Теза Тюрінга. Рекурсивні функції. Теза Чорча.

Теза Тюрінга. Приклади алгоритмічно нерозв'язних проблем

До цього часу нам удавалося для всіх процедур, які „претендують" на алгоритмічність, тобто ефективних, будувати машини Тюрінга, що їх реалізують. Чи правильно це загалом? Ствердна відповідь на це питання міститься в тезі Тюрінга, яку формулюють так: будь-який алгоритм можна реалізувати машиною Тюрінга. Довести тезу Тюрінга неможливо, бо саме поняття алгоритму (ефективної процедури) інтуїтивне (неточне). Це не теорема й не аксіома математичної теорії, а твердження, яке пов'язує теорію з тими об'єктами, для яких її побудовано. За своїм характером теза Тюрінга нагадує гіпотези фізики про адекватність математичних моделей фізичним явищам і процесам. Підтверджує тезу Тюрінга, по-перше, математична практика, по-друге — та обставина, що опис алгоритму в термінах будь-якої іншої відомої алгоритмічної моделі можна звести до його опису у вигляді машини Тюрінга. Теза Тюрінга дає змогу, по-перше, замінити неточні (інтуїтивні) твердження про існування ефективних процедур (алгоритмів) точними твердженнями про існування машин Тюрінга, а, по-друге, тлумачити твердження про неіснування машин Тюрінга як твердження про неіснування алгоритму.

Розглянемо тепер два приклади алгоритмічно нерозв'язних проблем.

Проблема самозастосовності. Закодуємо всі машини Тюрінга символами алфавіту {*, 1} так, щоб за системою команд машини Тюрінга можна було ефективно побудувати код машини та, навпаки, за кодом можна було ефективно записувати її програму. Це можна зробити, наприклад, так. Нехай Т — будь-яка машина Тюрінга, зовнішній алфавіт якої — $[Лa_1, a_2, \ldots, a_m]$, а множина внутрішніх станів —$[q_0, q_1, \ldots, q_k]$. Занумеруємо всі символи, що є в командах (окрім →), числами 0, і, 2, ... так, як це показано в табл. 1.

Таблиця 1.

Символ	$Н$	Л	$П$	$Л$	a_1	...	a_m	q_0	q_1	...	q_k
Номер	0	1	2	3	4	$\ldots m+3$		$m+4$	$m+5$	$\ldots$	$m+4+k$

Число і подаватимемо набором з $i+1$ одиниць. Якщо b — один із символів Н, Л, $П, Л, a_1, \ldots, a_m, q_0, q_1, \ldots, q_k$, то як $N(b)$ позначимо набір з одиниць, кількість яких відповідає номеру символу b. Наприклад,$N(H)-1$, $N(Л)=11$, $N(П)=111, N(Л)=1111$, $N(a_1)=11111$, $N(a_2)=111111$. Команді $q_1a_1 \to a_1Dq_1$поставимо у відповідність слово

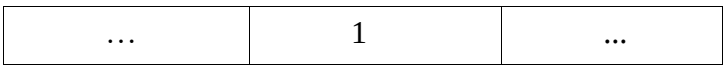
$$C(q_1a_1) = N(q_1) * N(a_2) * N(a_j) * N(D) * N(q_1),$$

яке називають *кодом* цієї *команди*. *Кодом машини Т* називають слово

$$W(T) =** C(q_1Л) ** C(q_1a_1) **\ldots** C(q_1a_m) ** C(q_2Л) ** C(q_2a_1) **\ldots** C(q_2a_m) ** C(q_3Л) **\ldots* * C(q_kЛ) **\ldots** C(q_ka_m) **$$

(тут у визначеному порядку записано коди всіх команд, відокремлені один від другого двома зірочками).

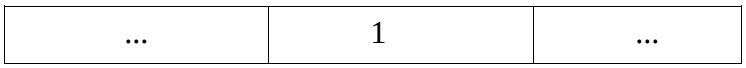
Розглядатимемо машини Тюрінга, зовнішній алфавіт яких містить символи * й 1. Нехай *Т* — одна з таких машин. Якщо машина *Т* застосовна до слова *W(T)*, тобто до власного коду, то її називають самозаапосовною, а ні, то несамозастосовною. Кожна машина розглянутого типу або самозастосовна, або несамозастосовна.



q_0

Рис. 1

Проблема самозастосовності полягає в тому, що потрібно навести алгоритм для визначення того, чи дана машина Тюрінга самозастосовна, чи ні. Згідно з тезою Тюрінга цю задачу можна сформулювати як задачу про побудову машини Тюрінга, котра застосовна до кодів усіх машин, і заключні конфігурації в разі роботи над кодами самозастосовних і несамозастосовних машин відрізняються. Наприклад, вимагатимемо, щоб у разі роботи над кодом самозастосовної машини заключна конфігурація мала такий вигляд, як на рис. 1, а в разі роботи над кодом несамозастосовної машини — як на рис. 2. На цих рисунках поза зчитуваним символом можуть бути будь-які символи зовнішнього алфавіту.



q_0

Рис. 2

ТЕОРЕМА 1. Проблема самозастосовності алгоритмічно нерозв'язна, тобто не існує машини Тюрінга, яка розв'язує в наведеному вище розумінні цю проблему. Доведення. Припустимо протилежне. Нехай існує машина *L*, яка розв'язує проблему самозастосовності. За нею побудуємо машину *T*, що має такі властивості.

1. Машина *T* застосовна до кодів несамозастосовних машин.

2. Машина *T* незастосовна до кодів самозастосовних машин.

Машину *T* будують так. Усі команди машини *L* оголошують командами машини *T*, але заключний стан q_0 машини *L* оголошують незаключним станом машини *T*. Будемо вважати заключним станом машини *T* новий стан q_0' додамо досистеми команд машини *T* ще такі дві команди:

$$q_0 0 \to 0 H q_0' \text{і } q_0 1 \to 1 H q_0$$

Очевидно, що машина *T* має зазначені властивості 1 і 2, але вона чи самозастосовна, чи несамозастосовна. Нехай машина *T* самозастосовна, тобто вона застосовна до свого коду (коду самозастосовної машини), проте це суперечить властивості 2. Якщо ж машина *T* несамозастосовна, то вона незастосовна до своготоду (коду несамозастосовної машини). Але це суперечить властивості 1. Отже,машина *T* не може бути ні самозастосовною, ні несамозастосовною. Одержали суперечність. Оскільки машину *T* побудовано з машини *L* цілком конструктивно, то машини *L* не існує.

Проблема зупинки. Серед вимог, яким має задовольняти алгоритм названо результативність. Найрадикальнішим ЇЇ формулюванням була б вимога, щоб для будь-якого алгоритму *A* та даних а можна було визначити, чи приведе робота алгоритму *A* для початкових данихα до результату. Інакше кажучи, потрібно побудувати такий алгоритм *B*, що B(A, α) = 1, якщо A(α) дає результат, і B(A, α) = 0, якщо A(α) не дає результату.

Згідно з тезою Тюрінга цю задачу можна сформулювати так. За машиною *T* та словом α розпізнати, чи зупиниться машина *T,* розпочавши роботу над словом α (тобто розпізнати, чи застосовна машина *T* до словаα). Отже, потрібно побудувати таку машину *P*, яка була б застосовна до всіх слів *W(T)A*α, де *T* — довільна машина, α— довільне слово. Окрім того, заключні конфігурації машини *P* мають бути різними залежно від того, чи застосовна машина *T* до словаα , чи незастосовна.

ТЕОРЕМА 2. Проблема зупинки алгоритмічно нерозв'язна, тобто не існує машини Тюрінга *P*, яка розв'язує в наведеному вище розумінні названу проблему.

Доведення. Нехай існує машина *P*, яка розв'язує проблему зупинки, а *T* — якась машина Тюрінга. Тоді машина *P* має бути застосовна до всіх слів *W(T)Λ*α, зокрема якщоα = *W(T).* Отже,

машина *P* має бути застосовна до слова W(T)ΛW(T). У разі роботи над словом W(T)ΛW(T) вона зупиниться в конфігурації, зображеній на рис. 1, якщо машина *T* застосовна до слова *W(T)* (тобто якщо машина *T* самозастосовна), і в конфігурації, зображеній на рис. 9.8, якщо машина *T* незастосовна до слова W(T) (тобто якщо машина *T* несамозастосовна).

Нагадаємо, що машина, яка розв'язує проблему самозастосовності, починає працювати зі словом *W(T)* на стрічці.

Нерозв'язність проблеми зупинки можна інтерпретувати як неіснування загального алгоритму для налагодження програм, котрий за текстом довільної програми й даними для неї міг би визначити, чи зациклиться програма на цих даних. Якщо врахувати зроблене перед цим зауваження, то така інтерпретація не суперечить тому емпіричному факту, що більшість програм вдається налагодити, тобто виявити зациклення, знайти його причину й усунути її. При цьому вирішальну роль відіграють досвід і майстерність програміста.

Рекурсивні функції. Означення рекурсивної функції тісно пов'язане з поняттям алгоритму, оскільки саме визначення алгоритму поділяється на три основні типи алгоритмічних моделей. Перше з понять безпосередньо пов'язане з математичними визначеннями обчислення та числових функцій. Найпопулярнішою моделлю цього типу і є рекурсивні функції — історично перше і найбільш узагальнене поняття алгоритму.

Оскільки кожен алгоритм ставить у відповідність певний кінцевий результат, то окремо взятий алгоритм однозначно пов'язаний з функцією, значення якої він обчислює. Однак дослідження з використанням машини Тюрінга показують, що не для усіх функцій можна побудувати алгоритм для її обрахунку. На основі цих досліджень і була створена теорія рекурсивних функцій. Ця теорія базується на скінченному або ж конструктивному підході, при якому вся множина досліджуваних об'єктів (в нашому випадку функцій) складається із скінченної кількості вихідних об'єктів з допомогою простих операцій, ефективність виконання яких очевидна.

Простіше можна сказати так: складну функцію можна обчислити за допомогою її розбиття на прості функції, які визначені на всій множині дійсних чисел і їх обчислення індуктивним методом.

Рекурсивні функцій піддавались інтенсивним дослідженням із самого моменту їх введення, оскільки вони виступали як еквівалент поняття ефективно рекурсивних функцій. Перш за все було виділено підкласи простіших функцій(елементарних).

Існує три основні види простих функцій, такі, що набувають своїх значень на множині простих чисел:

- *Нуль-функція* — функція, де кожне значення її аргумент з області визначення набуває нульового значення, тобто $O^n(x_1, x_2, \ldots, x_n) = 0$.
- *Функція наступності* — функція, де кожне значення аргумента відповідає значенню функції, на одиницю більшому, тобто $S(x) = x + 1$для усіх x.
- *Функції вибору аргумента(селективна функція)* – $S_m^n(x_1, x_2, \ldots, x_n) = x_m$ при всіх $(x_1, \ldots, x_n) \in N_0^n, m = \overline{(1, n)}, n = 1,2,3\ldots$наприклад $S_1^3(x_1, x_2, x_3) = x_3$.

Для більш складних функцій використовуються різноманітні операції. Одними з найважливіших є операції суперпозиції, примітивної рекурсії та мінімізації. Перші дві операції розглядаються в рамках поняття примітивно-рекурсивної функції. Визначення такої функції дав американський математик Кліні, а потім і Черч. Останній висловив гіпотезу про те, що множина всіх рекурсивних функцій співпадає з множиною всіх обчислюваних функцій. Цю гіпотезу називають гіпотезою Черча. Розглянемо кожну з операцій окремо.

Операція суперпозиції (іноді вживають термін оператор підстановки або Composition operator). Визначимо функцію від m змінних $f(x_1, x_2, \ldots, x_m)$та впорядкований набір функцій від змінної n

$g_1(x_1, x_2, \ldots, x_n), g_2(x_1, x_2, \ldots, x_n), \ldots, g_m(x_1, x_2, \ldots, x_n)$. Суперпозицією функцій g_k в функцію f називається така функція від n змінних, яка співставляє будь-якому впорядкованому набору натуральних чисел наступну послідовні $h(x_1, x_2, \ldots, x_n) = f(g_1(x_1, x_2, \ldots, x_n), \ldots, g_m(x_1, x_2, \ldots, x_n))$. Наприклад, суперпозицією функцій $o(x)$ та $I_m^n(x_1, x_2, \ldots, x_n) = x_m$ буде функція $o(x_1, x_2, \ldots, x_n) =$ 0або ж, якщо розписати, то $o(x_1, x_2, \ldots, x_n) = o(I_m^n(x_1, x_2, \ldots, x_n))$.

Операція примітивної рекурсії (Primitive recursion operator). Ця операція будує функцію від *n+1* аргумента, якщо визначено дві числові функції $g(x_1, x_2, \ldots, x_n)$ та $h(x_1, x_2, \ldots, x_n, x_{n+1}, x_{n+2})$. Отже, функція *f*, яку ми хочемо побудувати буде базуватись на двох функція: *g*, область визначення якої є на одиницю меншою за область визначення функції *f* та *h*, область визначення якої є на одиницю більшою. Оператор примітивної рекурсії позначається як $f = R(g, h)$і визначається наступним чином:

$$f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n)$$
$$f(x_1, x_2, \ldots, x_n, y + 1) = h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y))$$

Змінна *y* тут розуміється як лічильник ітерації, функція *g* – вихідна функція, яка видає деяку послідовність функції від *n* змінних, а *h* — як оператор, який на вході приймає *n* змінних $x_1, x_2, \ldots, x_n$, номер кроку ітерації *y,* та функцію $f(x_1, x_2, \ldots, x_n, y)$і обраховує значення функції на наступному кроці ітерації. Як найпростіший приклад розглянемо функцію множення $f_x(x, y) = xy$, яка є примітивно рекурсивною, що можна показати так:

$$f_x(x, 0) = 0$$
$$f_x(x, y + 1) = f_x(x, y) + x = f_x(x, f_x(x, y))^{\cdot}$$

Операція мінімізації (Minimisation operator. або ж μ-оператор). Припустимо нам задана числова функція $f(x_1, x_2, \ldots, x_{n-1}, x_n, y)$ у якій значення *n* є більше або рівне нуля. Аргументи $x_1, x_2, \ldots, x_{n-1}$вважаються зафіксованими. Після цього розглянемо рівняння $f(x_1, x_2, \ldots, x_{n-1}, y) = x_n$, де $y = \overline{0, n}$, тобто знаходиться на множині всіх натуральних цілих чисел включно з нулем. Почергово підставляючи *y* в функцію, ми можемо визначити таке мінімальне значення *y,* яке не дорівнює x_n, для якого буде справедлива рівність $g(x_1, x_2, \ldots, x_n) = y$. Мінімальне значення з множини визначення функції *y* позначають μ, звідки й пішла назва оператора. Кінцеву рівність записують як $g(x_1, x_2, \ldots, x_{n-1}, x_n) = \mu_y[f(x_1, x_2, \ldots, x_{n-1}, y) = x_n]$ і кажуть, що функція $g(x_1, x_2, \ldots, x_n) = y$одержана з функції $f(x_1, x_2, \ldots, x_{n-1}, y) = x_n$за допомогою оператора мінімізації за змінною x_n.

Функція $f(x_1, x_2, \ldots, x_{n-1}, x_n, y)$може бути невизначеною, якщо не існує такого *y,* яке набуває мінімального значення і водночас не дорівнює x_n.

Якщо функція може бути записана за допомогою простих функцій, які були приведені вище, та з використанням скінченної кількості операцій суперпозиції та примітивної рекурсії, то вона називається *примітивно-рекурсивною*. У випадку, якщо функція може бути записана з використанням двох попередніх операцій та операцією мінімізації, то вона називається *частково рекурсивно*. Частково рекурсивні функції можуть бути невизначеними для деяких значень аргументів, оскільки їхня мінімізація не завжди визначена коректно, у зв'язку з тим, що функція *f* може бути не рівна нулю для усіх значень аргумента. Коли частково рекурсивна функція визначена на усій множині значень, то вона називається *загальнорекурсивною*.

Рекурсивні функції набули поширення у програмуванні. З такими функціями треба поводитись обережно, бо при неправильному заданні умови можна створити безкінечний цикл. Та все ж при розв'язуванні деяких типів задач, рекурсія є, іноді, єдиним виходом. Для прикладу функція обчислення факторіалу:

```
int factorial (int n)
{
	return (n > 0) ? n * factorial (n – 1) : 1;
}
```

де функція factorial викликає сама себе доти, доки змінна *n* не буде дорівнювати нулю.

Ще одним прикладом може бути функція швидкого сортування. Тут зручно використовувати одну і ту ж функцію, яка базується на попередніх обчислених результатах.

```
void QuickSort ( int array[], int left, int right)
{
	int I = left, j = right, tmp;
	int base = array [(left + right) / 2];

	while (i <=  j)
	{
		while (array[i] < base) i++;
		while (array[j] > base) j--;
		if (i <= j)
		{
			tmp = array[i];
			array[i++] = array[j];
			array[j- -] = tmp;
		}
	}
			if (left < j) QuickSort(array, left, j);
	if (i < right) QuickSort(array, i, right)
}
```

Функція QuickSort обирає базовий елемент серед заданих, ділячи таким чином дані на дві частини — праву та ліву і сортуючи дані методом порівняння їх з базовим. Якщо елемент менший від базового — ставимо його ліворуч від нього, якщо більший — справа. Посортувавши так дві частини ми, знову викликаємо ту ж функцію, яка тепер буде застосовуватись до обох сортованих частин і так до тих пір, поки не буде відсортовано весь масив даних.

Тут рядки if “(left < j) QuickSort(array, left, j);” та “if (i < right) QuickSort(array, i, right)” означають рекурсивний виклик функції всередині самої себе для повторення вищеозначених дій вже для менших послідовностей чисел. Виклик функції продовжується доти, доки не буде відсортовано найменшу послідовність чисел.

У світі математики рекурсія має більш вражаючий вигляд, оскільки за її допомогою представлені деякі класи дуже цікавих об'єктів.

Теза Чорча. В середині 30-х років XX століття, А. Чорч дійшов висновку, що увійшов у історію як Теза Чорча: Числова функція алгоритмічно обчислювана тоді і лише тоді , коли вона частково рекурсивна. Функція називається частково рекурсивною, якщо отримана із вказаних функцій застосуванням скінченної кількості операцій суперпозиції, примітивної рекурсії та мінімізації. Всюди визначена частково рекурсивна функція називається загальнорекурсивною.

Нехай функція частково рекурсивна. Потрібно довести, що вона є обчислюваною за Тьюрінгом. Спочатку доводять, що найпростіші функції обчислювані за Тьюрінгом (це доволі очевидно), а потім, що оператори суперпозиції, примітивної рекурсії та мінімізації зберігають обчилюваність. Нехай функція є обчислюваною за Тьюрінгом. Потрібно довести, що вона частково рекурсивна. Таке твердження може видатись дуже слабким для того, щоб говорити про еквівалентність рекурсивних функцій, які мають справу з невід'ємними цілими числами, і машин Тьюрінга, які можуть не тільки обчислювати. Проте, від будь-яких об'єктів можна перейти до числових за допомогою кодування, причому це кодування виявляється вкрай простим. З точки зору машини, яка не вкладає в символи ніякого змісту, а лише виконує з ними задані операції, немає особливої різниці між числами й “нечислами”. Свої елементарні дії (читання, зсув, запис, зміну стану) машина може виконувати за наявності на стрічці як цифр, так і інших символів. Відмінність існує

лише для нас в інтерпретації отриманих результатів. Зокрема, ніщо не перешкоджає інтерпретувати будь-який зовнішній алфавіт (алфавіт стрічки) $A = \{a_1, a_2, \ldots, a_m\}$ як множину цифр *m-кової* системи числення, а слово, написане на стрічці, - як *m-кове* число. Тоді будь-яку роботу машини Тьюрінга розглядають як перетворення чисел, тобто як обчислення числової функції. Саме така інтерпретація тут і береться до уваги.