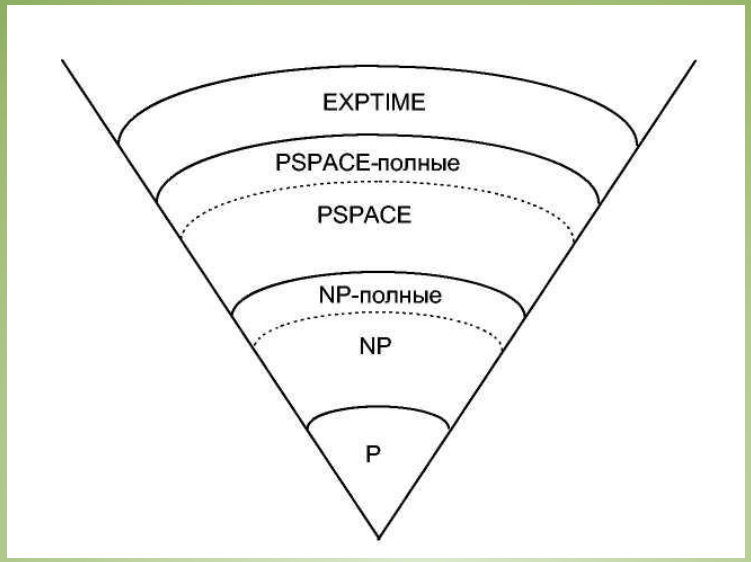
Класс ***PSPACE*** состоит из задач, требующих полиномиальных объемов машинной памяти, но не обязательно решаемых за полиномиальное время.

Классы сложности



Класс EXPTIME – эти задачи решаются за экспоненциальное время

Классы сложности



Таким образом, выбор наиболее сложных задач, для которых известно решение, и использование их в основе построения криптосистемы, позволяет создавать практически устойчивые шифры, раскрытие которых в принципе возможно, но для этого потребуется столько времени, что эта процедура дешифрования теряет практический смысл.