Математична криптологія

Алгоритми та їх складність

Задачі, алгоритми і складність

Під ***масовою задачею*** (або просто ***задачею***) будемо розуміти деяке загальне питання, на яке потрібно дати відповідь. Задача містить декілька ***параметрів*** або вільних змінних, конкретні значення яких не визначені.

Задача **П** визначена такою інформацією:

1. загальним списком усіх її параметрів;
2. формулюванням тих властивостей, які повинна задовольняти відповідь або, іншими словами, розв’язок задачі.

Індивідуальну задачу І одержують з масової задачі **П**, якщо всім параметрам задачі **П** присвоїти конкретні значення.

Масову задачу можна вважати нескінченим набором індивідуальних задач.

Класична задача комівояжера

Параметри задачі:

- скінченна множини міст $C=\{c_1,c_2,..,c_m\}$
- відстаней $d(c_i, c_j) \in Z^+$ для кожної пари міст c_i, c_j з C.

Чи існує замкнутий маршрут, що проходить через кожне місто лише один раз і довжина цього маршруту мінімальна?

Чи існує перестановка $\pi=(c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(m)})$, така, що мінімізує величину:

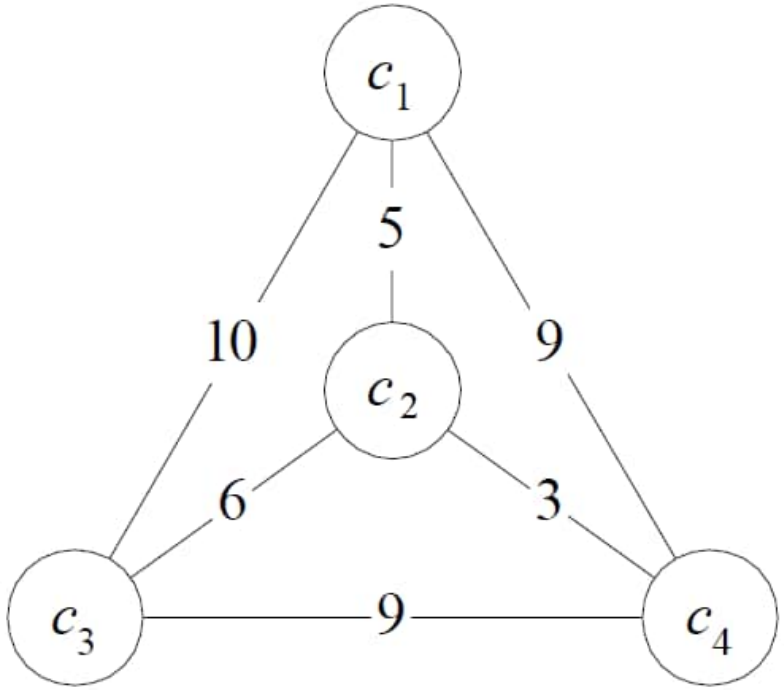
$$\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(m)}, c_{\pi(1)})$$

Останнє є довжиною маршруту, що починається в місті $c_{\pi(1)}$, проходить послідовно через всі міста і повертається в $c_{\pi(1)}$, безпосередньо з останнього міста $c_{\pi(m)}$.

Індивідуальна задача комівояжера

- Параметри задачі:
- $C=\{c_1, c_2, c_3, c_4\}$
- $d(c_1, c_3)=10$, $d(c_1, c_2)=5$, $d(c_1, c_4)=9$, $d(c_2, c_4)=9$, $d(c_2, c_3)=6$, $d(c_2, c_4)=3$

Послідовність $\{c_1, c_2, c_4, c_3,\}$ є розв'язком задачі, оскільки відповідний маршрут має найменшу можливу довжину, яка дорівнює 27.



Поняття алгоритму

- Під ***алгоритмом*** будемо розуміти загальну, здійснювану крок за кроком процедуру розв'язування задачі. З метою визначеності можемо вважати її програмою для комп'ютера, що написана формальною мовою.
- Кажуть, що алгоритм *розв'язує* масову задачу ***П***, якщо він застосовний **до будь-якої індивідуальної задачі *І*** із ***П*** і **обов'язково** дає розв'язок задачі ***І***.
- **Час роботи алгоритму** зручно виражати у вигляді функції однієї змінної. Ця єдина змінна характеризує розмір індивідуальної задачі, тобто об'єм вхідних даних, які потрібні для опису цієї задачі. У подальшому порівняльну складність задачі будемо оцінювати через її розміри.
- Опис індивідуальної задачі, який ми даємо в термінах входу для комп'ютера, можна розглядати як один скінченний ланцюжок (або слово) символів зі скінченного вхідного алфавіту.

Вхідні параметри алгоритму

- Входом задачі може служити об’єкт, до якого належить граф, матриця, множина, перестановка тощо.
- Для введення такого об’єкту в комп’ютер необхідно яким-небудь чином його закодувати – представити послідовністю символів над деяким фіксованим алгоритмом. Наприклад, у вигляді двійкового алфавіту {0, 1}.
- Розглядаючи вхід алгоритму як послідовність (ланцюжок) символів, визначимо розмір входу як довжину цієї послідовності, що дорівнює кількості символів, які містяться в ній.
- Будемо вважати, що з кожною масовою задачею пов’язана деяка фіксована ***схема кодування***, яка відображає індивідуальні задачі у відповідні ланцюжки символів. Вхідну довжину індивідуальної задачі ***I*** із ***П*** визначають як кількість символів у ланцюжку, отриманому застосуванням до задачі ***I*** схеми кодування для масової задачі ***П***.

Приклад

- *Задача визначення чи є задане ціле число простим.*

Деяке додатне число записане послідовністю цифр:

$$\overline{d_k d_{k-1} \ldots d_1},\ 0 \le d_i \le 9,$$

це дорівнює:

$$d_k 10^{k-1} + d_{k-1} 10^{k-2} + \ldots + d_1.$$

Числа записуються у *десятковій системі числення*.

Число, записане у *двійковій системі числення* послідовністю цифр, має такий вигляд:

$$\overline{b_m b_{m-1} \ldots b_1},\ 0 \le b_i \le 1,$$

дорівнює

$$b_m 2^{m-1} + b_{m-1} 2^{m-2} + \ldots + b_1.$$

Приклад

Представимо десяткове число 18 у двійковій системі числення:

$$18=9*2+0$$
$$9=4*2+1$$
$$4=2*2+0$$
$$2=1*2+0$$
$$1=0*2+1.$$

Одержана послідовність 10010.

Дійсно $1*2^4+0*2^3+0*2^2+1*2^1+0*2^0=18$.

Таким чином, якщо **входом алгоритму** є число 18,

закодоване послідовністю 10010,

то **розмір входу дорівнює 5.**

Складність алгоритму

- **Складність алгоритмів вимірюється** за необхідними ресурсами, в основному це тривалість обчислень або необхідний обсяг пам’яті.
- **Зазвичай під найефективнішим алгоритмом розуміється найшвидший.**
- **Логічна складність** — кількість людино-місяців, витрачених на створення алгоритму.
- **Статична складність** — довжина опису алгоритмів (кількість операторів).
- **Часова складність** — час виконання алгоритму.
- **Ємнісна складність** — кількість умовних одиниць пам'яті, необхідних для роботи алгоритму.
- *Головне питання теорії складності обчислень*: які задачі розв’язуються ефективно, а які ні. Під ефективність тут розуміється як швидкість знаходження відповіді так і витрата невеликого обсягу ресурсів.
- Час роботи алгоритму зручно виражати у вигляді функції від однієї змінної, що характеризує розмір індивідуальної задачі - обсяг вхідних даних, необхідних для опису задачі.

Часова складність алгоритму

- **Часова складність алгоритму** відображає потрібні для його роботи витрати часу.
- Це функція, яка кожній вхідній довжині n ставить у відповідність **максимальний** (по всім індивідуальним задачам даної довжини) **час, що витрачається алгоритмом на розв'язання індивідуальних задач цієї довжини**.
- Ця функція не буде повністю визначена до тих пір, поки **не зафіксована схема кодування**, яка визначає **вхідну довжину індивідуальної задачі** і **не вибрано обчислювальний пристрій (або його модель), що визначає час роботи.**

Часова складність алгоритму

- **Часова складність** часто оцінюється шляхом обчислення числа елементарних операцій, що здійснює алгоритм, де елементарна операція займає для виконання фіксований час.
- Оскільки продуктивність алгоритму може відрізнятися при входах одного і того ж розміру, як правило використовується **тимчасова складність** найгіршого випадку поведінки алгоритму, яка позначається як $T(n)$ і яка визначається як максимальний час, який потрібен для будь-якого входу довжини n.
- **Складність** – це функція від довжини входу, значення якої дорівнює максимальної (за будь-якими входами даної довжини) кількості операцій, необхідних алгоритму для отримання відповіді.

Складність алгоритму в гіршому та середньому випадках

- **Для вирішення задачі для двох вхідних даних *x* та *y* однакового розміру (|*x*|=|*y*|) алгоритм може витрачати різний час.**
- Тому для отримання оцінок часової складності в залежності від розміру задачі визначають її як *максимальний час*, що витрачається алгоритмом для вхідних даних довжини *n:*

$$T(n) = \max\{T(x): |x| = n\}.$$

- Ця функція називається **складністю алгоритму в гіршому випадку**.

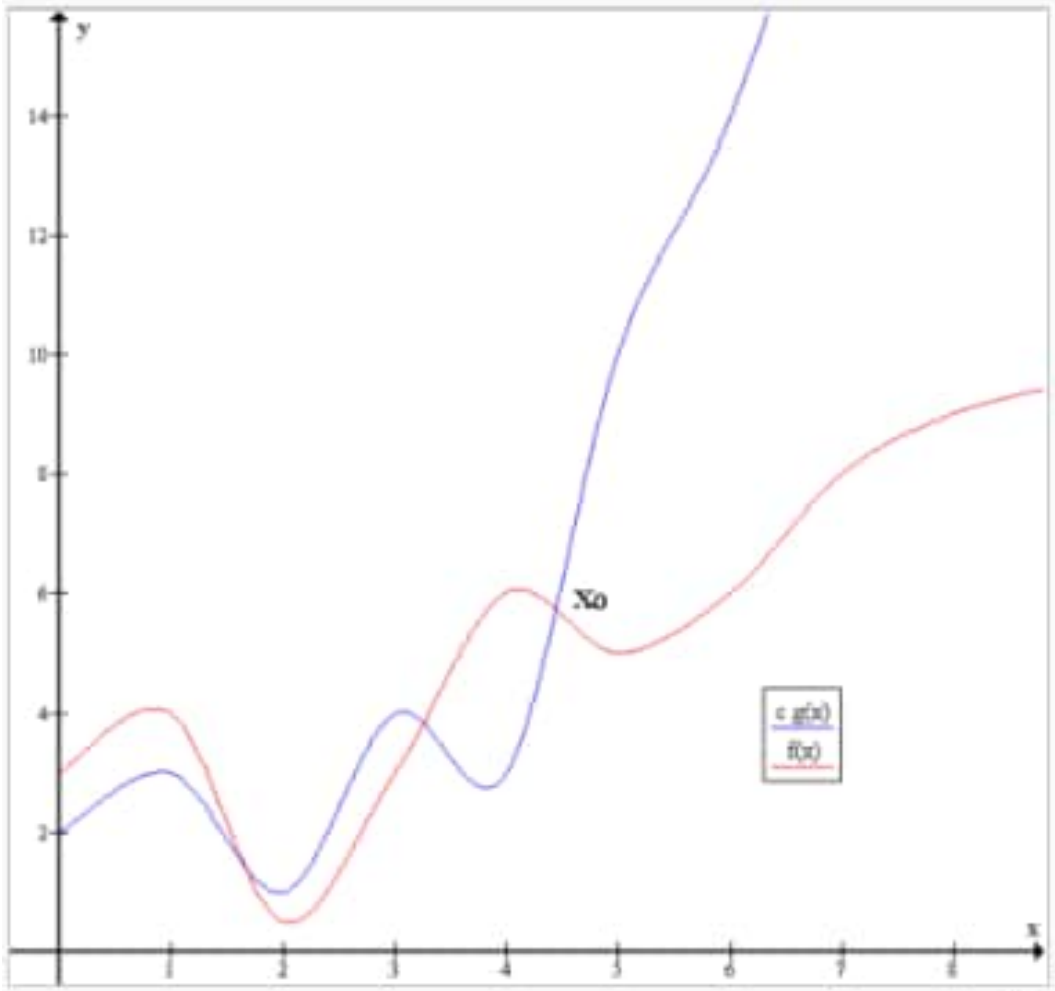
- Формально **складність в середньому порядку** визначається так:

$$T_{ср}(n) = \Sigma T(x)p(x)$$

- де $p(x)$ – ймовірність появи вхідних даних x, а підсумовування ведеться за всіма можливими вхідними даними розміру n.
- **При оцінці складності алгоритму зазвичай розглядають його складність в гіршому випадку.**
- **Оцінюють не точне значення функції складності T(n), а порядок зростання цієї функції, знаходять таку функцію $f(n)$, що $T(n)=O(f(n))$ при $n \to \infty$.**

Нотація Ландау

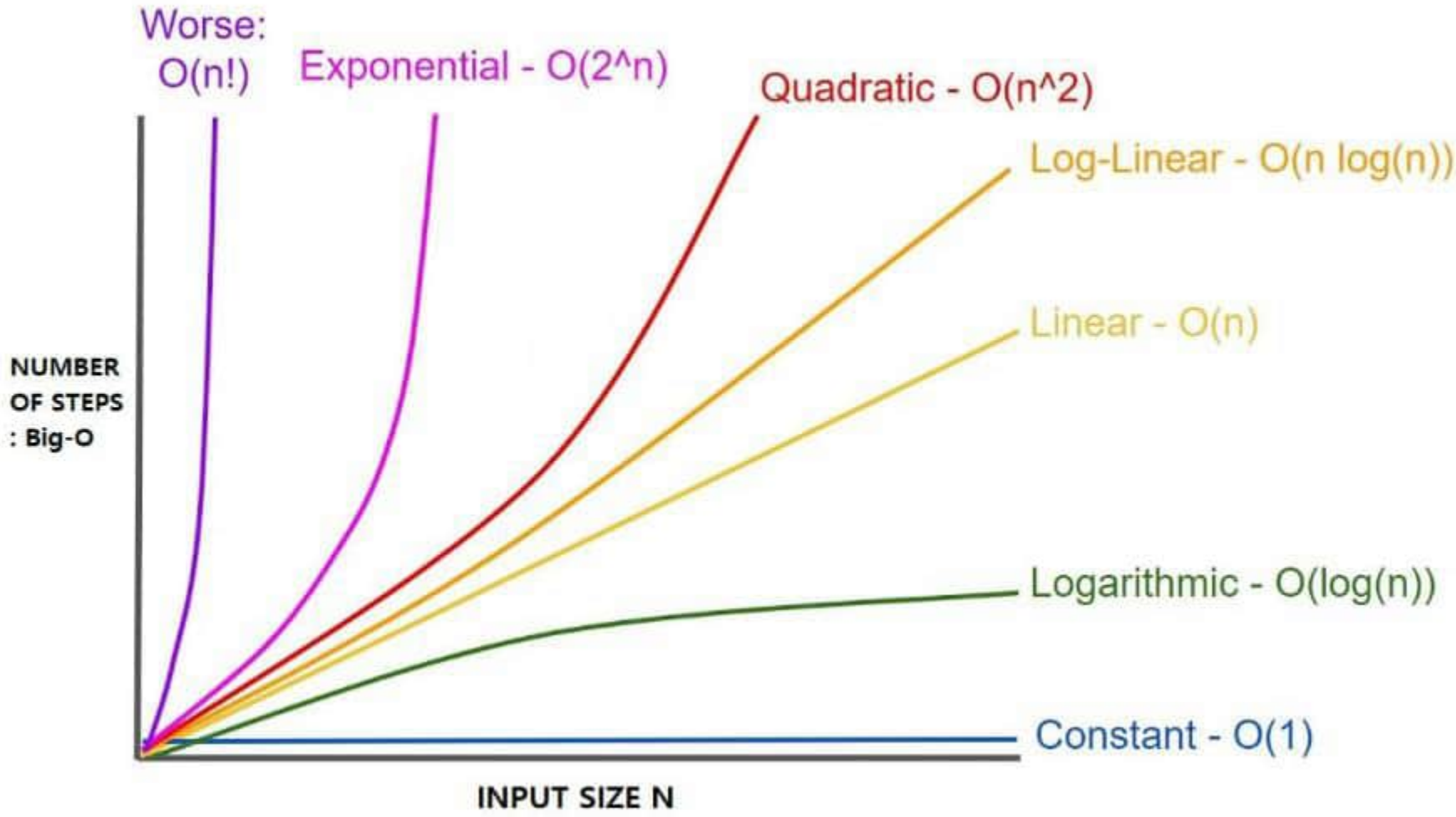
- **Асимптотична нотація великого *O***, відома також як **нотація Ландау** — поширена математична нотація для формального запису асимптотичної поведінки функцій. Широко вживається в теорії складності обчислень, інформатиці та математиці.
- Функція з невід'ємними значеннями $f(x)=O(g(x))$, якщо існують додатна константа M і натуральне x_0 такі, що $f(x) \leq Mg(x)$ для всіх значень $x \geq x_0$.



Приклад використання нотації О-велике:
$f(x) \in O(g(x))$, бо існують $M > 0$ (наприклад, $M = 1$) та x_0 (наприклад, $x_0 = 5$), такі, що $f(x) \leqslant Mg(x)$ для кожного $x \geqslant x_0$.

Деякі важливі класи відношень

нотація	назва
$O(1)$	константа
$O(\log n)$	логарифмічне
$O([\log n]^c)$	полілогарифмічне
$O(n)$	лінійне
$O(n \cdot \log n)$	лінеарітмічне
$O(n^2)$	квадратичне
$O(n^c)$	поліноміальне
$O(c^n)$	експоненціальне
$O(n!)$	факторіальне



Асимптотична складність

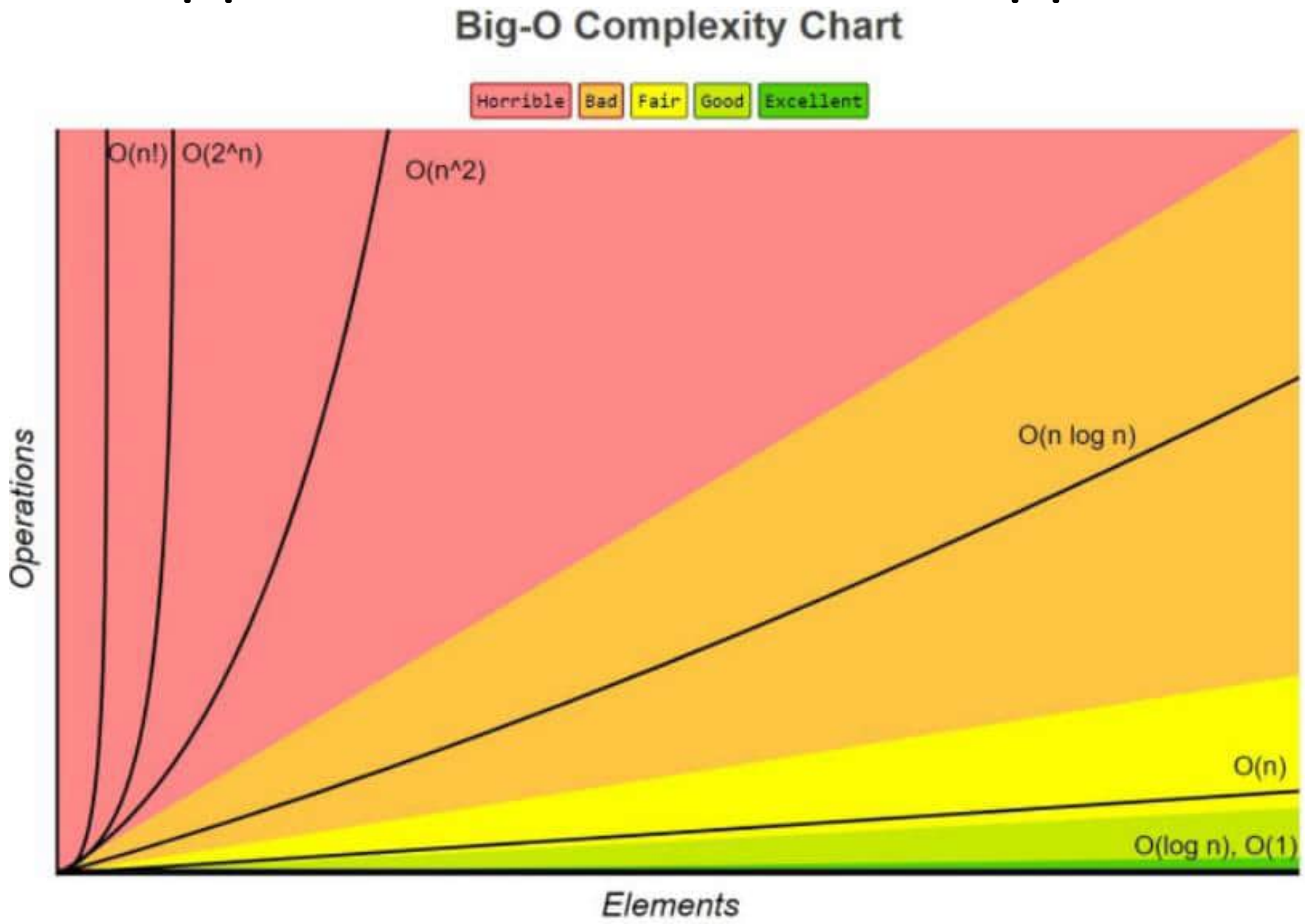
Позначення	Інтуїтивне пояснення	Визначення
$f(n) \in O(g(n))$	f обмежена згори функцією g (з точністю до постійного множника) асимптотично	$\exists(C>0), n_0 : \forall(n>n_0)\ \lvert f(n)\rvert \le C\lvert g(n)\rvert$ або $\exists(C>0), n_0 : \forall(n>n_0)\ f(n) \le Cg(n)$
$f(n) \in \Omega(g(n))$	f обмежена знизу функцією g (з точністю до постійного множника) асимптотично	$\exists(C>0), n_0 : \forall(n>n_0)\ \lvert f(n)\rvert \ge C\lvert g(n)\rvert$
$f(n) \in \Theta(g(n))$	f обмежена знизу і згори функцією g асимптотично	$\exists(C, C'>0), n_0 : \forall(n>n_0)\ C\lvert g(n)\rvert \le \lvert f(n)\rvert \le C'\lvert g(n)\rvert$
$f(n) \in o(g(n))$	g домінує над f асимптотично	$\forall(C>0)\exists n_0 : \forall(n>n_0)\ \lvert f(n)\rvert < C\lvert g(n)\rvert$
$f(n) \in \omega(g(n))$	f домінує над g асимптотично	$\forall(C>0)\exists n_0 : \forall(n>n_0)\ \lvert f(n)\rvert > C\lvert g(n)\rvert$
$f(n) \sim g(n)$	f еквівалентна g асимптотично	$\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 1$

Big-O Notation	Comparison Notation	Limit Definition
$f \in o(g)$	$f \lessdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
$f \in O(g)$	$f \lesssim g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$
$f \in \Theta(g)$	$f \eqsim g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} \in \mathbb{R}_{>0}$
$f \in \Omega(g)$	$f \gtrsim g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$
$f \in \omega(g)$	$f \gtrdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

Приклади асимптотичних складностей

Складність	Коментар	Приклади
$O(1)$	Сталий час роботи не залежно від розміру задачі	Очікуваний час пошуку в хеші
$O(\log \log n)$	Дуже повільне зростання необхідного часу	Очікуваний час роботи інтерполюючого пошуку n елементів
$O(\log n)$	Логарифмічне зростання — подвоєння розміру задачі збільшує час роботи на сталу величину	Обчислення x^n; двійковий пошук в масиві з n елементів
$O(n)$	Лінійне зростання — подвоєння розміру задачі подвоїть і необхідний час	Додавання/віднімання чисел з n цифр; лінійний пошук в масиві з n елементів
$O(n \log n)$	Лінеаритмічне зростання — подвоєння розміру задачі збільшить необхідний час трохи більше ніж вдвічі	Сортування злиттям або купою n елементів; нижня границя сортування порівнянням n елементів
$O(n^2)$	Квадратичне зростання — подвоєння розміру задачі вчетверо збільшує необхідний час	Елементарні алгоритми сортування
$O(n^3)$	Кубічне зростання — подвоєння розміру задачі збільшує необхідний час у вісім разів	Звичайне множення матриць
$O(c^n)$	Експоненціальне зростання — збільшення розміру задачі на 1 призводить до c-кратного збільшення необхідного часу; подвоєння розміру задачі підносить необхідний час у квадрат	Деякі задачі комівояжера, алгоритми пошуку повним перебором

Приклади асимптотичних складностей



Поліномний та експоненціальний алгоритми

- **Поліномний алгоритм** (або алгоритм поліномної часової складності) – це алгоритм, у якого часова складність дорівнює $O(p(n))$. Тут $p(n)$ – деяка поліномна функція, а n – вхідна довжина.
- Алгоритми, часова складність яких не піддається подібній оцінці, називають ***експоненціальними***. Більшість експоненціальних алгоритмів – це просто варіанти повного перебирання, тоді як поліномні алгоритми звичайно можна побудувати тільки тоді, коли вдається заглибитись у суть задачі. Задачу називають важкорозв'язною, якщо для її розв'язування не існує поліномного алгоритму.
- Зазначимо, що є експоненціальні алгоритми, які добре зарекомендували себе на практиці. Річ у тому, що часова складність означена нами як міра поведінки алгоритму у найгіршому випадку. Насправді може виявитись, що більшість індивідуальних задач потребують для розв'язування значно менше часу. Наприклад алгоритм гілок і меж настільки успішно розв'язує задачу про наплічник, що багато дослідників уважають її "добре розв'язною", хоча алгоритми гілок і меж мають експоненціальну часову складність.

Класи складності задач

- **Клас складності *NP* (Non-deterministic Polynomial time)** клас складності, до якого належать задачі, що можна розв'язати недетермінованими алгоритмами за поліноміальний час; тобто, недетермінованими алгоритмами в яких завжди існує шлях успішного обчислення за поліноміальний час відносно довжини вхідного рядка. **Приклади** : задача пакування рюкзака, задача комівояжера, розфарбування графа у три кольори, розкладання числа на прості множники.

- **Клас складності *P* (Polynomial time)** клас задач, що можна розв'язати алгоритмами з поліноміальним часом. **Приклад**: стандартне множення матриць.
- **Алгоритмом з поліноміальним часом** називається такий алгоритм, час роботи якого (тобто, кількість елементарних двійкових операцій, необхідних для його виконання на детермінованій машині Тюринга) на вхідному рядку довжиною l обмежено згори деяким поліномом $P(l)$.

$\mathbf{P \subseteq NP}$

Класи складності *P* та *NP*

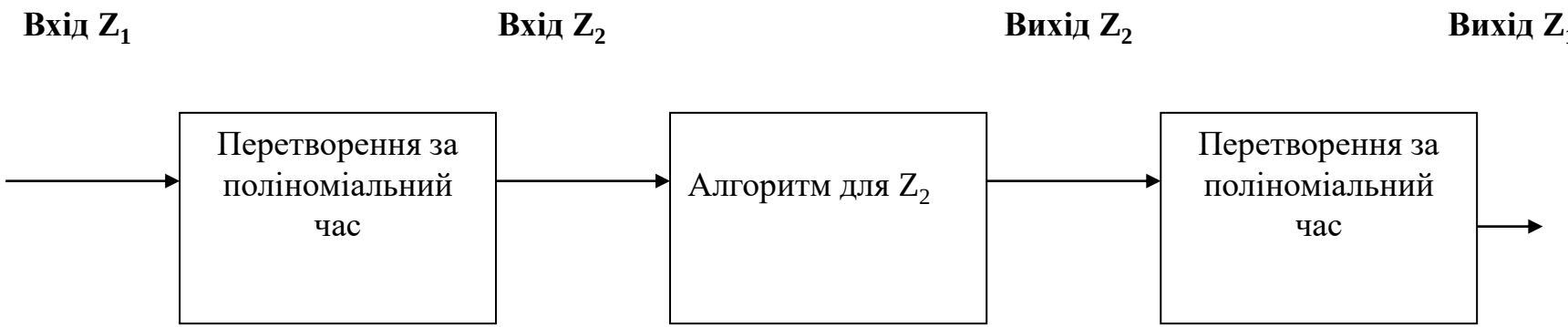
- **Центральним в теорії зведеності є питання про те, чи збігаються класи *P* і *NP*.**
- Гіпотеза ***P≠NP*** **породжує істотну різницю між класами *P* і *NP\P*.**
- Всі задачі з ***P*** **можуть бути розв'язані поліномними алгоритмами.**
- У той час як всі задачі ***NP\P*** **належать до складно розв'язуваних,** тобто до таких для яких **не існує ефективних алгоритмів розв'язання.**
- До тих пір, поки **нерівність *P≠NP* залишається гіпотезою, немає підстав вважати, що задача *Z*, яка не піддається ефективному розв'язанню, належить до *NP\P*.**

якщо *P≠NP*, то *Z∈NP\P*

Поліномна зведеність

- Задача $Z1$ перетвориться в задачу $Z2$, якщо будь-яка індивідуальна задача $I \in Z1$ перетворюється за поліномний час в деяку індивідуальну задачу $J \in Z2$, тобто розв'язок задачі $Z1$ можна отримати з розв'язку задачі $J \in Z2$ за поліномний час.
- Кажуть, що $Z1$ поліномно зводиться до задачі $Z2$, і позначають.

$$Z1 \propto Z2.$$



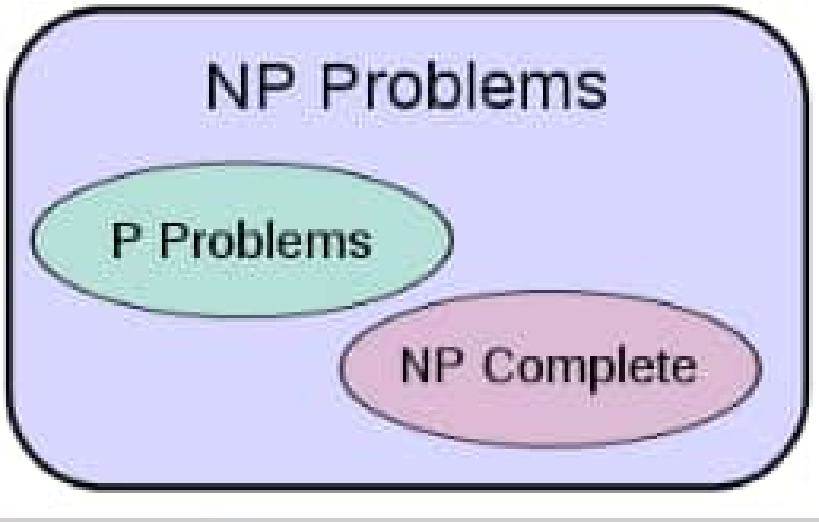
NP-повні задачі

- Започаткував теорію NP-повних задач С. Кук у праці “Складність процедур виведення теорем” опублікованій 1971 р. Він довів важливість поняття “звідність за поліномний час”, тобто звідність, яку алгоритм виконує з поліномною часовою складністю. Якщо одна задача зводиться за поліномний час до іншої, то будь-який поліномний алгоритм розв’язування другої задачі може бути перетворений у поліномний алгоритм розв’язування першої.
- У класі *NP* виявлені універсальні або *NP*-повні задачі, до яких поліномно зводиться будь-яка задача з *NP*. У цьому сенсі *NP*-повні задачі дають еталон складності.
- Задача Z називається *NP*-повною, якщо $Z \in NP$ і будь-яка інша задача $Z' \in NP$ зводиться до Z.
- *NP*-повна задача розглядається як сама важка задача з *NP*.
- Якщо існує поліноміальний алгоритм хоча б для однієї *NP*-повної задачі, то і всі задачі з *NP* можуть бути розв’язані за поліноміальний час.

Твердження поліноміальної зведеності

- якщо Z1∞ Z2, то з Z2∈*P* випливає, що Z1∈*P*;
- якщо Z1∞ Z2 і Z1 ∈ *P*, то Z2 ∈ *P*;
- якщо Z1∞ Z2 і Z2∞ Z3, то Z2∞ Z3;
- якщо Z1∈*NP*, Z2∈*NP*, задача Z1 *NP*-повна і Z1∞ Z2, то задача Z2 *NP*-повна.

- Співвідношення між класами задач *P* та *NP*



Діаграма Венна відношення між класами складності задач (у випадку вірності гіпотези P ≠ NP).

Доведення *NP*-повноти задачі

- Шлях доведення *NP*-повноти нової задачі Z. Для того, щоб встановити *NP*-повноту задачі Z, необхідно довести, що:

  - $Z \in NP$;

  - відома *NP*-повна задача Z' зводиться до Z.

- Перед тим, як застосовувати таку схему доведення, нам необхідна задача Z', про яку відомо, що вона *NP*-повна.

Основне призначення теорії *NP*-повних задач

- Основне призначення теорії *NP*-повних задач в тому, щоб допомогти розробникам алгоритмів і направити їх зусилля на вибір таких підходів до вирішення задач, які приведуть до практично корисних алгоритмів.

Чому ж *NP*-повні задачі представляють інтерес?

- По-перше, не зважаючи на те, що до сих пір не знайдені ефективні алгоритми їх розв’язання, також не доведено, що такого алгоритму не існує.
- По-друге, набору *NP*-повних задач притаманна така властивість, якщо ефективний алгоритм існує хоча б для однієї з таких задач, то його можна сформулювати і для усіх інших.
- По-третє, деякі *NP*-повні задачі схожі (але не ідентичні) на задачі, для яких вже відомі ефективні алгоритми.

Односпрямовані функції

- Базовим поняттям криптографії з відкритим ключем є поняття односпрямованої функції (one-way function). За заданим аргументом $x \in X$ нескладно обчислити значення цієї функції $F(x)$, тоді як визначити x з $F(x)$ є надто складно, тобто немає алгоритму для розв'язування цього завдання з поліноміальним часом роботи.

- ***Односпрямованою*** називається функція $F(x)$: $X \rightarrow Y$, $x \in X$, яка має дві властивості:
  – існує поліномний алгоритм обчислення значень $y = F(x)$;
  – не існує поліноміального алгоритму інвертування функції $F(x) = y$.

Прикладом кандидата на назву односпрямованої функції є модульне піднесення до степеня, тобто функція $F(x) \equiv a^x \bmod p$, де a – примітивний елемент поля $GF(p)$; p – велике просте число.