Математическая логика и теория алгоритмов
Лекция 13
Рекурсивные функции
Нормальные алгоритмы Маркова

Куценко Дмитрий Александрович

Белгородский государственный технологический университет имени В. Г. Шухова

Институт информационных технологий и управляющих систем

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

2 декабря 2011 г.

Рекурсивные функции

Термин «рекурсивные функции» в теории алгоритмов используют для обозначения трёх множеств функций:

1. примитивно рекурсивные функции;
2. общерекурсивные функции;
3. частично рекурсивные функции.

Термины «частично рекурсивная функция» и «общерекурсивная функция» прижились в силу исторических причин.
По сути они являются результатом неточного перевода английских терминов partial recursive function и total recursive function, которые по смыслу более правильно переводить как «частично определённые рекурсивные функции» и «везде определённые рекурсивные функции» соответственно.

Частично рекурсивные функции совпадают с множеством вычислимых по Тьюрингу функций.

Рекурсивные функции

Термин «рекурсивные функции» в теории алгоритмов используют для обозначения трёх множеств функций:

1. примитивно рекурсивные функции;
2. общерекурсивные функции;
3. частично рекурсивные функции.

Термины «частично рекурсивная функция» и «общерекурсивная функция» прижились в силу исторических причин.
По сути они являются результатом неточного перевода английских терминов partial recursive function и total recursive function, которые по смыслу более правильно переводить как «частично определённые рекурсивные функции» и «везде определённые рекурсивные функции» соответственно.

Частично рекурсивные функции совпадают с множеством вычислимых по Тьюрингу функций.

Рекурсивные функции

Термин «рекурсивные функции» в теории алгоритмов используют для обозначения трёх множеств функций:

1. примитивно рекурсивные функции;
2. общерекурсивные функции;
3. частично рекурсивные функции.

Термины «частично рекурсивная функция» и «общерекурсивная функция» прижились в силу исторических причин.
По сути они являются результатом неточного перевода английских терминов partial recursive function и total recursive function, которые по смыслу более правильно переводить как «частично определённые рекурсивные функции» и «везде определённые рекурсивные функции» соответственно.

Частично рекурсивные функции совпадают с множеством вычислимых по Тьюрингу функций.

Рекурсивные функции

Термин «рекурсивные функции» в теории алгоритмов используют для обозначения трёх множеств функций:

1. примитивно рекурсивные функции;
2. общерекурсивные функции;
3. частично рекурсивные функции.

Термины «частично рекурсивная функция» и «общерекурсивная функция» прижились в силу исторических причин.
По сути они являются результатом неточного перевода английских терминов partial recursive function и total recursive function, которые по смыслу более правильно переводить как «частично определённые рекурсивные функции» и «везде определённые рекурсивные функции» соответственно.

Частично рекурсивные функции совпадают с множеством вычислимых по Тьюрингу функций.

Рекурсивные функции

Термин «рекурсивные функции» в теории алгоритмов используют для обозначения трёх множеств функций:

1. примитивно рекурсивные функции;
2. общерекурсивные функции;
3. частично рекурсивные функции.

Термины «частично рекурсивная функция» и «общерекурсивная функция» прижились в силу исторических причин.
По сути они являются результатом неточного перевода английских терминов partial recursive function и total recursive function, которые по смыслу более правильно переводить как «частично определённые рекурсивные функции» и «везде определённые рекурсивные функции» соответственно.

Частично рекурсивные функции совпадают с множеством вычислимых по Тьюрингу функций.

Рекурсивные функции

Термин «рекурсивные функции» в теории алгоритмов используют для обозначения трёх множеств функций:

1. примитивно рекурсивные функции;
2. общерекурсивные функции;
3. частично рекурсивные функции.

Термины «частично рекурсивная функция» и «общерекурсивная функция» прижились в силу исторических причин.
По сути они являются результатом неточного перевода английских терминов partial recursive function и total recursive function, которые по смыслу более правильно переводить как «частично определённые рекурсивные функции» и «везде определённые рекурсивные функции» соответственно.

Частично рекурсивные функции совпадают с множеством вычислимых по Тьюрингу функций.

Классы рекурсивных функций

Рекурсивные функции были введены Гёделем с целью формализации понятия вычислимости.

Определения этих трёх классов сильно связаны.

Множество частично рекурсивных функций ($\mathscr{F}_{\text{ЧРФ}}$) включает в себя множество общерекурсивных функций ($\mathscr{F}_{\text{ОРФ}}$), а множество общерекурсивных функций включает в себя множество примитивно рекурсивные функций ($\mathscr{F}_{\text{ПРФ}}$):

$$\mathscr{F}_{\text{ПРФ}} \subset \mathscr{F}_{\text{ОРФ}} \subset \mathscr{F}_{\text{ЧРФ}}.$$

Частично рекурсивные функции иногда называют просто рекурсивными функциями.

Классы рекурсивных функций

Рекурсивные функции были введены Гёделем с целью формализации понятия вычислимости.

Определения этих трёх классов сильно связаны.

Множество частично рекурсивных функций ($\mathscr{F}_{\text{ЧРФ}}$) включает в себя множество общерекурсивных функций ($\mathscr{F}_{\text{ОРФ}}$), а множество общерекурсивных функций включает в себя множество примитивно рекурсивные функций ($\mathscr{F}_{\text{ПРФ}}$):

$$\mathscr{F}_{\text{ПРФ}} \subset \mathscr{F}_{\text{ОРФ}} \subset \mathscr{F}_{\text{ЧРФ}}.$$

Частично рекурсивные функции иногда называют просто рекурсивными функциями.

Классы рекурсивных функций

Рекурсивные функции были введены Гёделем с целью формализации понятия вычислимости.

Определения этих трёх классов сильно связаны.

Множество частично рекурсивных функций ($\mathscr{F}_{\text{ЧРФ}}$) включает в себя множество общерекурсивных функций ($\mathscr{F}_{\text{ОРФ}}$), а множество общерекурсивных функций включает в себя множество примитивно рекурсивные функций ($\mathscr{F}_{\text{ПРФ}}$):

$$\mathscr{F}_{\text{ПРФ}} \subset \mathscr{F}_{\text{ОРФ}} \subset \mathscr{F}_{\text{ЧРФ}}.$$

Частично рекурсивные функции иногда называют просто рекурсивными функциями.

Классы рекурсивных функций

Рекурсивные функции были введены Гёделем с целью формализации понятия вычислимости.

Определения этих трёх классов сильно связаны.

Множество частично рекурсивных функций ($\mathscr{F}_{\text{ЧРФ}}$) включает в себя множество общерекурсивных функций ($\mathscr{F}_{\text{ОРФ}}$), а множество общерекурсивных функций включает в себя множество примитивно рекурсивные функций ($\mathscr{F}_{\text{ПРФ}}$):

$$\mathscr{F}_{\text{ПРФ}} \subset \mathscr{F}_{\text{ОРФ}} \subset \mathscr{F}_{\text{ЧРФ}}.$$

Частично рекурсивные функции иногда называют просто рекурсивными функциями.

Классы рекурсивных функций

Рекурсивные функции были введены Гёделем с целью формализации понятия вычислимости.

Определения этих трёх классов сильно связаны.

Множество частично рекурсивных функций ($\mathscr{F}_{\text{ЧРФ}}$) включает в себя множество общерекурсивных функций ($\mathscr{F}_{\text{ОРФ}}$), а множество общерекурсивных функций включает в себя множество примитивно рекурсивные функций ($\mathscr{F}_{\text{ПРФ}}$):

$$\mathscr{F}_{\text{ПРФ}} \subset \mathscr{F}_{\text{ОРФ}} \subset \mathscr{F}_{\text{ЧРФ}}.$$

Частично рекурсивные функции иногда называют просто рекурсивными функциями.

Примитивно рекурсивные функции

Примитивно рекурсивная функция — функция от конечного числа аргументов, которую можно получить применением конечного числа операторов подстановки и примитивной рекурсии, исходя лишь из базовых примитивно рекурсивных функций.

Базовые примитивно рекурсивные функции

1. Нуль-функция O — функция, всегда возвращающая нуль: $O(x) = 0$.
2. Функция следования S одного переменного, сопоставляющая любому натуральному числу x непосредственно следующее за ним натуральное число $x + 1$: $S(x) = x + 1$.
3. Функции-проекторы I_n^m, где $1 \leqslant m \leqslant n$, от n переменных, сопоставляющие любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число x_m из этого набора: $I_n^m(x_1, \ldots, x_m, \ldots, x_n) = x_m$.

Примитивно рекурсивные функции

Примитивно рекурсивная функция — функция от конечного числа аргументов, которую можно получить применением конечного числа операторов подстановки и примитивной рекурсии, исходя лишь из базовых примитивно рекурсивных функций.

Базовые примитивно рекурсивные функции

1. Нуль-функция O — функция, всегда возвращающая нуль: $O(x) = 0$.
2. Функция следования S одного переменного, сопоставляющая любому натуральному числу x непосредственно следующее за ним натуральное число $x + 1$: $S(x) = x + 1$.
3. Функции-проекторы I_n^m, где $1 \leqslant m \leqslant n$, от n переменных, сопоставляющие любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число x_m из этого набора: $I_n^m(x_1, \ldots, x_m, \ldots, x_n) = x_m$.

Примитивно рекурсивные функции

Примитивно рекурсивная функция — функция от конечного числа аргументов, которую можно получить применением конечного числа операторов подстановки и примитивной рекурсии, исходя лишь из базовых примитивно рекурсивных функций.

Базовые примитивно рекурсивные функции

1. Нуль-функция O — функция, всегда возвращающая нуль: $O(x) = 0$.
2. Функция следования S одного переменного, сопоставляющая любому натуральному числу x непосредственно следующее за ним натуральное число $x + 1$: $S(x) = x + 1$.
3. Функции-проекторы I_n^m, где $1 \leqslant m \leqslant n$, от n переменных, сопоставляющие любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число x_m из этого набора: $I_n^m(x_1, \ldots, x_m, \ldots, x_n) = x_m$.

Примитивно рекурсивные функции

Примитивно рекурсивная функция — функция от конечного числа аргументов, которую можно получить применением конечного числа операторов подстановки и примитивной рекурсии, исходя лишь из базовых примитивно рекурсивных функций.

Базовые примитивно рекурсивные функции

1. Нуль-функция O — функция, всегда возвращающая нуль: $O(x) = 0$.
2. Функция следования S одного переменного, сопоставляющая любому натуральному числу x непосредственно следующее за ним натуральное число $x + 1$: $S(x) = x + 1$.
3. Функции-проекторы I_n^m, где $1 \leqslant m \leqslant n$, от n переменных, сопоставляющие любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число x_m из этого набора: $I_n^m(x_1, \ldots, x_m, \ldots, x_n) = x_m$.

Оператор подстановки (суперпозиция функций)

Пусть f — функция от m переменных, а $g_1, \ldots, g_m$ — упорядоченный набор функций от n переменных каждая.

Тогда результатом подстановки функций $g_1, \ldots, g_m$ в функцию f называется функция h от n переменных, сопоставляющая любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число

$$h(x_1, \ldots, x_n) = f\big(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)\big).$$

Ясно, что если все функции $g_1, \ldots, g_m$ и f всюду определены, то функция h всюду определена.

Функция h будет не всюду определённой, если хотя бы одна из функций $g_1, \ldots, g_m$ не всюду определена, или если можно найти такие значения аргументов $a_1, \ldots, a_n$, что $b_i = g_i(a_1, \ldots, a_n)$, но $f(b_1, \ldots, b_m)$ не определено $(i = 1, \ldots, m)$.

Оператор подстановки (суперпозиция функций)

Пусть f — функция от m переменных, а $g_1, \ldots, g_m$ — упорядоченный набор функций от n переменных каждая. Тогда результатом подстановки функций $g_1, \ldots, g_m$ в функцию f называется функция h от n переменных, сопоставляющая любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число

$$h(x_1, \ldots, x_n) = f\big(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)\big).$$

Ясно, что если все функции $g_1, \ldots, g_m$ и f всюду определены, то функция h всюду определена.

Функция h будет не всюду определённой, если хотя бы одна из функций $g_1, \ldots, g_m$ не всюду определена, или если можно найти такие значения аргументов $a_1, \ldots, a_n$, что $b_i = g_i(a_1, \ldots, a_n)$, но $f(b_1, \ldots, b_m)$ не определено $(i = 1, \ldots, m)$.

Оператор подстановки (суперпозиция функций)

Пусть f — функция от m переменных, а $g_1, \ldots, g_m$ — упорядоченный набор функций от n переменных каждая. Тогда результатом подстановки функций $g_1, \ldots, g_m$ в функцию f называется функция h от n переменных, сопоставляющая любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число

$$h(x_1, \ldots, x_n) = f\big(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)\big).$$

Ясно, что если все функции $g_1, \ldots, g_m$ и f всюду определены, то функция h всюду определена.

Функция h будет не всюду определённой, если хотя бы одна из функций $g_1, \ldots, g_m$ не всюду определена, или если можно найти такие значения аргументов $a_1, \ldots, a_n$, что $b_i = g_i(a_1, \ldots, a_n)$, но $f(b_1, \ldots, b_m)$ не определено $(i = 1, \ldots, m)$.

Оператор подстановки (суперпозиция функций)

Пусть f — функция от m переменных, а $g_1, \ldots, g_m$ — упорядоченный набор функций от n переменных каждая. Тогда результатом подстановки функций $g_1, \ldots, g_m$ в функцию f называется функция h от n переменных, сопоставляющая любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число

$$h(x_1, \ldots, x_n) = f\big(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)\big).$$

Ясно, что если все функции $g_1, \ldots, g_m$ и f всюду определены, то функция h всюду определена.

Функция h будет не всюду определённой, если хотя бы одна из функций $g_1, \ldots, g_m$ не всюду определена, или если можно найти такие значения аргументов $a_1, \ldots, a_n$, что $b_i = g_i(a_1, \ldots, a_n)$, но $f(b_1, \ldots, b_m)$ не определено $(i = 1, \ldots, m)$.

Оператор подстановки (суперпозиция функций)

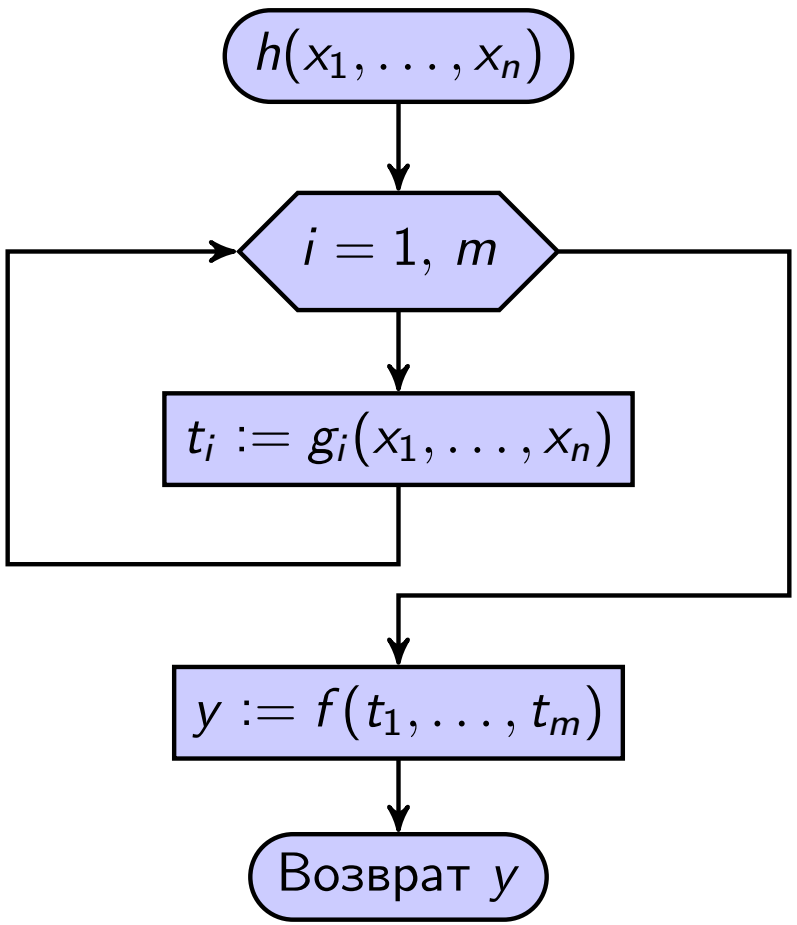
Пусть f — функция от m переменных, а $g_1, \ldots, g_m$ — упорядоченный набор функций от n переменных каждая.
Тогда результатом подстановки функций $g_1, \ldots, g_m$ в функцию f называется функция h от n переменных, сопоставляющая любому упорядоченному набору $x_1, \ldots, x_n$ натуральных чисел число

$$h(x_1, \ldots, x_n) = f\big(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)\big).$$

Ясно, что если все функции $g_1, \ldots, g_m$ и f всюду определены, то функция h всюду определена.
Функция h будет не всюду определённой, если хотя бы одна из функций $g_1, \ldots, g_m$ не всюду определена, или если можно найти такие значения аргументов $a_1, \ldots, a_n$, что $b_i = g_i(a_1, \ldots, a_n)$, но $f(b_1, \ldots, b_m)$ не определено $(i = 1, \ldots, m)$.

Если мы каким-либо образом умеем вычислять функции $g_1, \ldots, g_m$ и f, то функция h, являющаяся результатом подстановки функций $g_1, \ldots, g_m$ в функцию f, может быть вычислена с помощью следующего алгоритма:

Если мы каким-либо образом умеем вычислять функции $g_1, \ldots, g_m$ и f, то функция h, являющаяся результатом подстановки функций $g_1, \ldots, g_m$ в функцию f, может быть вычислена с помощью следующего алгоритма:



Оператор примитивной рекурсии

Пусть f — функция от n переменных, а g — функция от $n+2$ переменных.

Тогда результатом применения оператора примитивной рекурсии к паре функций f и g называется функция h от $n+1$ переменной вида

$$\begin{cases} h(x_1, \ldots, x_n,\ 0) = f(x_1, \ldots, x_n), \\ h(x_1, \ldots, x_n,\ y+1) = g\big(x_1, \ldots, x_n,\ y,\ h(x_1, \ldots, x_n,\ y)\big). \end{cases}$$

Очевидно, что если функции f и g всюду определены, то будет всюду определена и функция h.

Оператор примитивной рекурсии

Пусть f — функция от n переменных, а g — функция от $n+2$ переменных.
Тогда результатом применения оператора примитивной рекурсии к паре функций f и g называется функция h от $n+1$ переменной вида

$$\begin{cases} h(x_1, \ldots, x_n,\ 0) = f(x_1, \ldots, x_n), \\ h(x_1, \ldots, x_n,\ y+1) = g\big(x_1, \ldots, x_n,\ y,\ h(x_1, \ldots, x_n,\ y)\big). \end{cases}$$

Очевидно, что если функции f и g всюду определены, то будет всюду определена и функция h.

Оператор примитивной рекурсии

Пусть f — функция от n переменных, а g — функция от $n+2$ переменных.
Тогда результатом применения оператора примитивной рекурсии к паре функций f и g называется функция h от $n+1$ переменной вида

$$\begin{cases} h(x_1, \ldots, x_n,\ 0) = f(x_1, \ldots, x_n), \\ h(x_1, \ldots, x_n,\ y+1) = g\big(x_1, \ldots, x_n,\ y,\ h(x_1, \ldots, x_n,\ y)\big). \end{cases}$$

Очевидно, что если функции f и g всюду определены, то будет всюду определена и функция h.

Оператор примитивной рекурсии

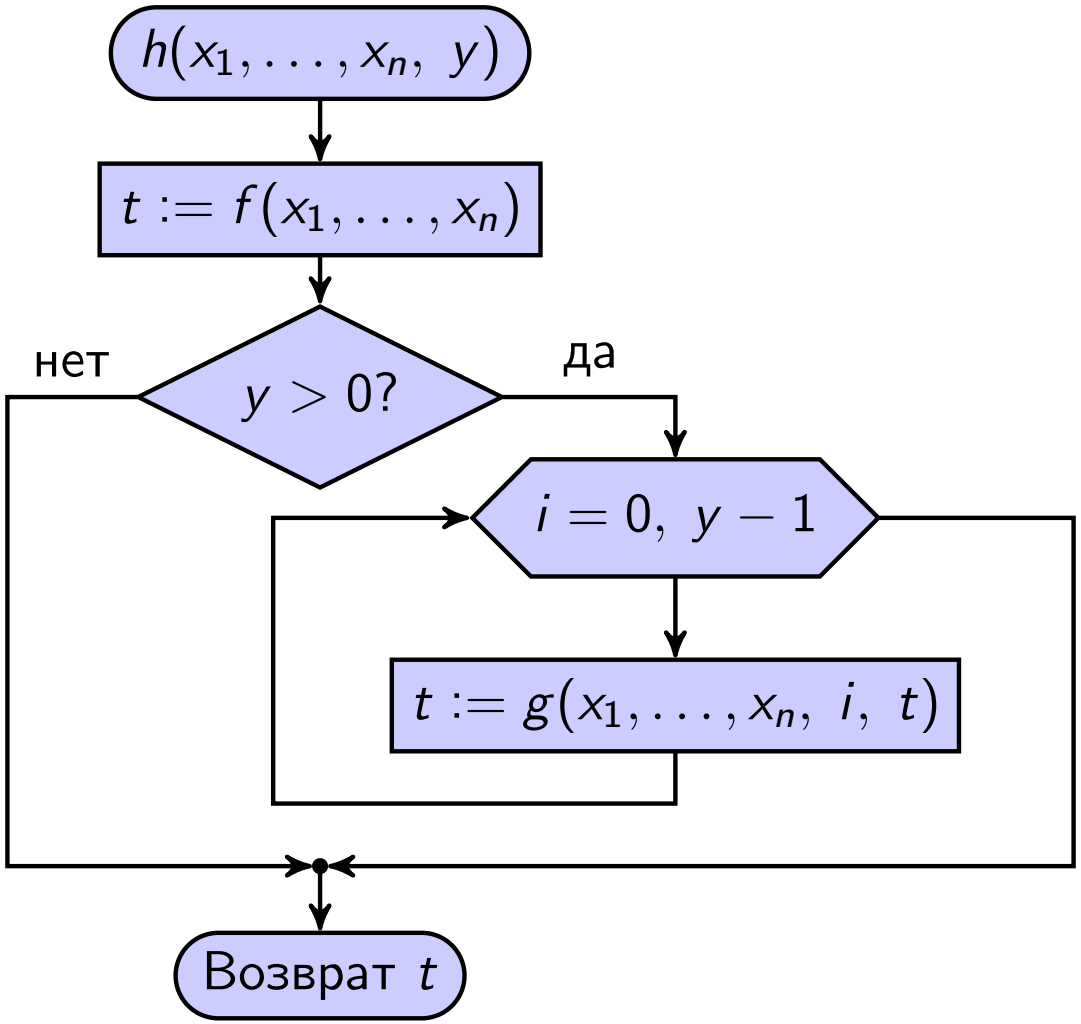
Пусть f — функция от n переменных, а g — функция от $n+2$ переменных.
Тогда результатом применения оператора примитивной рекурсии к паре функций f и g называется функция h от $n+1$ переменной вида

$$\begin{cases} h(x_1, \ldots, x_n,\ 0) = f(x_1, \ldots, x_n), \\ h(x_1, \ldots, x_n,\ y+1) = g\big(x_1, \ldots, x_n,\ y,\ h(x_1, \ldots, x_n,\ y)\big). \end{cases}$$

Очевидно, что если функции f и g всюду определены, то будет всюду определена и функция h.

Если мы можем вычислить функции f и g, то функция h, являющаяся результатом применения оператора примитивной рекурсии к паре функций f и g, вычисляется согласно следующему алгоритму:

Если мы можем вычислить функции f и g, то функция h, являющаяся результатом применения оператора примитивной рекурсии к паре функций f и g, вычисляется согласно следующему алгоритму:



Пример

Функция сложения двух натуральных чисел

$$\Sigma(x,\ y) = x + y$$

может быть рассмотрена в качестве примитивно рекурсивной функции двух переменных, получаемой в результате применения оператора примитивной рекурсии к функциям I_1^1 и F, вторая из которых получается подстановкой базовой функции I_3^3 в базовую функцию S:

$$\begin{cases} \Sigma(x,\ 0) = I_1^1(x), \\ \Sigma(x,\ y+1) = F\big(x,\ y,\ \Sigma(x,\ y)\big), \end{cases}$$

где $F(x,\ y,\ z) = S\big(I_3^3(x,\ y,\ z)\big) = z + 1.$

Пример

Функция сложения двух натуральных чисел

$$\Sigma(x,\ y) = x + y$$

может быть рассмотрена в качестве примитивно рекурсивной функции двух переменных, получаемой в результате применения оператора примитивной рекурсии к функциям I_1^1 и F, вторая из которых получается подстановкой базовой функции I_3^3 в базовую функцию S:

$$\begin{cases} \Sigma(x,\ 0) = I_1^1(x), \\ \Sigma(x,\ y+1) = F\big(x,\ y,\ \Sigma(x,\ y)\big), \end{cases}$$

где $F(x,\ y,\ z) = S\big(I_3^3(x,\ y,\ z)\big) = z + 1.$

Рассмотрим процесс вычисления функции $\Sigma(x,\ y)$ при $x = 2$, $y = 5$.

Т. к. $\Sigma(2,\ 0) = 2$ из первого равенства схемы примитивной рекурсии, то из второго равенства последовательно находим:

$$\begin{cases} \Sigma(2,\ 1) = F(2,\ 0,\ 2) = 2 + 1 = 3, \\ \Sigma(2,\ 2) = F(2,\ 1,\ 3) = 3 + 1 = 4, \\ \Sigma(2,\ 3) = F(2,\ 2,\ 4) = 4 + 1 = 5, \\ \Sigma(2,\ 4) = F(2,\ 3,\ 5) = 5 + 1 = 6, \\ \Sigma(2,\ 5) = F(2,\ 4,\ 6) = 6 + 1 = 7. \end{cases}$$

Рассмотрим процесс вычисления функции $\Sigma(x,\ y)$ при $x = 2$, $y = 5$.
Т. к. $\Sigma(2,\ 0) = 2$ из первого равенства схемы примитивной рекурсии, то из второго равенства последовательно находим:

$$\begin{cases} \Sigma(2,\ 1) = F(2,\ 0,\ 2) = 2 + 1 = 3, \\ \Sigma(2,\ 2) = F(2,\ 1,\ 3) = 3 + 1 = 4, \\ \Sigma(2,\ 3) = F(2,\ 2,\ 4) = 4 + 1 = 5, \\ \Sigma(2,\ 4) = F(2,\ 3,\ 5) = 5 + 1 = 6, \\ \Sigma(2,\ 5) = F(2,\ 4,\ 6) = 6 + 1 = 7. \end{cases}$$

Рассмотрим процесс вычисления функции $\Sigma(x,\ y)$ при $x = 2$, $y = 5$.
Т. к. $\Sigma(2,\ 0) = 2$ из первого равенства схемы примитивной рекурсии, то из второго равенства последовательно находим:

$$\begin{cases} \Sigma(2,\ 1) = F(2,\ 0,\ 2) = 2 + 1 = 3, \\ \Sigma(2,\ 2) = F(2,\ 1,\ 3) = 3 + 1 = 4, \\ \Sigma(2,\ 3) = F(2,\ 2,\ 4) = 4 + 1 = 5, \\ \Sigma(2,\ 4) = F(2,\ 3,\ 5) = 5 + 1 = 6, \\ \Sigma(2,\ 5) = F(2,\ 4,\ 6) = 6 + 1 = 7. \end{cases}$$

Частично рекурсивные функции

Частично рекурсивные функции — это рекурсивные функции, определённые на части множества возможных аргументов.

Они определяются аналогично примитивно рекурсивным, только к двум операторам подстановки и примитивной рекурсии добавляется ещё оператор минимизации аргумента.

Пусть задана некоторая функция $f(x_1, \ldots, x_n, y)$.
Зафиксируем значения $x_1, \ldots, x_n$ и выясним, при каком y функция $f(x_1, \ldots, x_n, y) = 0$.
Более сложной задачей является отыскание для функции $f(x_1, \ldots, x_n, y)$ и фиксированных $x_1, \ldots, x_n$ наименьшего из тех значений y, при которых функция $f(x_1, \ldots, x_n, y) = 0$.
Для этого вводится оператор минимизации аргумента.

Частично рекурсивные функции

Частично рекурсивные функции — это рекурсивные функции, определённые на части множества возможных аргументов.

Они определяются аналогично примитивно рекурсивным, только к двум операторам подстановки и примитивной рекурсии добавляется ещё оператор минимизации аргумента.

Пусть задана некоторая функция $f(x_1, \ldots, x_n, y)$.
Зафиксируем значения $x_1, \ldots, x_n$ и выясним, при каком y функция $f(x_1, \ldots, x_n, y) = 0$.
Более сложной задачей является отыскание для функции $f(x_1, \ldots, x_n, y)$ и фиксированных $x_1, \ldots, x_n$ наименьшего из тех значений y, при которых функция $f(x_1, \ldots, x_n, y) = 0$.
Для этого вводится оператор минимизации аргумента.

Частично рекурсивные функции

Частично рекурсивные функции — это рекурсивные функции, определённые на части множества возможных аргументов.

Они определяются аналогично примитивно рекурсивным, только к двум операторам подстановки и примитивной рекурсии добавляется ещё оператор минимизации аргумента.

Пусть задана некоторая функция $f(x_1, \ldots, x_n, y)$.
Зафиксируем значения $x_1, \ldots, x_n$ и выясним, при каком y функция $f(x_1, \ldots, x_n, y) = 0$.
Более сложной задачей является отыскание для функции $f(x_1, \ldots, x_n, y)$ и фиксированных $x_1, \ldots, x_n$ наименьшего из тех значений y, при которых функция $f(x_1, \ldots, x_n, y) = 0$.
Для этого вводится оператор минимизации аргумента.

Частично рекурсивные функции

Частично рекурсивные функции — это рекурсивные функции, определённые на части множества возможных аргументов.

Они определяются аналогично примитивно рекурсивным, только к двум операторам подстановки и примитивной рекурсии добавляется ещё оператор минимизации аргумента.

Пусть задана некоторая функция $f(x_1, \ldots, x_n, y)$.
Зафиксируем значения $x_1, \ldots, x_n$ и выясним, при каком y функция $f(x_1, \ldots, x_n, y) = 0$.
Более сложной задачей является отыскание для функции $f(x_1, \ldots, x_n, y)$ и фиксированных $x_1, \ldots, x_n$ наименьшего из тех значений y, при которых функция $f(x_1, \ldots, x_n, y) = 0$.
Для этого вводится оператор минимизации аргумента.

Частично рекурсивные функции

Частично рекурсивные функции — это рекурсивные функции, определённые на части множества возможных аргументов.

Они определяются аналогично примитивно рекурсивным, только к двум операторам подстановки и примитивной рекурсии добавляется ещё оператор минимизации аргумента.

Пусть задана некоторая функция $f(x_1, \ldots, x_n, y)$.
Зафиксируем значения $x_1, \ldots, x_n$ и выясним, при каком y функция $f(x_1, \ldots, x_n, y) = 0$.
Более сложной задачей является отыскание для функции $f(x_1, \ldots, x_n, y)$ и фиксированных $x_1, \ldots, x_n$ наименьшего из тех значений y, при которых функция $f(x_1, \ldots, x_n, y) = 0$.
Для этого вводится оператор минимизации аргумента.

Частично рекурсивные функции

Частично рекурсивные функции — это рекурсивные функции, определённые на части множества возможных аргументов.

Они определяются аналогично примитивно рекурсивным, только к двум операторам подстановки и примитивной рекурсии добавляется ещё оператор минимизации аргумента.

Пусть задана некоторая функция $f(x_1, \ldots, x_n, y)$.
Зафиксируем значения $x_1, \ldots, x_n$ и выясним, при каком y функция $f(x_1, \ldots, x_n, y) = 0$.
Более сложной задачей является отыскание для функции $f(x_1, \ldots, x_n, y)$ и фиксированных $x_1, \ldots, x_n$ наименьшего из тех значений y, при которых функция $f(x_1, \ldots, x_n, y) = 0$.
Для этого вводится оператор минимизации аргумента.

Оператор минимизации аргумента

Пусть f — функция от $n+1$ натуральной переменной.
Тогда результатом применения оператора минимизации аргумента к функции f называется функция h от n переменных, задаваемая следующим определением:

$$h(x_1, \ldots, x_n) = \mu_y \left[f(x_1, \ldots, x_n, y) = 0 \right] = \\ = \min \left\{ y \mid f(x_1, \ldots, x_n, y) = 0 \right\},$$

т. е. функция h возвращает минимальное значение последнего аргумента функции f, при котором значение f равно 0.

Запись вида $\mu_y \left[f(x, y) = 0 \right]$ читается так:
«наименьшее y, такое, что $f(x, y)$ принимает значение 0»

Оператор минимизации аргумента

Пусть f — функция от $n+1$ натуральной переменной.
Тогда результатом применения оператора минимизации аргумента к функции f называется функция h от n переменных, задаваемая следующим определением:

$$h(x_1, \ldots, x_n) = \mu_y \left[f(x_1, \ldots, x_n, y) = 0 \right] = \\ = \min \left\{ y \mid f(x_1, \ldots, x_n, y) = 0 \right\},$$

т. е. функция h возвращает минимальное значение последнего аргумента функции f, при котором значение f равно 0.

Запись вида $\mu_y \left[f(x, y) = 0 \right]$ читается так:
«наименьшее y, такое, что $f(x, y)$ принимает значение 0»

Оператор минимизации аргумента

Пусть f — функция от $n+1$ натуральной переменной.
Тогда результатом применения оператора минимизации аргумента к функции f называется функция h от n переменных, задаваемая следующим определением:

$$h(x_1, \ldots, x_n) = \mu_y \big[f(x_1, \ldots, x_n, y) = 0 \big] = \\ = \min \big\{ y \;\big|\; f(x_1, \ldots, x_n, y) = 0 \big\},$$

т. е. функция h возвращает минимальное значение последнего аргумента функции f, при котором значение f равно 0.

Запись вида $\mu_y \big[f(x, y) = 0 \big]$ читается так:
«наименьшее y, такое, что $f(x, y)$ принимает значение 0»

Оператор минимизации аргумента

Пусть f — функция от $n+1$ натуральной переменной.
Тогда результатом применения оператора минимизации аргумента к функции f называется функция h от n переменных, задаваемая следующим определением:

$$h(x_1, \ldots, x_n) = \mu_y \big[f(x_1, \ldots, x_n, y) = 0 \big] = \\ = \min \big\{ y \;\big|\; f(x_1, \ldots, x_n, y) = 0 \big\},$$

т. е. функция h возвращает минимальное значение последнего аргумента функции f, при котором значение f равно 0.

Запись вида $\mu_y \big[f(x, y) = 0 \big]$ читается так:
«наименьшее y, такое, что $f(x, y)$ принимает значение 0»

Оператор минимизации аргумента

Пусть f — функция от $n+1$ натуральной переменной.
Тогда результатом применения оператора минимизации аргумента к функции f называется функция h от n переменных, задаваемая следующим определением:

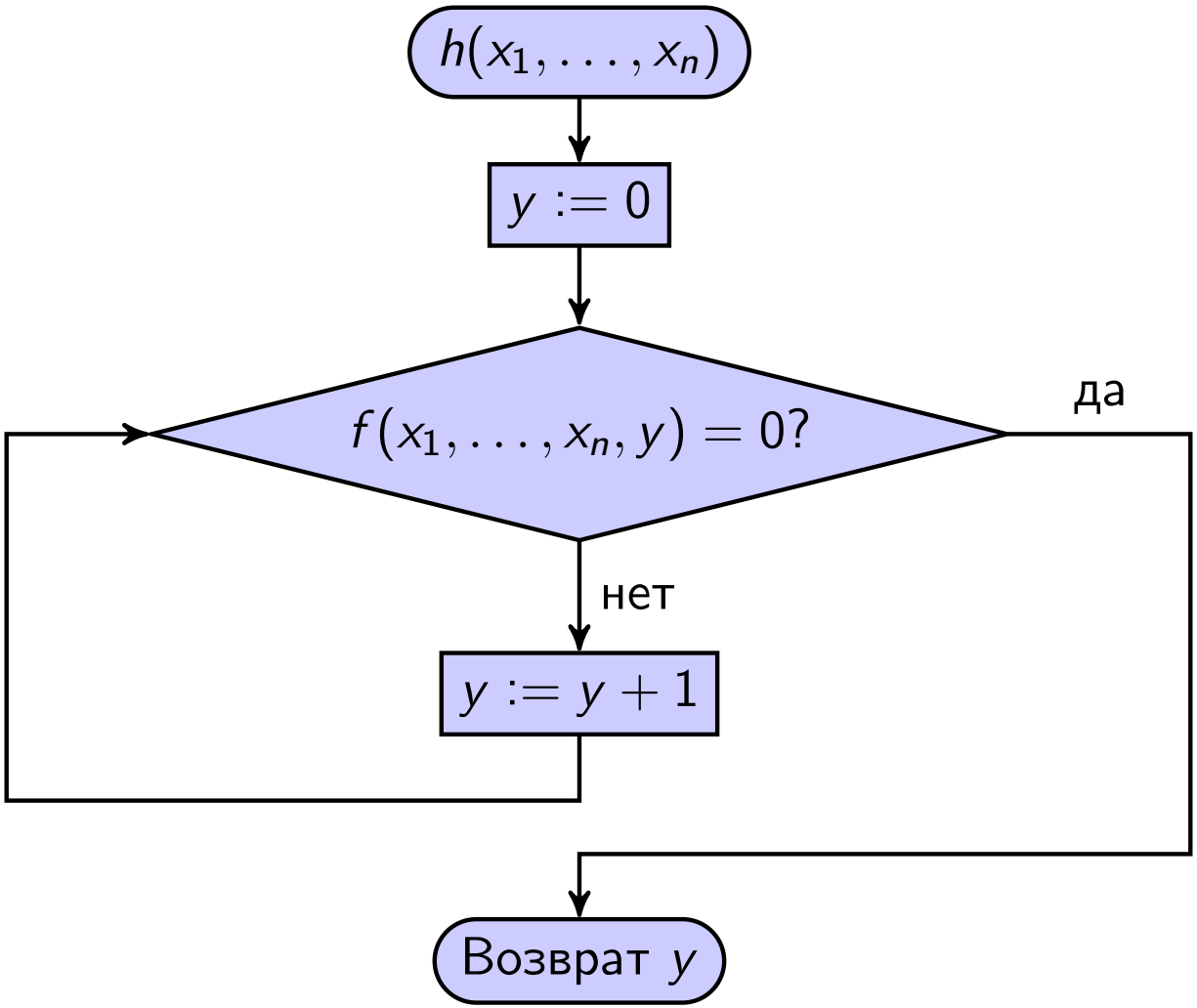
$$h(x_1, \ldots, x_n) = \mu_y \big[f(x_1, \ldots, x_n, y) = 0 \big] = \\ = \min \big\{ y \;\big|\; f(x_1, \ldots, x_n, y) = 0 \big\},$$

т. е. функция h возвращает минимальное значение последнего аргумента функции f, при котором значение f равно 0.

Запись вида $\mu_y \big[f(x, y) = 0 \big]$ читается так:
«наименьшее y, такое, что $f(x, y)$ принимает значение 0»

Для вычисления функции h можно предложить следующий переборный алгоритм:

Для вычисления функции h можно предложить следующий переборный алгоритм:



Аналогия с императивными языками программирования

Примитивно рекурсивные функции соответствуют программным функциям, в которых используется только арифметические операции, а также условный оператор и оператор цикла с фиксированным количеством шагов (цикл `for`).

Если также использовать оператор цикла `while`, в котором число итераций заранее неизвестно, и в принципе, может быть бесконечным, то мы переходим в класс частично рекурсивных функций.

Аналогия с императивными языками программирования

Примитивно рекурсивные функции соответствуют программным функциям, в которых используется только арифметические операции, а также условный оператор и оператор цикла с фиксированным количеством шагов (цикл `for`).

Если также использовать оператор цикла `while`, в котором число итераций заранее неизвестно, и в принципе, может быть бесконечным, то мы переходим в класс частично рекурсивных функций.

Особенность частично рекурсивных функций

Частично рекурсивные функции для некоторых значений аргумента могут быть не определены, так как оператор минимизации аргумента не всегда корректно определён, а именно, функция f может быть не равной нулю ни при каких значениях аргументов.

С точки зрения программиста результатом частично рекурсивной функции может быть не только число, но и исключение (exception) или «зависание», соответствующее неопределённому значению.

Особенность частично рекурсивных функций

Частично рекурсивные функции для некоторых значений аргумента могут быть не определены, так как оператор минимизации аргумента не всегда корректно определён, а именно, функция f может быть не равной нулю ни при каких значениях аргументов.

С точки зрения программиста результатом частично рекурсивной функции может быть не только число, но и исключение (exception) или «зависание», соответствующее неопределённому значению.

Пример

Частичная функция вычитания двух натуральных чисел

$$M(x,\ y) = x - y$$

получается с помощью применения оператора минимизации:

$$M(x,\ y) = \mu_z\big[f(x,\ y,\ z) = 0\big],$$

где $f(x,\ y,\ z) = \big|x - (y + z)\big|$.

Докажите самостоятельно, что функция $\Delta(x, y) = |x - y|$ (симметрическая разность) является примитивно рекурсивной.

Очевидно, что функция M неприменима, когда $x < y$.

Пример

Частичная функция вычитания двух натуральных чисел

$$M(x,\ y) = x - y$$

получается с помощью применения оператора минимизации:

$$M(x,\ y) = \mu_z \big[f(x,\ y,\ z) = 0 \big],$$

где $f(x,\ y,\ z) = \big| x - (y + z) \big|$.

Докажите самостоятельно, что функция $\Delta(x, y) = |x - y|$ (симметрическая разность) является примитивно рекурсивной.

Очевидно, что функция M неприменима, когда $x < y$.

Пример

Частичная функция вычитания двух натуральных чисел

$$M(x,\ y) = x - y$$

получается с помощью применения оператора минимизации:

$$M(x,\ y) = \mu_z \big[f(x,\ y,\ z) = 0 \big],$$

где $f(x,\ y,\ z) = \big| x - (y + z) \big|$.

Докажите самостоятельно, что функция $\Delta(x, y) = |x - y|$ (симметрическая разность) является примитивно рекурсивной.

Очевидно, что функция M неприменима, когда $x < y$.

Пример

Частичная функция вычитания двух натуральных чисел

$$M(x,\ y) = x - y$$

получается с помощью применения оператора минимизации:

$$M(x,\ y) = \mu_z \big[f(x,\ y,\ z) = 0 \big],$$

где $f(x,\ y,\ z) = \big| x - (y + z) \big|$.

Докажите самостоятельно, что функция $\Delta(x, y) = |x - y|$ (симметрическая разность) является примитивно рекурсивной.

Очевидно, что функция M неприменима, когда $x < y$.

Тезис Чёрча

Тезис Чёрча

Каждая интуитивно вычислимая функция является частично рекурсивной.

Этот тезис нельзя строго доказать, так как он связывает нестрогое математическое понятие «интуитивно вычислимая функция» со строгим математическим понятием «частично рекурсивная функция».

Но тезис может быть опровергнут, если построить пример функции интуитивно вычислимой, но не являющейся частично рекурсивной.

Пока таких примеров построено не было.

Тезис Чёрча является достаточным средством, позволяющим придать необходимую точность формулировкам алгоритмических проблем, он делает возможным доказательство их неразрешимости.

Тезис Чёрча

Тезис Чёрча

Каждая интуитивно вычислимая функция является частично рекурсивной.

Этот тезис нельзя строго доказать, так как он связывает нестрогое математическое понятие «интуитивно вычислимая функция» со строгим математическим понятием «частично рекурсивная функция».

Но тезис может быть опровергнут, если построить пример функции интуитивно вычислимой, но не являющейся частично рекурсивной.

Пока таких примеров построено не было.

Тезис Чёрча является достаточным средством, позволяющим придать необходимую точность формулировкам алгоритмических проблем, он делает возможным доказательство их неразрешимости.

Тезис Чёрча

Тезис Чёрча

Каждая интуитивно вычислимая функция является частично рекурсивной.

Этот тезис нельзя строго доказать, так как он связывает нестрогое математическое понятие «интуитивно вычислимая функция» со строгим математическим понятием «частично рекурсивная функция».
Но тезис может быть опровергнут, если построить пример функции интуитивно вычислимой, но не являющейся частично рекурсивной.
Пока таких примеров построено не было.
Тезис Чёрча является достаточным средством, позволяющим придать необходимую точность формулировкам алгоритмических проблем, он делает возможным доказательство их неразрешимости.

Тезис Чёрча

Тезис Чёрча

Каждая интуитивно вычислимая функция является частично рекурсивной.

Этот тезис нельзя строго доказать, так как он связывает нестрогое математическое понятие «интуитивно вычислимая функция» со строгим математическим понятием «частично рекурсивная функция».

Но тезис может быть опровергнут, если построить пример функции интуитивно вычислимой, но не являющейся частично рекурсивной.

Пока таких примеров построено не было.

Тезис Чёрча является достаточным средством, позволяющим придать необходимую точность формулировкам алгоритмических проблем, он делает возможным доказательство их неразрешимости.

Тезис Чёрча

Тезис Чёрча

Каждая интуитивно вычислимая функция является частично рекурсивной.

Этот тезис нельзя строго доказать, так как он связывает нестрогое математическое понятие «интуитивно вычислимая функция» со строгим математическим понятием «частично рекурсивная функция».
Но тезис может быть опровергнут, если построить пример функции интуитивно вычислимой, но не являющейся частично рекурсивной.
Пока таких примеров построено не было.
Тезис Чёрча является достаточным средством, позволяющим придать необходимую точность формулировкам алгоритмических проблем, он делает возможным доказательство их неразрешимости.

Общерекурсивные функции

Общерекурсивные функции — это подмножество частично рекурсивных функций, определённых для всех значений аргументов.

К сожалению, задача определения того, является ли частично рекурсивная функция с данным описанием общерекурсивной или нет, алгоритмически неразрешима.

Легко понять, что любая примитивно рекурсивная функция является частично рекурсивной, так как по определению операторы для построения частично рекурсивных функций включают в себя операторы для построения примитивно рекурсивных функций.

Также понятно, что примитивно рекурсивная функция определена везде и поэтому является общерекурсивной функцией (у примитивно рекурсивной функции нет повода «зависать», так как при её построении используются операторы, определяющие везде определённые функции).

Общерекурсивные функции

Общерекурсивные функции — это подмножество частично рекурсивных функций, определённых для всех значений аргументов.

К сожалению, задача определения того, является ли частично рекурсивная функция с данным описанием общерекурсивной или нет, алгоритмически неразрешима.

Легко понять, что любая примитивно рекурсивная функция является частично рекурсивной, так как по определению операторы для построения частично рекурсивных функций включают в себя операторы для построения примитивно рекурсивных функций.

Также понятно, что примитивно рекурсивная функция определена везде и поэтому является общерекурсивной функцией (у примитивно рекурсивной функции нет повода «зависать», так как при её построении используются операторы, определяющие везде определённые функции).

Общерекурсивные функции

Общерекурсивные функции — это подмножество частично рекурсивных функций, определённых для всех значений аргументов.

К сожалению, задача определения того, является ли частично рекурсивная функция с данным описанием общерекурсивной или нет, алгоритмически неразрешима.

Легко понять, что любая примитивно рекурсивная функция является частично рекурсивной, так как по определению операторы для построения частично рекурсивных функций включают в себя операторы для построения примитивно рекурсивных функций.

Также понятно, что примитивно рекурсивная функция определена везде и поэтому является общерекурсивной функцией (у примитивно рекурсивной функции нет повода «зависать», так как при её построении используются операторы, определяющие везде определённые функции).

Общерекурсивные функции

Общерекурсивные функции — это подмножество частично рекурсивных функций, определённых для всех значений аргументов.

К сожалению, задача определения того, является ли частично рекурсивная функция с данным описанием общерекурсивной или нет, алгоритмически неразрешима.

Легко понять, что любая примитивно рекурсивная функция является частично рекурсивной, так как по определению операторы для построения частично рекурсивных функций включают в себя операторы для построения примитивно рекурсивных функций.

Также понятно, что примитивно рекурсивная функция определена везде и поэтому является общерекурсивной функцией (у примитивно рекурсивной функции нет повода «зависать», так как при её построении используются операторы, определяющие везде определённые функции).

Пример ОРФ, не являющейся ПРФ

Довольно сложно доказать, что $\mathscr{F}_{\text{ОРФ}} \neq \mathscr{F}_{\text{ПРФ}}$.

Для этого нужно привести пример функции, которая является ОРФ и не является ЧРФ.

Такая функция была предложена Аккерманом в 1926 г.

Идея состоит в том, чтобы построить такую вычислимую функцию, которая растёт быстрее любой примитивно рекурсивной и поэтому примитивно рекурсивной функцией не является.

Пример ОРФ, не являющейся ПРФ

Довольно сложно доказать, что $\mathscr{F}_{\text{ОРФ}} \neq \mathscr{F}_{\text{ПРФ}}$.
Для этого нужно привести пример функции, которая является ОРФ и не является ЧРФ.

Такая функция была предложена Аккерманом в 1926 г.

Идея состоит в том, чтобы построить такую вычислимую функцию, которая растёт быстрее любой примитивно рекурсивной и поэтому примитивно рекурсивной функцией не является.

Пример ОРФ, не являющейся ПРФ

Довольно сложно доказать, что $\mathscr{F}_{\text{ОРФ}} \neq \mathscr{F}_{\text{ПРФ}}$.
Для этого нужно привести пример функции, которая является ОРФ и не является ЧРФ.
Такая функция была предложена Аккерманом в 1926 г.

Идея состоит в том, чтобы построить такую вычислимую функцию, которая растёт быстрее любой примитивно рекурсивной и поэтому примитивно рекурсивной функцией не является.

Пример ОРФ, не являющейся ПРФ

Довольно сложно доказать, что $\mathscr{F}_{\text{ОРФ}} \neq \mathscr{F}_{\text{ПРФ}}$.
Для этого нужно привести пример функции, которая является ОРФ и не является ЧРФ.
Такая функция была предложена Аккерманом в 1926 г.

Идея состоит в том, чтобы построить такую вычислимую функцию, которая растёт быстрее любой примитивно рекурсивной и поэтому примитивно рекурсивной функцией не является.

Функция Аккермана

Вильгельм Фридрих Áккерман (1896—1962) — немецкий математик.

Функция Аккермана определяется рекурсивно для неотрицательных целых чисел m и n следующим образом:

$$A(m,\ n) = \begin{cases} n+1, & \text{если } m = 0; \\ A(m-1,\ 1), & \text{если } m > 0 \text{ и } n = 0; \\ A\big(m-1,\ A(m,\ n-1)\big), & \text{если } m > 0 \text{ и } n > 0. \end{cases}$$

Эта функция растёт очень быстро, например, число $A(4,\ 4)$ настолько велико, что количество цифр в порядке этого числа многократно превосходит количество атомов в наблюдаемой части вселенной.

Функция Аккермана

Вильгельм Фридрих Áккерман (1896—1962) — немецкий математик.

Функция Аккермана определяется рекурсивно для неотрицательных целых чисел m и n следующим образом:

$$A(m,\ n) = \\ = \begin{cases} n+1, & \text{если } m = 0; \\ A(m-1,\ 1), & \text{если } m > 0 \text{ и } n = 0; \\ A\big(m-1,\ A(m,\ n-1)\big), & \text{если } m > 0 \text{ и } n > 0. \end{cases}$$

Эта функция растёт очень быстро, например, число $A(4,\ 4)$ настолько велико, что количество цифр в порядке этого числа многократно превосходит количество атомов в наблюдаемой части вселенной.

Функция Аккермана

Вильгельм Фридрих Áккерман (1896—1962) — немецкий математик.

Функция Аккермана определяется рекурсивно для неотрицательных целых чисел m и n следующим образом:

$$A(m,\ n) = \begin{cases} n+1, & \text{если } m = 0; \\ A(m-1,\ 1), & \text{если } m > 0 \text{ и } n = 0; \\ A\big(m-1,\ A(m,\ n-1)\big), & \text{если } m > 0 \text{ и } n > 0. \end{cases}$$

Эта функция растёт очень быстро, например, число $A(4,\ 4)$ настолько велико, что количество цифр в порядке этого числа многократно превосходит количество атомов в наблюдаемой части вселенной.

Анализ функции Аккермана

Может показаться не очевидным, что рекурсия всегда заканчивается.

Это следует из того, что при рекурсивном вызове или уменьшается значение m, или значение m сохраняется, но уменьшается значение n.

Это означает, что каждый раз пара $(m,\ n)$ уменьшается с точки зрения лексикографического порядка, значит, значение m в итоге достигнет нуля: для одного значения m существует конечное число возможных значений n (так как первый вызов с данным m был произведён с каким-то определённым значением n, а в дальнейшем, если значение m сохраняется, значение n может только уменьшаться), а количество возможных значений m, в свою очередь, тоже конечно.

Однако, при уменьшении m значение, на которое увеличивается n, не ограничено и обычно очень велико.

Анализ функции Аккермана

Может показаться не очевидным, что рекурсия всегда заканчивается.
Это следует из того, что при рекурсивном вызове или уменьшается значение m, или значение m сохраняется, но уменьшается значение n.
Это означает, что каждый раз пара $(m,\ n)$ уменьшается с точки зрения лексикографического порядка, значит, значение m в итоге достигнет нуля: для одного значения m существует конечное число возможных значений n (так как первый вызов с данным m был произведён с каким-то определённым значением n, а в дальнейшем, если значение m сохраняется, значение n может только уменьшаться), а количество возможных значений m, в свою очередь, тоже конечно.
Однако, при уменьшении m значение, на которое увеличивается n, не ограничено и обычно очень велико.

Анализ функции Аккермана

Может показаться не очевидным, что рекурсия всегда заканчивается.
Это следует из того, что при рекурсивном вызове или уменьшается значение m, или значение m сохраняется, но уменьшается значение n.
Это означает, что каждый раз пара $(m,\ n)$ уменьшается с точки зрения лексикографического порядка, значит, значение m в итоге достигнет нуля: для одного значения m существует конечное число возможных значений n (так как первый вызов с данным m был произведён с каким-то определённым значением n, а в дальнейшем, если значение m сохраняется, значение n может только уменьшаться), а количество возможных значений m, в свою очередь, тоже конечно.
Однако, при уменьшении m значение, на которое увеличивается n, не ограничено и обычно очень велико.

Анализ функции Аккермана

Может показаться не очевидным, что рекурсия всегда заканчивается.
Это следует из того, что при рекурсивном вызове или уменьшается значение m, или значение m сохраняется, но уменьшается значение n.
Это означает, что каждый раз пара $(m,\ n)$ уменьшается с точки зрения лексикографического порядка, значит, значение m в итоге достигнет нуля: для одного значения m существует конечное число возможных значений n (так как первый вызов с данным m был произведён с каким-то определённым значением n, а в дальнейшем, если значение m сохраняется, значение n может только уменьшаться), а количество возможных значений m, в свою очередь, тоже конечно.
Однако, при уменьшении m значение, на которое увеличивается n, не ограничено и обычно очень велико.

Пример вычисления $A(1, 2)$

Вычислим значение $A(1, 2)$:

$$\begin{aligned} A(1, 2) &= A\big(0, A(1, 1)\big) = \\ &= A\Big(0, A\big(0, A(1, 0)\big)\Big) = \\ &= A\Big(0, A\big(0, A(0, 1)\big)\Big) = \\ &= A\big(0, A(0, 2)\big) = \\ &= A(0, 3) = \\ &= 4. \end{aligned}$$

Пример вычисления $A(1, 2)$

Вычислим значение $A(1, 2)$:

$$\begin{aligned} A(1, 2) &= A\big(0, A(1, 1)\big) = \\ &= A\Big(0, A\big(0, A(1, 0)\big)\Big) = \\ &= A\Big(0, A\big(0, A(0, 1)\big)\Big) = \\ &= A\big(0, A(0, 2)\big) = \\ &= A(0, 3) = \\ &= 4. \end{aligned}$$

Таблица значений функции Аккермана

m	$A(m,\ 0)$	$A(m,\ 1)$	$A(m,\ 2)$	$A(m,\ 3)$	$A(m,\ 4)$	$A(m,\ n)$
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$2+(n+3)-3$
2	3	5	7	9	11	$2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{n+3}-3$
4	13	65 533	$2^{65\,536}-3$	$2^{2^{65\,536}}-3$	$2^{2^{2^{65\,536}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3}-3$

Нормальные алгоритмы Маркова

Нормальный алгоритм Маркова (НАМ) — один из стандартизованных вариантов представления об алгоритме.

Понятие нормального алгоритма введено А. А. Марковым в конце 1940-х годов.

Нормальные алгоритмы являются вербальными, т. е. они применяются к словам в различных алфавитах.

Определение всякого нормального алгоритма состоит из двух частей:

1. определения алфавита алгоритма (к словам которого алгоритм будет применяться),
2. определения его схемы.

Нормальные алгоритмы Маркова

Нормальный алгоритм Маркова (НАМ) — один из стандартизованных вариантов представления об алгоритме.

Понятие нормального алгоритма введено А. А. Марковым в конце 1940-х годов.

Нормальные алгоритмы являются вербальными, т. е. они применяются к словам в различных алфавитах.

Определение всякого нормального алгоритма состоит из двух частей:

1. определения алфавита алгоритма (к словам которого алгоритм будет применяться),
2. определения его схемы.

Нормальные алгоритмы Маркова

Нормальный алгоритм Маркова (НАМ) — один из стандартизованных вариантов представления об алгоритме.

Понятие нормального алгоритма введено А. А. Марковым в конце 1940-х годов.

Нормальные алгоритмы являются вербальными, т. е. они применяются к словам в различных алфавитах.

Определение всякого нормального алгоритма состоит из двух частей:

1. определения алфавита алгоритма (к словам которого алгоритм будет применяться),
2. определения его схемы.

Нормальные алгоритмы Маркова

Нормальный алгоритм Маркова (НАМ) — один из стандартизованных вариантов представления об алгоритме.

Понятие нормального алгоритма введено А. А. Марковым в конце 1940-х годов.

Нормальные алгоритмы являются вербальными, т. е. они применяются к словам в различных алфавитах.

Определение всякого нормального алгоритма состоит из двух частей:

1. определения алфавита алгоритма (к словам которого алгоритм будет применяться),
2. определения его схемы.

Нормальные алгоритмы Маркова

Нормальный алгоритм Маркова (НАМ) — один из стандартизованных вариантов представления об алгоритме.

Понятие нормального алгоритма введено А. А. Марковым в конце 1940-х годов.

Нормальные алгоритмы являются вербальными, т. е. они применяются к словам в различных алфавитах.

Определение всякого нормального алгоритма состоит из двух частей:

1. определения алфавита алгоритма (к словам которого алгоритм будет применяться),
2. определения его схемы.

Определение нормального алгоритма

Схемой нормального алгоритма называется конечный упорядоченный набор формул подстановки, каждая из которых может быть простой или заключительной.

Простыми формулами подстановки называются слова вида $L \to D$, где L и D — два произвольных слова в алфавите алгоритма (называемые, соответственно, левой и правой частями формулы подстановки).

Заключительными формулами подстановки называются слова вида $L \to \cdot D$, где L и D — два произвольных слова в алфавите алгоритма.

При этом предполагается, что вспомогательные буквы $\to$ и $\to\cdot$ не принадлежат алфавиту алгоритма.

Определение нормального алгоритма

Схемой нормального алгоритма называется конечный упорядоченный набор формул подстановки, каждая из которых может быть простой или заключительной.

Простыми формулами подстановки называются слова вида $L \to D$, где L и D — два произвольных слова в алфавите алгоритма (называемые, соответственно, левой и правой частями формулы подстановки).

Заключительными формулами подстановки называются слова вида $L \to \cdot D$, где L и D — два произвольных слова в алфавите алгоритма.

При этом предполагается, что вспомогательные буквы $\to$ и $\to\cdot$ не принадлежат алфавиту алгоритма.

Определение нормального алгоритма

Схемой нормального алгоритма называется конечный упорядоченный набор формул подстановки, каждая из которых может быть простой или заключительной.

Простыми формулами подстановки называются слова вида $L \to D$, где L и D — два произвольных слова в алфавите алгоритма (называемые, соответственно, левой и правой частями формулы подстановки).

Заключительными формулами подстановки называются слова вида $L \to \cdot D$, где L и D — два произвольных слова в алфавите алгоритма.

При этом предполагается, что вспомогательные буквы $\to$ и $\to\cdot$ не принадлежат алфавиту алгоритма.

Определение нормального алгоритма

Схемой нормального алгоритма называется конечный упорядоченный набор формул подстановки, каждая из которых может быть простой или заключительной.

Простыми формулами подстановки называются слова вида $L \to D$, где L и D — два произвольных слова в алфавите алгоритма (называемые, соответственно, левой и правой частями формулы подстановки).

Заключительными формулами подстановки называются слова вида $L \to \cdot D$, где L и D — два произвольных слова в алфавите алгоритма.

При этом предполагается, что вспомогательные буквы $\to$ и $\to\cdot$ не принадлежат алфавиту алгоритма.

Пример определения НАМ

Примером схемы нормального алгоритма в пятибуквенном алфавите $\{\,|\,, *, a, b, c\,\}$ может служить схема

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

Пример определения НАМ

Примером схемы нормального алгоритма в пятибуквенном алфавите $\{\,|\,, *, a, b, c\,\}$ может служить схема

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

Процесс применения НАМ

Процесс применения нормального алгоритма к произвольному слову V в алфавите этого алгоритма представляет собой дискретную последовательность элементарных шагов, состоящих в следующем.

Пусть V' — слово, полученное на предыдущем шаге работы алгоритма (или исходное слово V, если текущий шаг является первым).

Процесс применения НАМ

Процесс применения нормального алгоритма к произвольному слову V в алфавите этого алгоритма представляет собой дискретную последовательность элементарных шагов, состоящих в следующем.

Пусть V' — слово, полученное на предыдущем шаге работы алгоритма (или исходное слово V, если текущий шаг является первым).

1. Если среди формул подстановки нет такой, левая часть которой входила бы в V', то работа алгоритма считается завершённой, и результатом этой работы считается слово V'.
2. Иначе, среди формул подстановки, левая часть которых входит в V', выбирается самая верхняя:
    1. если эта формула подстановки имеет вид $L \to \cdot D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего работа алгоритма считается завершённой с результатом RDS;
    2. если же эта формула подстановки имеет вид $L \to D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего слово RDS считается результатом текущего шага, подлежащим дальнейшей переработке на следующем шаге.

1. Если среди формул подстановки нет такой, левая часть которой входила бы в V', то работа алгоритма считается завершённой, и результатом этой работы считается слово V'.
2. Иначе, среди формул подстановки, левая часть которых входит в V', выбирается самая верхняя:
    1. если эта формула подстановки имеет вид $L \to \cdot D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего работа алгоритма считается завершённой с результатом RDS;
    2. если же эта формула подстановки имеет вид $L \to D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего слово RDS считается результатом текущего шага, подлежащим дальнейшей переработке на следующем шаге.

1. Если среди формул подстановки нет такой, левая часть которой входила бы в V', то работа алгоритма считается завершённой, и результатом этой работы считается слово V'.
2. Иначе, среди формул подстановки, левая часть которых входит в V', выбирается самая верхняя:
   1. если эта формула подстановки имеет вид $L \to \cdot D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего работа алгоритма считается завершённой с результатом RDS;
   2. если же эта формула подстановки имеет вид $L \to D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего слово RDS считается результатом текущего шага, подлежащим дальнейшей переработке на следующем шаге.

1. Если среди формул подстановки нет такой, левая часть которой входила бы в V', то работа алгоритма считается завершённой, и результатом этой работы считается слово V'.
2. Иначе, среди формул подстановки, левая часть которых входит в V', выбирается самая верхняя:
   1. если эта формула подстановки имеет вид $L \to \cdot D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего работа алгоритма считается завершённой с результатом RDS;
   2. если же эта формула подстановки имеет вид $L \to D$, то из всех возможных представлений слова V' в виде RLS выбирается такое, при котором R — самое короткое, после чего слово RDS считается результатом текущего шага, подлежащим дальнейшей переработке на следующем шаге.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *, a, b, c\,\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*|$, $aba|*|$, $baa|*|$, $aa|*|$, $aa|c|$, $aac|$, $ac||$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{|, *, a, b, c\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *, a, b, c\,\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac||$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *, a, b, c\,\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *\,, a\,, b\,, c\,\}$ и схемой

$$\begin{cases} |b & \to & ba| \\ ab & \to & ba \\ b & \to & \\ *| & \to & b* \\ * & \to & c \\ |c & \to & c \\ ac & \to & c| \\ c & \to \cdot & \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac||$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *, a, b, c\,\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{|, *, a, b, c\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *\,, a\,, b\,, c\,\}$ и схемой

$$\begin{cases} |b & \to & ba| \\ ab & \to & ba \\ b & \to & \\ *| & \to & b* \\ * & \to & c \\ |c & \to & c \\ ac & \to & c| \\ c & \to \cdot & \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{|, *, a, b, c\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *, a, b, c\,\}$ и схемой

$$\begin{cases} |b \to ba| \\ ab \to ba \\ b \to \\ *| \to b* \\ * \to c \\ |c \to c \\ ac \to c| \\ c \to \cdot \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *\,, a\,, b\,, c\,\}$ и схемой

$$\begin{cases} |b & \to & ba| \\ ab & \to & ba \\ b & \to & \\ *| & \to & b* \\ * & \to & c \\ |c & \to & c \\ ac & \to & c| \\ c & \to \cdot & \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *\,, a\,, b\,, c\,\}$ и схемой

$$\begin{cases} |b & \to & ba| \\ ab & \to & ba \\ b & \to & \\ *| & \to & b* \\ * & \to & c \\ |c & \to & c \\ ac & \to & c| \\ c & \to \cdot & \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 1

Например, в ходе процесса применения НАМ с алфавитом $\{\,|\,, *\,, a\,, b\,, c\,\}$ и схемой

$$\begin{cases} |b & \to & ba| \\ ab & \to & ba \\ b & \to & \\ *| & \to & b* \\ * & \to & c \\ |c & \to & c \\ ac & \to & c| \\ c & \to \cdot & \end{cases}$$

к слову $|*||$ последовательно возникают слова $|b*|$, $ba|*|$, $a|*|$, $a|b*$, $aba|*$, $baa|*$, $aa|*$, $aa|c$, aac, $ac|$ и $c||$, после чего алгоритм завершает работу с результатом $||$.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».

Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем 101 в $|||||$:

Алфавит: $\{0, 1, |\}$.

Схема:
$$\begin{cases} |0 \to 0|| \\ 1 \to 0| \\ 0 \to \end{cases}$$

Исходная строка: 101

Выполнение: $101 \to 0|01 \to 00||1 \to 00||0| \to$
$\to 00|0|| \to 000|||| \to 00|||| \to 0||||| \to |||||$.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек. Например, преобразуем 101 в |||||:

Алфавит: {0, 1, |}.

Схема:
$$\begin{cases} |0 \to 0|| \\ 1 \to 0| \\ 0 \to \end{cases}$$

Исходная строка: 101

Выполнение: $101 \to 0|01 \to 00||1 \to 00||0| \to$
$\to 00|0|| \to 000|||| \to 00|||| \to 0||||| \to |||||$.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: {`0`, `1`, `|`}.

Схема:
- `|0` → `0||`
- `1` → `0|`
- `0` →

Исходная строка: `101`

Выполнение: `101` → `0|01` → `00||1` → `00||0|` → → `00|0||` → `000||||` → `00||||` → `0||||` → `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:
$$\begin{cases} |0 \to 0|| \\ 1 \to 0| \\ 0 \to \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} & \to & \texttt{0||} \\ \texttt{1} & \to & \texttt{0|} \\ \texttt{0} & \to & \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} & \to & \texttt{0||} \\ \texttt{1} & \to & \texttt{0|} \\ \texttt{0} & \to & \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$ `00|0||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} & \to & \texttt{0||} \\ \texttt{1} & \to & \texttt{0|} \\ \texttt{0} & \to & \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$ `00|0||` $\to$ `000||||` $\to$ `00||||` $\to$ `0||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} & \to & \texttt{0||} \\ \texttt{1} & \to & \texttt{0|} \\ \texttt{0} & \to & \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} & \to & \texttt{0||} \\ \texttt{1} & \to & \texttt{0|} \\ \texttt{0} & \to & \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0|||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} \to \texttt{0||} \\ \texttt{1} \to \texttt{0|} \\ \texttt{0} \to \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0|||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:
$$\begin{cases} \texttt{|0} \to \texttt{0||} \\ \texttt{1} \to \texttt{0|} \\ \texttt{0} \to \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0|||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:
$$\begin{cases} \texttt{|0} \to \texttt{0||} \\ \texttt{1} \to \texttt{0|} \\ \texttt{0} \to \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0|||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} \to \texttt{0||} \\ \texttt{1} \to \texttt{0|} \\ \texttt{0} \to \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0|||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||` .

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:

$$\begin{cases} \texttt{|0} \to \texttt{0||} \\ \texttt{1} \to \texttt{0|} \\ \texttt{0} \to \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0|||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Пример 2

Рассмотрим НАМ, преобразующий числа, записанные в бинарной системе счисления, в числа, записанные в унарной, где вместо единиц будем использовать чёрточки «|».
Иными словами, если на входе было число N в бинарной системе счисления, то на выходе получается строка из N чёрточек.Например, преобразуем `101` в `|||||`:

Алфавит: $\{$ `0` , `1` , `|` $\}$.

Схема:
$$\begin{cases} \texttt{|0} \to \texttt{0||} \\ \texttt{1} \to \texttt{0|} \\ \texttt{0} \to \end{cases}$$

Исходная строка: `101`

Выполнение: `101` $\to$ `0|01` $\to$ `00||1` $\to$ `00||0|` $\to$
$\to$ `00|0|||` $\to$ `000|||||` $\to$ `00|||||` $\to$ `0|||||` $\to$ `|||||`.

Принцип нормализации

Вариант тезиса Чёрча—Тьюринга, сформулированный применительно к нормальным алгоритмам, принято называть принципом нормализации.

Пусть задан алфавит $\mathscr{A}$. Добавим к нему новые буквы и получим алфавит $\mathscr{A}_1$ с условием $\mathscr{A} \subseteq \mathscr{A}_1$.

Алфавит $\mathscr{A}_1$ называется расширением алфавита $\mathscr{A}$.

Принцип нормализации

Пусть задан произвольный вербальный алгоритм $\mathfrak{A}$ над алфавитом $\mathscr{A}$. Тогда существует расширение $\mathscr{A}_1$ алфавита $\mathscr{A}$ и нормальный алгоритм $\mathfrak{A}_1$ в алфавите $\mathscr{A}_1$ с условием: произвольное слово P в алфавите $\mathscr{A}$ перерабатывается нормальным алгоритмом $\mathfrak{A}_1$ в тот же самый результат, в который слово P перерабатывается исходным вербальным алгоритмом $\mathfrak{A}$.

Любой нормальный алгоритм эквивалентен некоторой машине Тьюринга, и наоборот — любая машина Тьюринга эквивалентна некоторому нормальному алгоритму.

Принцип нормализации

Вариант тезиса Чёрча—Тьюринга, сформулированный применительно к нормальным алгоритмам, принято называть принципом нормализации.

Пусть задан алфавит $\mathscr{A}$. Добавим к нему новые буквы и получим алфавит $\mathscr{A}_1$ с условием $\mathscr{A} \subseteq \mathscr{A}_1$.

Алфавит $\mathscr{A}_1$ называется расширением алфавита $\mathscr{A}$.

Принцип нормализации

Пусть задан произвольный вербальный алгоритм $\mathfrak{A}$ над алфавитом $\mathscr{A}$. Тогда существует расширение $\mathscr{A}_1$ алфавита $\mathscr{A}$ и нормальный алгоритм $\mathfrak{A}_1$ в алфавите $\mathscr{A}_1$ с условием: произвольное слово P в алфавите $\mathscr{A}$ перерабатывается нормальным алгоритмом $\mathfrak{A}_1$ в тот же самый результат, в который слово P перерабатывается исходным вербальным алгоритмом $\mathfrak{A}$.

Любой нормальный алгоритм эквивалентен некоторой машине Тьюринга, и наоборот — любая машина Тьюринга эквивалентна некоторому нормальному алгоритму.

Принцип нормализации

Вариант тезиса Чёрча—Тьюринга, сформулированный применительно к нормальным алгоритмам, принято называть принципом нормализации.

Пусть задан алфавит $\mathscr{A}$. Добавим к нему новые буквы и получим алфавит $\mathscr{A}_1$ с условием $\mathscr{A} \subseteq \mathscr{A}_1$.

Алфавит $\mathscr{A}_1$ называется расширением алфавита $\mathscr{A}$.

Принцип нормализации

Пусть задан произвольный вербальный алгоритм $\mathfrak{A}$ над алфавитом $\mathscr{A}$. Тогда существует расширение $\mathscr{A}_1$ алфавита $\mathscr{A}$ и нормальный алгоритм $\mathfrak{A}_1$ в алфавите $\mathscr{A}_1$ с условием: произвольное слово P в алфавите $\mathscr{A}$ перерабатывается нормальным алгоритмом $\mathfrak{A}_1$ в тот же самый результат, в который слово P перерабатывается исходным вербальным алгоритмом $\mathfrak{A}$.

Любой нормальный алгоритм эквивалентен некоторой машине Тьюринга, и наоборот — любая машина Тьюринга эквивалентна некоторому нормальному алгоритму.

Принцип нормализации

Вариант тезиса Чёрча—Тьюринга, сформулированный применительно к нормальным алгоритмам, принято называть принципом нормализации.

Пусть задан алфавит $\mathscr{A}$. Добавим к нему новые буквы и получим алфавит $\mathscr{A}_1$ с условием $\mathscr{A} \subseteq \mathscr{A}_1$.

Алфавит $\mathscr{A}_1$ называется расширением алфавита $\mathscr{A}$.

Принцип нормализации

Пусть задан произвольный вербальный алгоритм $\mathfrak{A}$ над алфавитом $\mathscr{A}$. Тогда существует расширение $\mathscr{A}_1$ алфавита $\mathscr{A}$ и нормальный алгоритм $\mathfrak{A}_1$ в алфавите $\mathscr{A}_1$ с условием: произвольное слово P в алфавите $\mathscr{A}$ перерабатывается нормальным алгоритмом $\mathfrak{A}_1$ в тот же самый результат, в который слово P перерабатывается исходным вербальным алгоритмом $\mathfrak{A}$.

Любой нормальный алгоритм эквивалентен некоторой машине Тьюринга, и наоборот — любая машина Тьюринга эквивалентна некоторому нормальному алгоритму.

Принцип нормализации

Вариант тезиса Чёрча—Тьюринга, сформулированный применительно к нормальным алгоритмам, принято называть принципом нормализации.

Пусть задан алфавит $\mathscr{A}$. Добавим к нему новые буквы и получим алфавит $\mathscr{A}_1$ с условием $\mathscr{A} \subseteq \mathscr{A}_1$.

Алфавит $\mathscr{A}_1$ называется расширением алфавита $\mathscr{A}$.

Принцип нормализации

Пусть задан произвольный вербальный алгоритм $\mathfrak{A}$ над алфавитом $\mathscr{A}$. Тогда существует расширение $\mathscr{A}_1$ алфавита $\mathscr{A}$ и нормальный алгоритм $\mathfrak{A}_1$ в алфавите $\mathscr{A}_1$ с условием: произвольное слово P в алфавите $\mathscr{A}$ перерабатывается нормальным алгоритмом $\mathfrak{A}_1$ в тот же самый результат, в который слово P перерабатывается исходным вербальным алгоритмом $\mathfrak{A}$.

Любой нормальный алгоритм эквивалентен некоторой машине Тьюринга, и наоборот — любая машина Тьюринга эквивалентна некоторому нормальному алгоритму.

Сложность алгоритмов

До создания ЭВМ все задачи рассматривались лишь с точки зрения алгоритмической разрешимости — т. е. существования или отсутствия алгоритма их решения.

Вопрос сложности алгоритма не возникал из-за малого объёма входных данных.

С появлением ЭВМ размерность решаемых задач резко увеличилась.

Встал вопрос: до каких значений можно увеличивать размерность задачи, чтобы её решение могло быть получено за приемлемое время и при этом хватало бы памяти машины для хранения исходных, промежуточных и выходных данных?

В связи с этим встала задача разработки наиболее эффективных алгоритмов, что повлекло появление нового направления в теории алгоритмов: анализа сложности алгоритмов.

Сложность алгоритмов

До создания ЭВМ все задачи рассматривались лишь с точки зрения алгоритмической разрешимости — т. е. существования или отсутствия алгоритма их решения.
Вопрос сложности алгоритма не возникал из-за малого объёма входных данных.
С появлением ЭВМ размерность решаемых задач резко увеличилась.
Встал вопрос: до каких значений можно увеличивать размерность задачи, чтобы её решение могло быть получено за приемлемое время и при этом хватало бы памяти машины для хранения исходных, промежуточных и выходных данных?
В связи с этим встала задача разработки наиболее эффективных алгоритмов, что повлекло появление нового направления в теории алгоритмов: анализа сложности алгоритмов.

Сложность алгоритмов

До создания ЭВМ все задачи рассматривались лишь с точки зрения алгоритмической разрешимости — т. е. существования или отсутствия алгоритма их решения.
Вопрос сложности алгоритма не возникал из-за малого объёма входных данных.
С появлением ЭВМ размерность решаемых задач резко увеличилась.
Встал вопрос: до каких значений можно увеличивать размерность задачи, чтобы её решение могло быть получено за приемлемое время и при этом хватало бы памяти машины для хранения исходных, промежуточных и выходных данных?
В связи с этим встала задача разработки наиболее эффективных алгоритмов, что повлекло появление нового направления в теории алгоритмов: анализа сложности алгоритмов.

Сложность алгоритмов

До создания ЭВМ все задачи рассматривались лишь с точки зрения алгоритмической разрешимости — т. е. существования или отсутствия алгоритма их решения.
Вопрос сложности алгоритма не возникал из-за малого объёма входных данных.
С появлением ЭВМ размерность решаемых задач резко увеличилась.
Встал вопрос: до каких значений можно увеличивать размерность задачи, чтобы её решение могло быть получено за приемлемое время и при этом хватало бы памяти машины для хранения исходных, промежуточных и выходных данных?
В связи с этим встала задача разработки наиболее эффективных алгоритмов, что повлекло появление нового направления в теории алгоритмов: анализа сложности алгоритмов.

Сложность алгоритмов

До создания ЭВМ все задачи рассматривались лишь с точки зрения алгоритмической разрешимости — т. е. существования или отсутствия алгоритма их решения.
Вопрос сложности алгоритма не возникал из-за малого объёма входных данных.
С появлением ЭВМ размерность решаемых задач резко увеличилась.
Встал вопрос: до каких значений можно увеличивать размерность задачи, чтобы её решение могло быть получено за приемлемое время и при этом хватало бы памяти машины для хранения исходных, промежуточных и выходных данных?
В связи с этим встала задача разработки наиболее эффективных алгоритмов, что повлекло появление нового направления в теории алгоритмов: анализа сложности алгоритмов.

Эффективность алгоритма

Эффективность алгоритма понимается в двух аспектах:

1. по времени решения задачи — временна́я сложность алгоритма (ВСА);
2. по объёму требуемой памяти — пространственная сложность алгоритма (ПСА).

Обычно под «самым эффективным» алгоритмом понимается самый быстрый, т. к. именно ограничение по времени определяет пригодность алгоритма для решения практических задач.

Эффективность алгоритма

Эффективность алгоритма понимается в двух аспектах:

1. по времени решения задачи — временна́я сложность алгоритма (ВСА);
2. по объёму требуемой памяти — пространственная сложность алгоритма (ПСА).

Обычно под «самым эффективным» алгоритмом понимается самый быстрый, т. к. именно ограничение по времени определяет пригодность алгоритма для решения практических задач.

Эффективность алгоритма

Эффективность алгоритма понимается в двух аспектах:

1. по времени решения задачи — временна́я сложность алгоритма (ВСА);
2. по объёму требуемой памяти — пространственная сложность алгоритма (ПСА).

Обычно под «самым эффективным» алгоритмом понимается самый быстрый, т. к. именно ограничение по времени определяет пригодность алгоритма для решения практических задач.

Эффективность алгоритма

Эффективность алгоритма понимается в двух аспектах:

1. по времени решения задачи — временна́я сложность алгоритма (ВСА);
2. по объёму требуемой памяти — пространственная сложность алгоритма (ПСА).

Обычно под «самым эффективным» алгоритмом понимается самый быстрый, т. к. именно ограничение по времени определяет пригодность алгоритма для решения практических задач.

Классы сложности

Классами сложности называются множества вычислительных задач, примерно одинаковых по сложности вычисления.
Говоря более узко, классы сложности — это множества предикатов, использующих для вычисления примерно одинаковые количества ресурсов.

Для каждого класса существует категория задач, которые являются «самыми сложными».
Это означает, что любая задача из класса сводится к такой задаче, и притом сама «самая сложная» задача лежит в классе.
Такие задачи называют полными задачами для данного класса.
Наиболее известным примером являются NP-полные задачи.

Классы сложности

Классами сложности называются множества вычислительных задач, примерно одинаковых по сложности вычисления.
Говоря более узко, классы сложности — это множества предикатов, использующих для вычисления примерно одинаковые количества ресурсов.

Для каждого класса существует категория задач, которые являются «самыми сложными».
Это означает, что любая задача из класса сводится к такой задаче, и притом сама «самая сложная» задача лежит в классе.
Такие задачи называют полными задачами для данного класса.
Наиболее известным примером являются NP-полные задачи.

Классы сложности

Классами сложности называются множества вычислительных задач, примерно одинаковых по сложности вычисления. Говоря более узко, классы сложности — это множества предикатов, использующих для вычисления примерно одинаковые количества ресурсов.

Для каждого класса существует категория задач, которые являются «самыми сложными».
Это означает, что любая задача из класса сводится к такой задаче, и притом сама «самая сложная» задача лежит в классе.
Такие задачи называют полными задачами для данного класса.
Наиболее известным примером являются NP-полные задачи.

Классы сложности

Классами сложности называются множества вычислительных задач, примерно одинаковых по сложности вычисления.
Говоря более узко, классы сложности — это множества предикатов, использующих для вычисления примерно одинаковые количества ресурсов.

Для каждого класса существует категория задач, которые являются «самыми сложными».
Это означает, что любая задача из класса сводится к такой задаче, и притом сама «самая сложная» задача лежит в классе.
Такие задачи называют полными задачами для данного класса.
Наиболее известным примером являются NP-полные задачи.

Классы сложности

Классами сложности называются множества вычислительных задач, примерно одинаковых по сложности вычисления.
Говоря более узко, классы сложности — это множества предикатов, использующих для вычисления примерно одинаковые количества ресурсов.

Для каждого класса существует категория задач, которые являются «самыми сложными».
Это означает, что любая задача из класса сводится к такой задаче, и притом сама «самая сложная» задача лежит в классе.
Такие задачи называют полными задачами для данного класса.
Наиболее известным примером являются *NP*-полные задачи.

Классы сложности

Классами сложности называются множества вычислительных задач, примерно одинаковых по сложности вычисления.
Говоря более узко, классы сложности — это множества предикатов, использующих для вычисления примерно одинаковые количества ресурсов.

Для каждого класса существует категория задач, которые являются «самыми сложными».
Это означает, что любая задача из класса сводится к такой задаче, и притом сама «самая сложная» задача лежит в классе.
Такие задачи называют полными задачами для данного класса.
Наиболее известным примером являются *NP*-полные задачи.

Определение класса

Каждый класс сложности (в узком смысле) определяется как множество предикатов, обладающих некоторыми свойствами.

Типичное определение класса сложности выглядит так:

> Классом сложности X называется множество предикатов $P(x)$, вычислимых на машинах Тьюринга и использующих для вычисления $O(f(n))$ ресурса, где n — длина слова x.

В качестве ресурсов обычно берутся время вычисления (количество рабочих тактов машины Тьюринга) или рабочая зона (количество использованных ячеек на ленте во время работы).

Языки, распознаваемые предикатами из некоторого класса (т. е. множества слов, на которых предикат возвращает 1), также называются принадлежащими тому же классу.

Определение класса

Каждый класс сложности (в узком смысле) определяется как множество предикатов, обладающих некоторыми свойствами. Типичное определение класса сложности выглядит так:

Классом сложности X называется множество предикатов $P(x)$, вычислимых на машинах Тьюринга и использующих для вычисления $O(f(n))$ ресурса, где n — длина слова x.

В качестве ресурсов обычно берутся время вычисления (количество рабочих тактов машины Тьюринга) или рабочая зона (количество использованных ячеек на ленте во время работы).

Языки, распознаваемые предикатами из некоторого класса (т. е. множества слов, на которых предикат возвращает 1), также называются принадлежащими тому же классу.

Определение класса

Каждый класс сложности (в узком смысле) определяется как множество предикатов, обладающих некоторыми свойствами. Типичное определение класса сложности выглядит так:

Классом сложности X называется множество предикатов $P(x)$, вычислимых на машинах Тьюринга и использующих для вычисления $O(f(n))$ ресурса, где n — длина слова x.

В качестве ресурсов обычно берутся время вычисления (количество рабочих тактов машины Тьюринга) или рабочая зона (количество использованных ячеек на ленте во время работы).

Языки, распознаваемые предикатами из некоторого класса (т. е. множества слов, на которых предикат возвращает 1), также называются принадлежащими тому же классу.

Определение класса

Каждый класс сложности (в узком смысле) определяется как множество предикатов, обладающих некоторыми свойствами. Типичное определение класса сложности выглядит так:

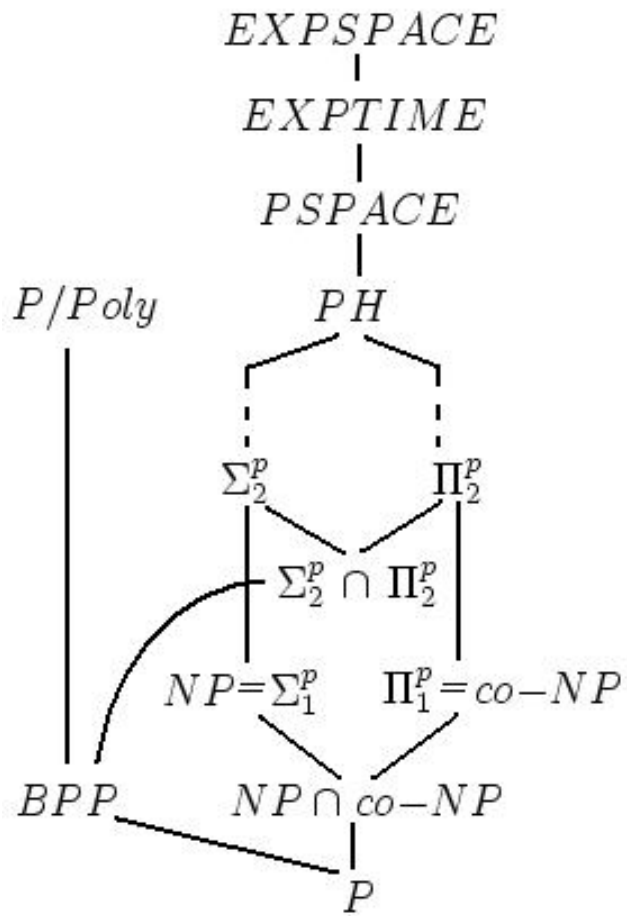
Классом сложности X называется множество предикатов $P(x)$, вычислимых на машинах Тьюринга и использующих для вычисления $O(f(n))$ ресурса, где n — длина слова x.

В качестве ресурсов обычно берутся время вычисления (количество рабочих тактов машины Тьюринга) или рабочая зона (количество использованных ячеек на ленте во время работы).
Языки, распознаваемые предикатами из некоторого класса (т. е. множества слов, на которых предикат возвращает 1), также называются принадлежащими тому же классу.

Иерархия классов сложности

Классы принято обозначать прописными буквами. Дополнение к классу C (т. е. класс всевозможных языков, которые не принадлежат C) обозначается *co*-C.

Все классы сложности находятся в иерархическом отношении: одни включают в себя другие.
Однако про большинство включений неизвестно, являются ли они строгими.
Одна из наиболее известных открытых проблем в этой области — равенство классов P и NP.



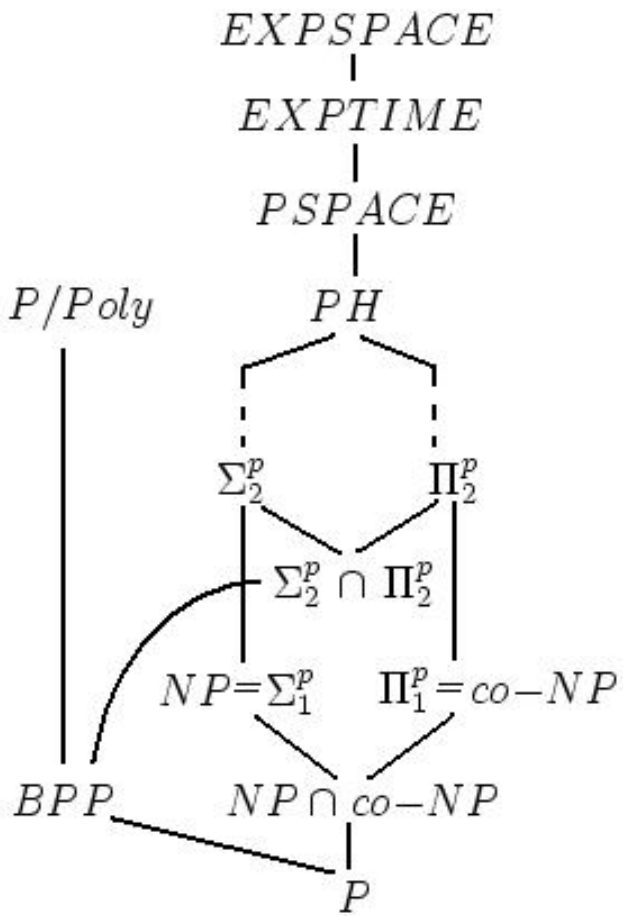
Иерархия классов сложности

Классы принято обозначать прописными буквами. Дополнение к классу C (т. е. класс всевозможных языков, которые не принадлежат C) обозначается *co*-C.

Все классы сложности находятся в иерархическом отношении: одни включают в себя другие.

Однако про большинство включений неизвестно, являются ли они строгими.

Одна из наиболее известных открытых проблем в этой области — равенство классов P и NP.

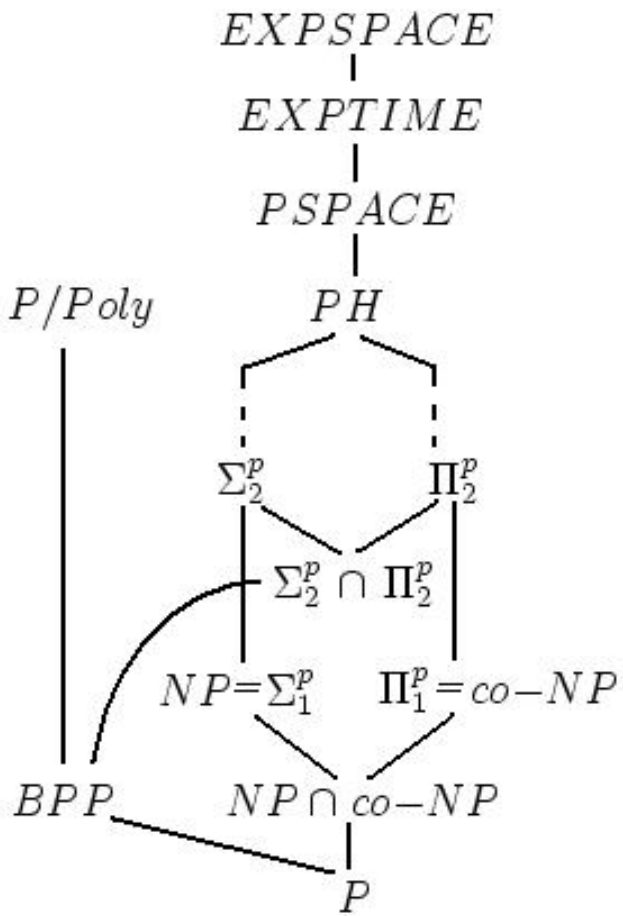


Иерархия классов сложности

Классы принято обозначать прописными буквами. Дополнение к классу C (т. е. класс всевозможных языков, которые не принадлежат C) обозначается *co*-C.

Все классы сложности находятся в иерархическом отношении: одни включают в себя другие.
Однако про большинство включений неизвестно, являются ли они строгими.

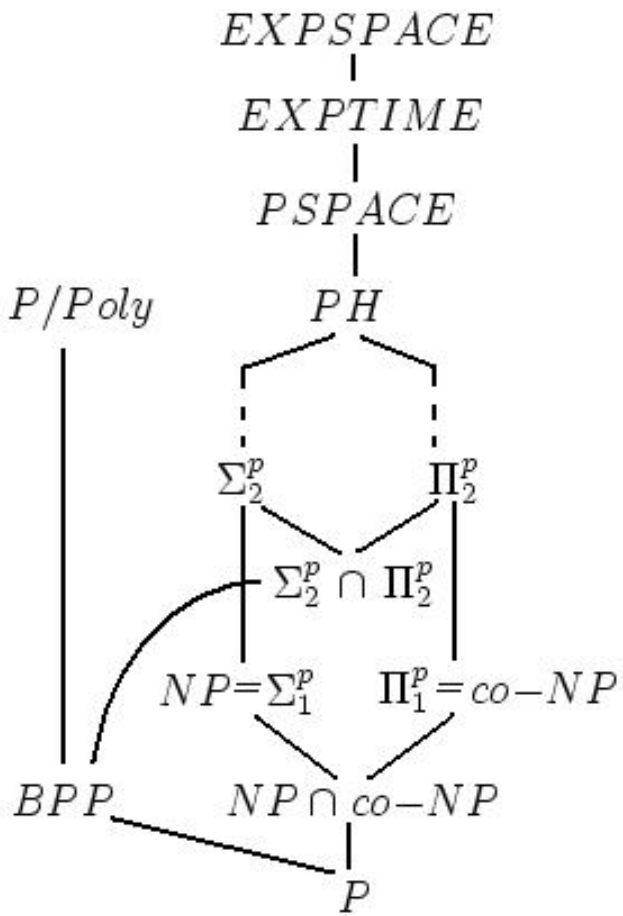
Одна из наиболее известных открытых проблем в этой области — равенство классов P и NP.



Иерархия классов сложности

Классы принято обозначать прописными буквами. Дополнение к классу C (т. е. класс всевозможных языков, которые не принадлежат C) обозначается *co*-C.

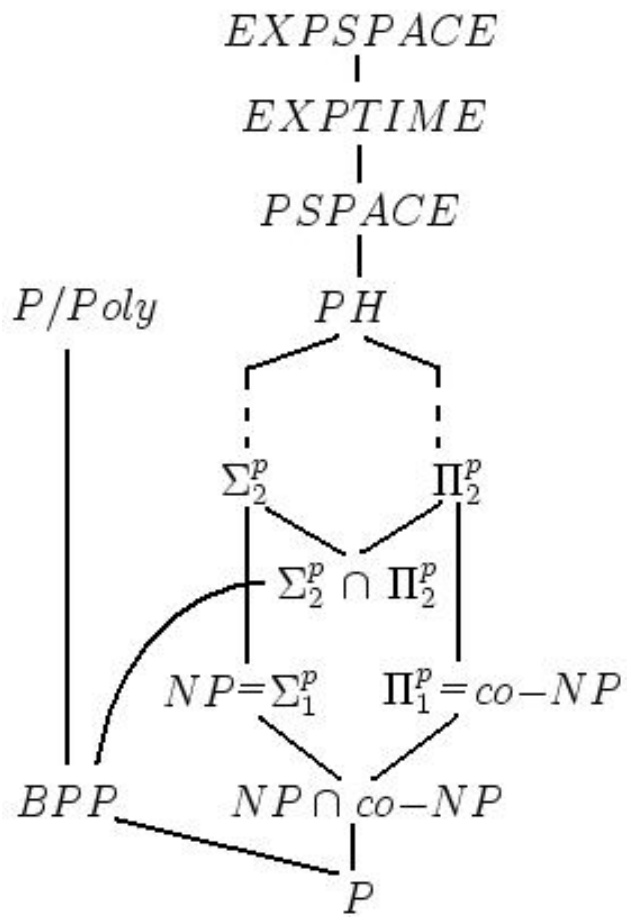
Все классы сложности находятся в иерархическом отношении: одни включают в себя другие.
Однако про большинство включений неизвестно, являются ли они строгими.
Одна из наиболее известных открытых проблем в этой области — равенство классов P и NP.



Гипотезы о равенстве классов

Если предположение о равенстве классов P и NP верно (в чём большинство учёных сомневается), то представленная справа иерархия классов сильно свернётся.

На данный момент наиболее распространённой является гипотеза о невырожденности иерархии (т. е. все классы различны).
Кроме того, известно, что класс *EXPSPACE* не равен классу *PSPACE*.

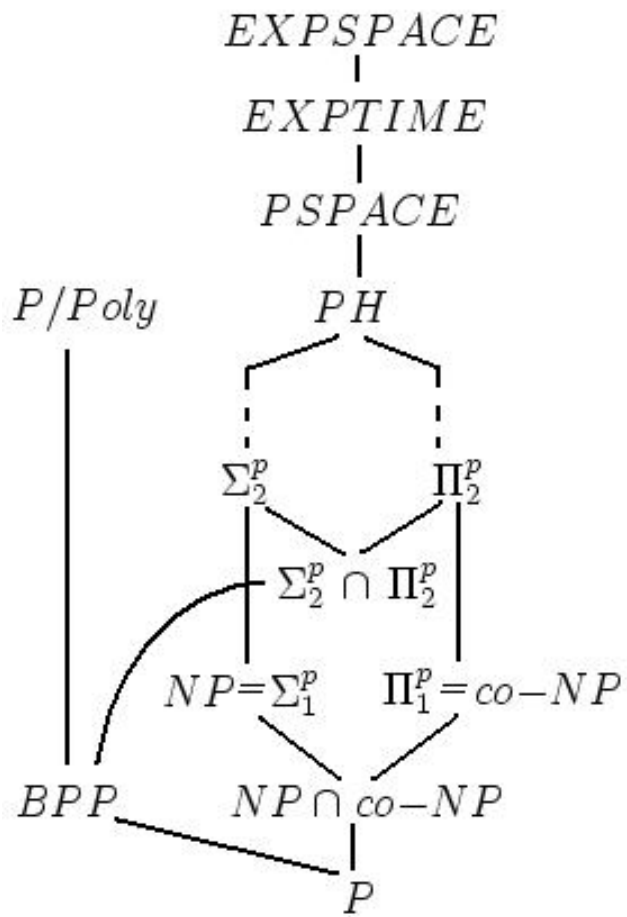


Гипотезы о равенстве классов

Если предположение о равенстве классов P и NP верно (в чём большинство учёных сомневается), то представленная справа иерархия классов сильно свернётся.

На данный момент наиболее распространённой является гипотеза о невырожденности иерархии (т. е. все классы различны).

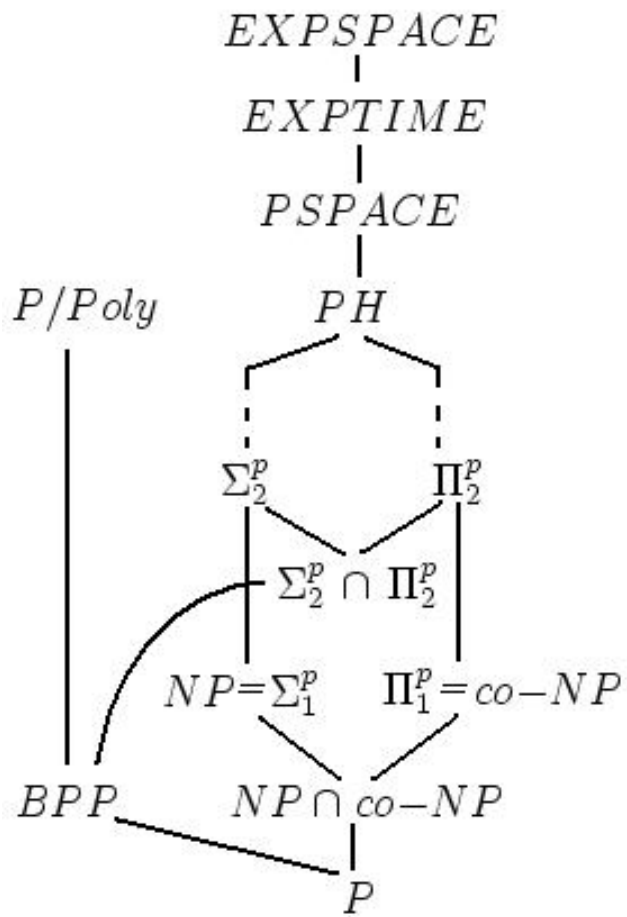
Кроме того, известно, что класс *EXPSPACE* не равен классу *PSPACE*.



Гипотезы о равенстве классов

Если предположение о равенстве классов P и NP верно (в чём большинство учёных сомневается), то представленная справа иерархия классов сильно свернётся.

На данный момент наиболее распространённой является гипотеза о невырожденности иерархии (т. е. все классы различны).
Кроме того, известно, что класс *EXPSPACE* не равен классу *PSPACE*.



Класс сложности P

Классом P (от *англ.* polynomial) называют множество задач, для которых существуют «быстрые» алгоритмы решения (время работы которых полиномиально зависит от размера входных данных).

Класс P включён в более широкие классы сложности алгоритмов.

Обычно алгоритмы класса P отождествляется с детерминированной машиной Тьюринга, которая вычисляет ответ по данному на входную ленту слову из входного алфавита Σ.
Временем работы алгоритма $T_M(x)$ при фиксированном входном слове x называется количество рабочих тактов машины Тьюринга от начала до остановки машины.

Класс сложности P

Классом P (от *англ.* polynomial) называют множество задач, для которых существуют «быстрые» алгоритмы решения (время работы которых полиномиально зависит от размера входных данных).

Класс P включён в более широкие классы сложности алгоритмов.

Обычно алгоритмы класса P отождествляется с детерминированной машиной Тьюринга, которая вычисляет ответ по данному на входную ленту слову из входного алфавита Σ.
Временем работы алгоритма $T_M(x)$ при фиксированном входном слове x называется количество рабочих тактов машины Тьюринга от начала до остановки машины.

Класс сложности P

Классом P (от *англ.* polynomial) называют множество задач, для которых существуют «быстрые» алгоритмы решения (время работы которых полиномиально зависит от размера входных данных).

Класс P включён в более широкие классы сложности алгоритмов.

Обычно алгоритмы класса P отождествляется с детерминированной машиной Тьюринга, которая вычисляет ответ по данному на входную ленту слову из входного алфавита Σ.

Временем работы алгоритма $T_M(x)$ при фиксированном входном слове x называется количество рабочих тактов машины Тьюринга от начала до остановки машины.

Класс сложности P

Классом P (от *англ.* polynomial) называют множество задач, для которых существуют «быстрые» алгоритмы решения (время работы которых полиномиально зависит от размера входных данных).

Класс P включён в более широкие классы сложности алгоритмов.

Обычно алгоритмы класса P отождествляется с детерминированной машиной Тьюринга, которая вычисляет ответ по данному на входную ленту слову из входного алфавита Σ.
Временем работы алгоритма $T_M(x)$ при фиксированном входном слове x называется количество рабочих тактов машины Тьюринга от начала до остановки машины.

Сложность P и машина Тьюринга

Сложностью функции $f\colon \Sigma^* \to \Sigma^*$, вычисляемой некоторой машиной Тьюринга, называется функция $C\colon \mathbb{N} \to \mathbb{N}$, зависящая от длины входного слова и равная максимуму времени работы машины $\mathfrak{M}$ по всем входным словам фиксированной длины n:

$$C_{\mathfrak{M}}(n) = \max_{x:\ |x|=n} T_{\mathfrak{M}}(x).$$

Если для функции f существует машина Тьюринга $\mathfrak{M}$ такая, что $C_{\mathfrak{M}}(n) < n^c$ для некоторого числа c и достаточно больших n, то говорят, что она принадлежит классу P, или полиномиальна по времени.

Сложность P и машина Тьюринга

Сложностью функции $f\colon \Sigma^* \to \Sigma^*$, вычисляемой некоторой машиной Тьюринга, называется функция $C\colon \mathbb{N} \to \mathbb{N}$, зависящая от длины входного слова и равная максимуму времени работы машины $\mathfrak{M}$ по всем входным словам фиксированной длины n:

$$C_{\mathfrak{M}}(n) = \max_{x:\ |x|=n} T_{\mathfrak{M}}(x).$$

Если для функции f существует машина Тьюринга $\mathfrak{M}$ такая, что $C_{\mathfrak{M}}(n) < n^c$ для некоторого числа c и достаточно больших n, то говорят, что она принадлежит классу P, или полиномиальна по времени.

Примеры задач класса P

Примерами задач из класса P являются целочисленное сложение, умножение, деление, взятие остатка от деления, умножения матриц, выяснение связности графов и некоторые другие.

Существует много задач, для которых не найдено полиномиального алгоритма, но не доказано, что его не существует.

Соответственно, неизвестно, принадлежат ли такие задачи классу P. Вот некоторые из них:

- Задача коммивояжёра (а также все остальные NP-полные задачи).
- Разложение числа на простые множители.
- Дискретное логарифмирование в конечном поле.
- Задача о скрытой подгруппе с n образующими.
- Дискретное логарифмирование в аддитивной группе точек на эллиптической кривой.

Примеры задач класса P

Примерами задач из класса P являются целочисленное сложение, умножение, деление, взятие остатка от деления, умножения матриц, выяснение связности графов и некоторые другие.

Существует много задач, для которых не найдено полиномиального алгоритма, но не доказано, что его не существует.

Соответственно, неизвестно, принадлежат ли такие задачи классу P. Вот некоторые из них:

- Задача коммивояжёра (а также все остальные NP-полные задачи).
- Разложение числа на простые множители.
- Дискретное логарифмирование в конечном поле.
- Задача о скрытой подгруппе с n образующими.
- Дискретное логарифмирование в аддитивной группе точек на эллиптической кривой.

Примеры задач класса P

Примерами задач из класса P являются целочисленное сложение, умножение, деление, взятие остатка от деления, умножения матриц, выяснение связности графов и некоторые другие.

Существует много задач, для которых не найдено полиномиального алгоритма, но не доказано, что его не существует.

Соответственно, неизвестно, принадлежат ли такие задачи классу P. Вот некоторые из них:

- Задача коммивояжёра (а также все остальные NP-полные задачи).
- Разложение числа на простые множители.
- Дискретное логарифмирование в конечном поле.
- Задача о скрытой подгруппе с n образующими.
- Дискретное логарифмирование в аддитивной группе точек на эллиптической кривой.

Примеры задач класса P

Примерами задач из класса P являются целочисленное сложение, умножение, деление, взятие остатка от деления, умножения матриц, выяснение связности графов и некоторые другие.
Существует много задач, для которых не найдено полиномиального алгоритма, но не доказано, что его не существует.
Соответственно, неизвестно, принадлежат ли такие задачи классу P. Вот некоторые из них:

- Задача коммивояжёра (а также все остальные NP-полные задачи).
- Разложение числа на простые множители.
- Дискретное логарифмирование в конечном поле.
- Задача о скрытой подгруппе с n образующими.
- Дискретное логарифмирование в аддитивной группе точек на эллиптической кривой.

Класс сложности *NP*

Классом *NP* (от *англ.* non-deterministic polynomial) называют множество задач, решение которых при наличии некоторых дополнительных сведений можно «быстро» (за время, не превосходящее полинома от размера данных) проверить на машине Тьюринга.

Класс *NP* можно также определить как содержащий задачи, которые можно «быстро» решить на недетерминированной машине Тьюринга.

У недетерминированной машины Тьюринга в программе могут существовать разные строки с одинаковой левой частью. Если машина встретила «развилку», т. е. неоднозначность в программе, то дальше возможны разные варианты вычисления (считается, что они выполняются одновременно). Если имеется информация о том, какой вариант когда необходимо выбирать, то такая машина является детерминированной.

Класс сложности *NP*

Классом *NP* (от *англ.* non-deterministic polynomial) называют множество задач, решение которых при наличии некоторых дополнительных сведений можно «быстро» (за время, не превосходящее полинома от размера данных) проверить на машине Тьюринга.

Класс *NP* можно также определить как содержащий задачи, которые можно «быстро» решить на недетерминированной машине Тьюринга.

У недетерминированной машины Тьюринга в программе могут существовать разные строки с одинаковой левой частью. Если машина встретила «развилку», т. е. неоднозначность в программе, то дальше возможны разные варианты вычисления (считается, что они выполняются одновременно). Если имеется информация о том, какой вариант когда необходимо выбирать, то такая машина является детерминированной.

Класс сложности *NP*

Классом *NP* (от *англ.* non-deterministic polynomial) называют множество задач, решение которых при наличии некоторых дополнительных сведений можно «быстро» (за время, не превосходящее полинома от размера данных) проверить на машине Тьюринга.

Класс *NP* можно также определить как содержащий задачи, которые можно «быстро» решить на недетерминированной машине Тьюринга.

У недетерминированной машины Тьюринга в программе могут существовать разные строки с одинаковой левой частью. Если машина встретила «развилку», т. е. неоднозначность в программе, то дальше возможны разные варианты вычисления (считается, что они выполняются одновременно). Если имеется информация о том, какой вариант когда необходимо выбирать, то такая машина является детерминированной.

Класс сложности *NP*

Классом *NP* (от *англ.* non-deterministic polynomial) называют множество задач, решение которых при наличии некоторых дополнительных сведений можно «быстро» (за время, не превосходящее полинома от размера данных) проверить на машине Тьюринга.

Класс *NP* можно также определить как содержащий задачи, которые можно «быстро» решить на недетерминированной машине Тьюринга.

У недетерминированной машины Тьюринга в программе могут существовать разные строки с одинаковой левой частью. Если машина встретила «развилку», т. е. неоднозначность в программе, то дальше возможны разные варианты вычисления (считается, что они выполняются одновременно). Если имеется информация о том, какой вариант когда необходимо выбирать, то такая машина является детерминированной.

Класс сложности *NP*

Классом *NP* (от *англ.* non-deterministic polynomial) называют множество задач, решение которых при наличии некоторых дополнительных сведений можно «быстро» (за время, не превосходящее полинома от размера данных) проверить на машине Тьюринга.

Класс *NP* можно также определить как содержащий задачи, которые можно «быстро» решить на недетерминированной машине Тьюринга.

У недетерминированной машины Тьюринга в программе могут существовать разные строки с одинаковой левой частью. Если машина встретила «развилку», т. е. неоднозначность в программе, то дальше возможны разные варианты вычисления (считается, что они выполняются одновременно). Если имеется информация о том, какой вариант когда необходимо выбирать, то такая машина является детерминированной.

Принадлежность классу *NP*

Предикат $R(x)$, который представляет данная недетерминированная машина Тьюринга, считается равным 1, если существует хотя бы один вариант вычисления, возвращающий 1, и 0, если все варианты возвращают 0.

Если длина вычисления, дающего 1, не превосходит некоторого многочлена от длины x, то предикат называется принадлежащим классу *NP*.

Если у языка существует распознающий его предикат из класса *NP*, то язык называется принадлежащим классу *NP*.

Принадлежность классу *NP*

Предикат $R(x)$, который представляет данная недетерминированная машина Тьюринга, считается равным 1, если существует хотя бы один вариант вычисления, возвращающий 1, и 0, если все варианты возвращают 0.

Если длина вычисления, дающего 1, не превосходит некоторого многочлена от длины x, то предикат называется принадлежащим классу *NP*.

Если у языка существует распознающий его предикат из класса *NP*, то язык называется принадлежащим классу *NP*.

Принадлежность классу *NP*

Предикат $R(x)$, который представляет данная недетерминированная машина Тьюринга, считается равным 1, если существует хотя бы один вариант вычисления, возвращающий 1, и 0, если все варианты возвращают 0.

Если длина вычисления, дающего 1, не превосходит некоторого многочлена от длины x, то предикат называется принадлежащим классу *NP*.

Если у языка существует распознающий его предикат из класса *NP*, то язык называется принадлежащим классу *NP*.

Соотношение с другими классами

Класс языков, дополнения которых принадлежат NP, называется классом $co\text{-}NP$, хотя и не доказано, что этот класс отличен от класса NP.

Пересечение классов $NP \cap co\text{-}NP$ содержит класс P.

В частности, класс NP включает в себя класс P.

Однако ничего не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов уже более 30 лет.

Если они равны, то любую задачу из класса NP можно будет решить за полиномиальное время.

Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос.

Соотношение с другими классами

Класс языков, дополнения которых принадлежат NP, называется классом $co\text{-}NP$, хотя и не доказано, что этот класс отличен от класса NP.

Пересечение классов $NP \cap co\text{-}NP$ содержит класс P.

В частности, класс NP включает в себя класс P.

Однако ничего не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов уже более 30 лет.

Если они равны, то любую задачу из класса NP можно будет решить за полиномиальное время.

Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос.

Соотношение с другими классами

Класс языков, дополнения которых принадлежат NP, называется классом $co\text{-}NP$, хотя и не доказано, что этот класс отличен от класса NP.
Пересечение классов $NP \cap co\text{-}NP$ содержит класс P.
В частности, класс NP включает в себя класс P.
Однако ничего не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов уже более 30 лет.
Если они равны, то любую задачу из класса NP можно будет решить за полиномиальное время.
Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос.

Соотношение с другими классами

Класс языков, дополнения которых принадлежат NP, называется классом $co\text{-}NP$, хотя и не доказано, что этот класс отличен от класса NP.
Пересечение классов $NP \cap co\text{-}NP$ содержит класс P.
В частности, класс NP включает в себя класс P.
Однако ничего не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов уже более 30 лет.
Если они равны, то любую задачу из класса NP можно будет решить за полиномиальное время.
Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос.

Соотношение с другими классами

Класс языков, дополнения которых принадлежат NP, называется классом $co\text{-}NP$, хотя и не доказано, что этот класс отличен от класса NP.
Пересечение классов $NP \cap co\text{-}NP$ содержит класс P.
В частности, класс NP включает в себя класс P.
Однако ничего не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов уже более 30 лет.
Если они равны, то любую задачу из класса NP можно будет решить за полиномиальное время.
Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос.

Соотношение с другими классами

Класс языков, дополнения которых принадлежат NP, называется классом $co\text{-}NP$, хотя и не доказано, что этот класс отличен от класса NP.
Пересечение классов $NP \cap co\text{-}NP$ содержит класс P.
В частности, класс NP включает в себя класс P.
Однако ничего не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов уже более 30 лет.
Если они равны, то любую задачу из класса NP можно будет решить за полиномиальное время.
Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос.

Соотношение с другими классами

Класс языков, дополнения которых принадлежат NP, называется классом $co\text{-}NP$, хотя и не доказано, что этот класс отличен от класса NP.
Пересечение классов $NP \cap co\text{-}NP$ содержит класс P.
В частности, класс NP включает в себя класс P.
Однако ничего не известно о строгости этого включения.

Задача о равенстве классов P и NP является одной из центральных открытых проблем теории алгоритмов уже более 30 лет.
Если они равны, то любую задачу из класса NP можно будет решить за полиномиальное время.
Однако научное сообщество склоняется в сторону отрицательного ответа на этот вопрос.

NP-полные задачи

NP-полная задача — задача из класса NP, к которой можно свести любую другую задачу из класса *NP* за полиномиальное время.

NP-полные задачи образуют в некотором смысле подмножество «самых сложных» задач в классе *NP*, и если для какой-то из них будет найден «быстрый» алгоритм решения, то и любая другая задача из класса *NP* может быть решена так же «быстро».

NP-полные задачи

NP-полная задача — задача из класса NP, к которой можно свести любую другую задачу из класса *NP* за полиномиальное время.
NP-полные задачи образуют в некотором смысле подмножество «самых сложных» задач в классе *NP*, и если для какой-то из них будет найден «быстрый» алгоритм решения, то и любая другая задача из класса *NP* может быть решена так же «быстро».

Примеры NP-полных задач

- Задача коммивояжёра
- Задача о выполнимости булевых формул
- Кратчайшее решение «пятнашек» размера $n \times n$
- Проблема Штейнера
- Проблема раскраски графа
- Задача о вершинном покрытии
- Задача о покрытии множества
- Задача о клике
- Задача о независимом множестве
- Игра «Сапёр»