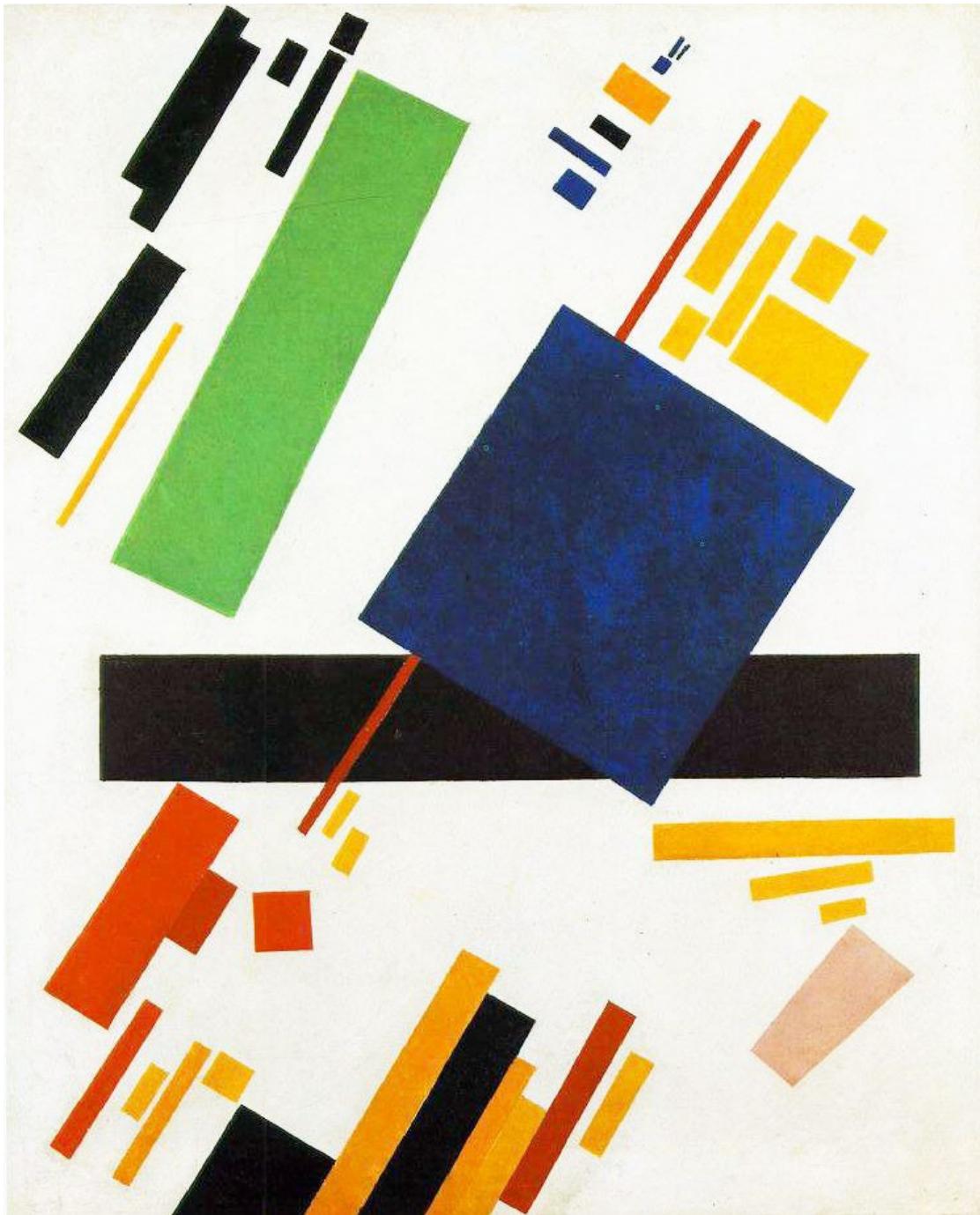


Комп'ютерна наука в питаннях та відповідях

(компілятор Дмитро Попов)

2024



Зміст

Про книгу.....	8
Що таке Комп'ютерна наука (інформатика)?.....	9
Що таке математика?.....	10
Що таке множина?.....	11
Що таке функція?.....	17
Що таке бієкція?.....	21
Що таке аксіома вибору?.....	22
Що таке математична індукція?.....	24
Які є види чисел?.....	26
Які основні арифметичні формули?.....	28
Визначення операцій для натуральних чисел (N).....	31
Модульна арифметика.....	39
Що таке алгоритм?.....	40
Що не може бути описане алгоритмом?.....	42
Що таке складність алгоритму?.....	44
Що таке логарифм?.....	47
Що таке моном і поліном?.....	48
Що означає $P = NP$?.....	49
Що таке жадібний алгоритм?.....	55
Яка мова слугує для опису алгоритмів?.....	57
Який мінімальний набір функцій для опису будь-якого алгоритму?.....	60
Що таке лямбда-числення?.....	63
Що таке комп'ютерна програма?.....	66
Що таке універсальний комп'ютер?.....	67
Що таке аналогова обчислювальна машина?.....	68
Що таке машина Тюрінга?.....	69
Що таке клітинний автомат?.....	82
Візерунки Тюрінга.....	85
Що не може універсальний комп'ютер?.....	87
Що таке Рівнодоступна адресна машина?.....	96
Що таке архітектура фон Неймана?.....	99
Що таке Гарвардська архітектура?.....	101
Що таке архітектура програмного забезпечення?.....	104
Що таке булева алгебра?.....	105
Що таке двійкова система числення?.....	108
Числовий автомат “ $1 \leftarrow 2$ ”.....	111
Що таке стандарт IEEE 754?.....	112
Що таке шістнадцяткова система числення?.....	114
Що таке десяткова система числення?.....	115
Що таке експоненційний запис?.....	117
Що таке ASCII та Unicode?.....	118
Що таке об'єм пам'яті?.....	119
Інформаційна ентропія.....	120
Що таке логічні вентилі та напівсуматор?.....	121
Що таке операційна система?.....	128
Що таке процес в операційній системі?.....	132
Що таке проблема обідаючих філософів?.....	133
Що таке мережі Петрі?.....	139

Що таке віртуальна машина?	141
Що таке мова програмування?	143
Що таке компілятор?	146
Що таке інтерпретатор?	147
Що таке функціональна мова програмування?	148
Функтор	149
Що таке об'єктно-орієнтована мова програмування?	152
Що таке шаблон Спостерігач (Observer)?	162
Що таке шаблон Декоратор?	164
Що таке система типів?	166
Що таке TypeScript?	168
Що таке Цикл подій в JavaScript?	175
Що таке граматика?	177
Граматика англійської мови	180
Що таке контекстно-вільна граматика?	184
Що таке нотація Бекуса-Наура?	185
Форма МакКімена	187
Що таке Абстрактне синтаксичне дерево?	190
Що таке польська нотація?	192
Що таке регулярний вираз?	193
Що таке скінчений автомат?	196
Що таке логіка предикатів?	197
Що таке комп'ютерний процесор і як його виготовляють?	204
Логічні вентилі.	206
Піни процесора	207
Метод Чохральського	209
Планарна технологія	211
Електронні компоненти	213
Що таке реляційні бази даних?	217
SQL запити	221
Що таке модель “сутність — зв'язок”?	224
Що таке теорема CAP?	226
Що таке блок-схема?	227
Що таке комп'ютерна мережа?	230
Код Морзе	230
Кількість унікальних можливих з'єднань у мережі	233
Протокол зв'язку	234
TCP/IP	236
JSON	245
HTML	247
Стільникова мережа	249
Теорема відліків	252
Теорема Шеннона — Гартлі про пропускну здатність	254
Коди Гемінга	256
Код Грэя	258
Алгоритм Гаффмана	259
Кодування довжин серій	262
Які принципи розробки інтерфейсу користувача (UI/UX)?	263
Закон Міллера	269
Чотири принципи доступності (Accessibility)	270

"Надлишкове число"	12.....	271
Що таке штучний інтелект?	272
Гра в імітацію (Тест Тюрінга).....		278
Млин Лейбніца.....		279
Мозок людини.....		280
Оптичні ілюзії.....		282
Закон Вебера — Фехнера.....		287
Лінійний градієнт.....		287
Розмивання Гауса.....		290
Простий метод знаходження контурів зображення.....		294
Що таке нейрона мережа?	295
Персепtron Розенблата.....		296
Лінійна регресія.....		299
Функція Хевісайда.....		299
Сигмоїда.....		300
DBSCAN.....		301
Що таке Перенавчання?		303
Що таке згорткова (конволюційна) нейронна мережа?	305
Які принципи розробки продукту?	310
Які принципи декомпозиції задачі?	311
Як проєктувати продукт?	312
Про які тести має піклуватися саме розробник?	314
Як тестувати штучний інтелект (AI) ?	320
Що таке якісний код (\approx чистий код)?	322
Які є принципи налагодження програм (debugging)?	324
Як встановлювати версії продукту?	326
Як оцінювати якість продукту?	330
Що таке ринкова економіка й відповідність зарплати?	335
Хто такий технічний лідер в команді?	338
Що таке Scrum?	344
Методологія Водоспад та V-модель.....		350
Екстремальне програмування.....		352
Цикл розробки програмного забезпечення.....		355
Що таке Неперервна інтеграція (CI/CD)?	356
Які є структури даних?	360
Хеш-таблиці.....		361
Які існують алгоритми сортування масивів?	364
Бульбашкове сортування.....		365
Сортувати вибором.....		366
Сортування включенням.....		367
Швидке сортування.....		367
Сортування підрахунком.....		369
Які є алгоритми пошуку?	372
Бінарний пошук.....		372
Алгоритм Кнута для пошуку рядків.....		372
Пошук в ширину та глибину.....		376
Що таке скаляр, вектор та матриця?	380
Які найпопулярніші алгоритми й теореми в арифметиці?	389
Квадратні числа.....		390
Числа Фіbonаччі.....		393

Магічні квадрати.....	396
Арифметична прогресія.....	398
Дерево Штерна-Броко.....	400
Алгоритм Евкліда для найбільшого спільного дільника.....	401
Сито Ератосфена.....	403
Доказ існування іrrаціональних чисел.....	404
Метод Герона для знаходження кореня степені п.....	405
Спіраль Теодора.....	408
Китайська теорема про остачі.....	410
Що таке аналітична геометрія?.....	413
Метричний простір.....	414
Які найпопулярніші алгоритми й формулі в геометрії та тригонометрії?.....	415
Аксіоми евклідової геометрії.....	415
Формули площі.....	416
Формула Герона.....	417
Теорема Піфагора.....	419
Число π.....	423
Формула Вієта для π.....	426
Сфери Данделіна.....	433
Формула площини Гаусса (формула шнурування).....	435
Стільникові мережі.....	439
Теорема Вівіані.....	441
Простий алгоритм тріангуляції.....	442
Діаграми Вороного.....	443
Алгоритм точкової агрегації на основі словника.....	452
Алгоритм кластеризації даних.....	455
Теорема про чотири кольори.....	459
Алгоритм Боуера – Ватсона.....	462
Мозаїка Пенроуза.....	468
Піфагорійська тригонометрична тотожність.....	475
Теорема синусів.....	476
Теорема косинусів.....	478
Крива Безье.....	483
Формула лінійної перспективи.....	486
Які найпопулярніші алгоритми в топології?.....	489
Характеристика Ейлера.....	489
Теорема Брауера про нерухому точку.....	492
Які найпопулярніші алгоритми в теорії графів?.....	494
Як знайти вихід з лабіринту?.....	494
Проблема семи Кенігсберзьких мостів.....	496
Алгоритм Флойда-Воршелла.....	501
Алгоритм Пріма.....	504
Алгоритм Дейкстри.....	506
Теорема про чотири кольори.....	510
Які найпопулярніші алгоритми в комбінаториці?.....	512
Трикутник Паскаля.....	512
Кількість комбінацій без повторень.....	514
Біном Ньютона.....	516
Числа Кatalана.....	517
Числа Стірлінга.....	519

Числа Фібоначчі.....	519
Які найпопулярніші алгоритми в теорії ймовірностей?.....	520
Метод Монте-Карло.....	522
Парадокс Монті Холла.....	523
Парадокс дня народження.....	524
Петербурзький парадокс.....	525
Математичне сподівання.....	526
Дошка Гальтона.....	528
Псевдовипадкові числа.....	530
Розподіл Пуассона.....	532
Що таке упередження виживання?.....	534
Аксіоми теорії ймовірності.....	535
Які найпопулярніші алгоритми та поняття в математичному аналізі?.....	537
Похідна функція.....	537
Оператор набла.....	542
Метод скінчених різниць.....	544
Метод Ейлера.....	546
Ліміт (межа) послідовності.....	547
Інтеграл.....	551
Планіметр.....	552
Число е.....	555
Перетворення Фур'є.....	558
Частота ноти.....	560
Які найпопулярніші алгоритми в криптографії?.....	562
Шифр Віженера.....	562
Шифр Вернама.....	567
Частотний аналіз та атака по часу.....	570
RSA.....	571
Які найпопулярніші алгоритми й теореми в теорії множин?.....	572
Діагональний аргумент Кантора про зліченність множини раціональних чисел.....	573
Теорема Кантора про потужність.....	574
Діагональний аргумент Кантора.....	576
Крива Пеано.....	586
Які найпопулярніші теореми та принципи в логіці?.....	590
Теорема Гьоделя про повноту.....	594
Теорема Гьоделя про неповноту.....	594
Теорема Тарського про невизначеність.....	599
Теорема Черча.....	599
Які найпопулярніші алгоритми в алгебрі?.....	600
Теорема Абеля — Руффіні.....	600
Формула Кардано.....	601
Основна теорема алгебри.....	605
Правило Крамера.....	611
Алгоритм Штрассена для множення матриць.....	616
Теорія груп.....	618
Які найпопулярніші поняття в статистиці?.....	620
Медіана.....	620
Рухоме середнє (Ковзне середнє).....	620
Лінійна інтерполяція.....	622
Сліпе рандомізоване контрольоване дослідження.....	622

Нульова гіпотеза.....	623
Кореляція.....	623
Упередження виживання.....	624
Дисперсія випадкової величини.....	625
Лінійна регресія.....	627
А/В тестування.....	629
Які найпопулярніші алгоритми й теореми в економіці?.....	630
Рівновага Неша.....	631
Алгоритм Гейла-Шеплі.....	633
Гроші та криптовалюта.....	635
Історія програмування.....	642
Література.....	651
Доповнення.....	657
Основи електрики.....	657

Про книгу

Ці нотатки призначені для того, щоб надати огляд комп'ютерної науки, принаймні з мінімальним степенем введення в практичну суть питання. Автор вибирає контент, враховуючи власне розуміння того, що є важливим для фахівців у галузі комп'ютерних наук та виходячи з власного інтересу. Він в основному охоплює набір знань SWEBOK. Контент не претендує на вичерпність. Його мета — опис певної структури комп'ютерної науки та навчання принципам прикладної математики й математичного доказу. В нотатках можуть бути повторення тексту для зручності користування.

Комп'ютерна наука ґрунтуються на математиці, зокрема на теорії інформації, теорії алгоритмів, та теорії формальних мов.

Незнання дефініцій (визначень) приведе до нездатності розрізняти. Незнання принципів приведе до нерозуміння можливостей і правил. Незнання алгоритмів приведе до неможливості розрахунків. Отже, в цій книзі буде, крім алгоритмів, теоретичний аспект, який стосується принципів та дефініцій.

Компілятор нотатків Дмитро Попов (Чернівці), Магістр комп'ютерних наук (ЧНУ), понад 10 років у сфері програмування, більше ніж 12 років у сфері інженерії.

Вся інформація була зібрана з авторитетних перевірених джерел, але, якщо ви знайшли неточність, або помилку, ви можете написати про це на електронну пошту: dmytropopov@ieee.org;

Цю книгу можна використовувати тільки в некомерційних цілях. Дозволяється публікація частин книги для навчальних цілей, збереження книги в онлайн бібліотеках, без створення окремої сторінки для книги, чи сайтів спеціально для цієї книги.

Що таке Комп'ютерна наука (інформатика)?

Комп'ютерна наука, або Інформатика — це наука про обчислення, інформацію та автоматизацію.

Ось фундаментальні питання Комп'ютерної науки:

1. Які повинні бути мови програмування?
2. Які повинні бути декларативні мови програмування (мови запитів, мови розмітки, мови конфігурацій)?
3. Які повинні бути формати даних?
4. Які повинні бути обчислювальні машини?
5. Які є алгоритми?
6. Що таке інформаційна ентропія?

Варто зауважити, що “чистий програміст” означає “спеціаліст в комп'ютерній науці, інформатиці”.

Основні постаті в комп'ютерній науці: Алан Тюрінг, Джон фон Нейман, Норберт Вінер, Клод Шеннон, Едсгер Дейкстра, Джон Маккарті. Більше дивіться в списку лауреатів премії Тюрінга.

Відомий комп'ютерний вчений Фрідріх Людвіг Бауер визначає інженерію програмного забезпечення (Software engineering) як "створення та використання надійних інженерних принципів для економічного отримання програмного забезпечення, яке є надійним і ефективно працює на реальних машинах".

Баррі Боем, відомий американський інженер програмного забезпечення, каже: "Інженерія програмного забезпечення — це практичне застосування наукових знань для творчого проєктування та створення комп'ютерних програм".

“Інженерія програмного забезпечення — систематичне застосування наукових і технологічних знань, методів і досвіду для розробки, впровадження, тестування та документування програмного забезпечення” — IEEE Systems and software engineering — Vocabulary.

ISCO Unit Group 2512 визначення:

Розробники програмного забезпечення (Software developers) досліджують, аналізують і оцінюють вимоги до існуючих або нових програмних додатків і операційних систем, а також проєктують, розробляють, тестують і обслуговують програмні рішення відповідно до цих вимог.

Що таке математика?

Наукою називають систему знань, або діяльність людини, що направлена на отримання наукових знань.

Математика — це наука, яка володіє абсолютним доказом.

До математики належать всі науки висновки яких є дедуктивними, тобто точно доведеними на основі певних аксіом. У природничих науках ми часто маємо справу тільки з ймовірними висновками, а не беззаперечними, наприклад, в історії та синоптичній метеорології. Фізика вивчає системи об'єктів, аксіомами яких невідомі та знаходяться в розшуку, математика вивчає системи об'єктів, аксіомами яких відомі.

“Природа моого розуму, без сумніву, така, що я не маю способу не висловити свою згоду з математичними істинами — принаймні до тих пір, поки я їх чітко сприймаю” (Рене Декарт, “Роздуми про першу філософію”, 5).

“До математики були віднесені лише ті питання, у яких досліджуються порядок і вимірювання, і що не має різниці, чи то в числах, фігурах, зірках, звуках чи будь-якому іншому предметі виникає питання про вимірювання” (Рене Декарт, “Правила для спрямування розуму”, 1628)

“Математика – це наука, яка робить беззаперечні висновки. Математика не відкриває закони, бо вона не експериментальна наука, вона також не є розробником теорій, бо не є гіпотезою, але математика суддя обох, і це арбітр, до якого кожен повинен звертатися” (Бенджамін Пірс, “Лінійна асоціативна алгебра”)

“Математика в її найширшому значенні — це розвиток усіх видів формальних, необхідних, дедуктивних міркувань” (Альфред Норт Вайтгед, “Трактат про універсальну алгебру”, том 1. Передмова, 1898 р.)

“У математиці основні постулати приймаються як аксіоми, і вся тема виводиться з цих постулатів. У науці та в інших галузях людської діяльності основні постулати (закони) невідомі, але їх необхідно відкрити” (Лайнус Полінг, “Загальна хімія”, Розділ 1)

Що таке множина?

Множина — це фундаментальна математична сутність, яка може містити елементи. Теорія множин формалізується системою аксіом Цермело — Френкеля. У математиці множини використовуються для групування об'єктів з однаковими характеристиками або спільними властивостями. Елементи множини можуть бути числами, буквами, словами, або будь-якими іншими об'єктами, які можна ідентифікувати.

Множини зазвичай позначають фігурними дужками та перераховують їх елементи через кому. Наприклад, множина цілих чисел від 1 до 5 може бути представлена так: $\{1, 2, 3, 4, 5\}$. Множина також може бути визначена за допомогою характеристичного опису. Наприклад, множина всіх парних чисел може бути позначена як $\{x \mid x \text{ є цілим числом і } x \text{ парне}\}$.

Основні операції з множинами включають об'єднання, перетин, різницю і доповнення. Множини використовуються в багатьох галузях математики та мають широкий спектр застосувань у науці та практиці.

Множини мають кілька важливих властивостей та правил, які характеризують їхню поведінку в математиці. Основні властивості множин включають наступні:

1. Унікальність елементів: У множині кожен елемент повинен бути унікальним, тобто не може бути дублікатів. Якщо в множині зустрічаються однакові елементи, вони рахуються лише один раз. $\{1\} = \{1, 1\}$;
2. Незалежність порядку: Порядок розташування елементів у множині не має значення. Множина $\{1, 2, 3\}$ і множина $\{3, 2, 1\}$ є ідентичними.
3. Розмір множини (Потужність множини): Кількість елементів у множині називається її розміром або кардинальністю. Цей розмір може бути скінченим або нескінченим.
4. Множина порожня: Множина, яка не містить жодних елементів, називається порожньою множиною і позначається символом " \emptyset " або " $\{\}$ ". Розмір порожньої множини дорівнює нулю.
5. Рівність множин: Дві множини вважаються рівними, якщо вони містять однакові елементи. Наприклад, $\{1, 2, 3\}$ і $\{3, 2, 1\}$ є рівними множинами.
6. Підмножини: Множина А є підмножиною множини В, якщо кожен елемент множини А також є елементом множини В. Позначається це як $A \subseteq B$.
7. Операції з множинами: Множини підтримують операції, такі як об'єднання ($A \cup B$), перетин ($A \cap B$), різниця ($A \setminus B$) та доповнення (A'). Ці операції використовуються для комбінування, взаємодії та аналізу множин.
8. Математичні властивості: Множини підкоряються різним математичним властивостям, таким як асоціативність, комутативність та розподільність при використанні операцій над множинами.

Ці властивості є фундаментальними для роботи з множинами в математиці та інших галузях науки. Вони дозволяють виконувати операції над множинами, порівнювати їх та використовувати для моделювання різних реальних ситуацій.

Деякі позначення теорії множин

Порожня множина - {} або \emptyset .

Множина з одним елементом - {a}.

$A = \{a, b, c\}$.

-A — обернена до множини A, яка означає множину всіх елементів, які існують у нашій області визначення, але яких немає в множині A.

$\neg A := \{x: \neg(x \in A)\}$.

$A \supset B$ означає, що B є власною підмножиною A, суворе включення.

$A \supset B := \forall x((x \in A \rightarrow x \in B) \wedge A \neq B)$.

$A \supset B = B \subset A$.

Об'єднання двох множин A і B

$A \cup B := \{x: x \in A \vee x \in B\}$.

Перетин двох множин A і B

$A \cap B := \{x: x \in A \wedge x \in B\}$.

Різниця A і B

$A - B$ (або $A \setminus B$) := $A \cap \neg B$.

Декартів добуток

$A \times B := \{(x, y): x \in A \wedge y \in B\}$.

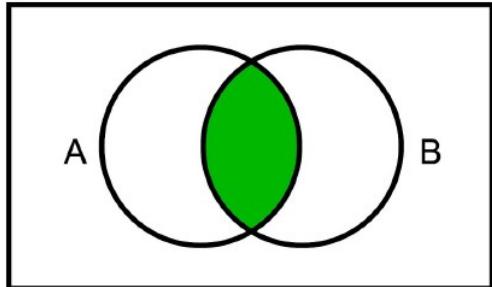
$A \times B \times C := (A \times B) \times C$.

$A \times B \times C \times D = ((A \times B) \times C) \times D$.

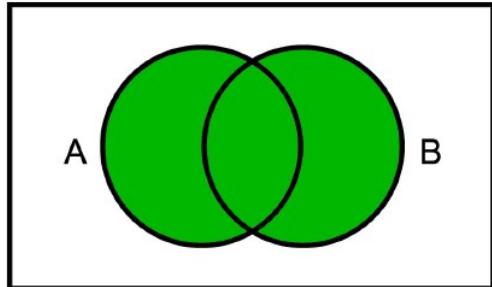
Якщо $A = \{1, 2\}$, $B = \{3, 4\}$, тоді $A \times B = \{\langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle\}$.

$A \times B \neq B \times A$.

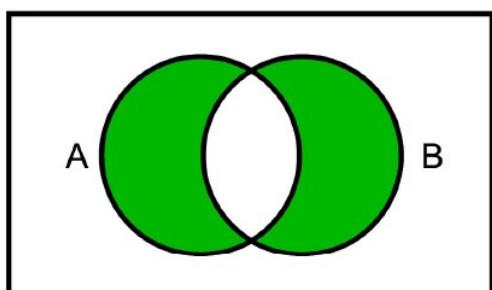
Діаграми Венна



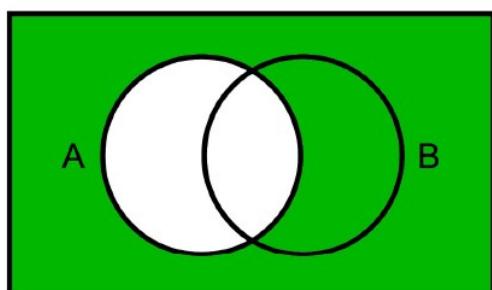
$A \cap B$



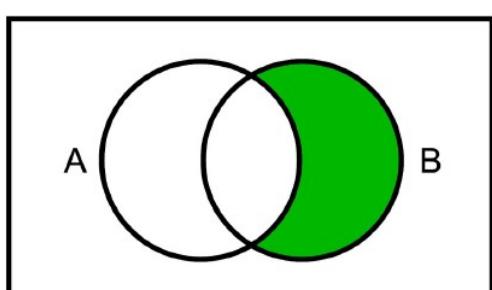
$A \cup B$



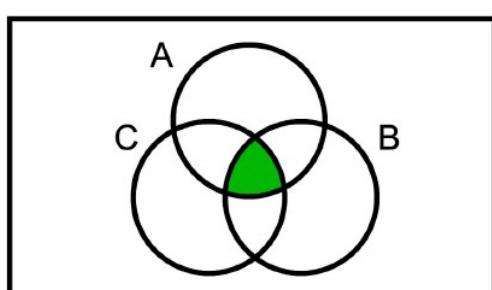
$-(A \cap B)$



$-A$



$-A \cap B$



$A \cap B \cap C$

Аксіоматика Цермело-Френкеля для теорії множин

Наївна теорія множин мала низку парадоксів, тому були розроблені декілька систем аксіом, щоб усунути парадокси. На основі теорії множин тримається весь математичний аналіз. Відомий парадокс Рассела (1901): “Дано множину А, яка містить усі множини, які не містять себе як підмножини. Чи містить множина А сама себе?” Відповідаючи на це питання, отримуємо автореференцію, у будь-якому випадку протиріччя. Парадокс Рассела не можливий в аксіоматиці Цермело-Френкеля, завдяки аксіомі регулярності.

Аксіоматика Цермело-Френкеля (ZF) для теорії множин

Аксіома порожньої множини

SA0. $\exists A (\forall x \neg(x \in A))$.

Існує порожня множина {} або \emptyset .

Аксіома об'ємності

SA1. $A = B \rightarrow \forall x (x \in A \leftrightarrow x \in B)$.

Це означає, що множини рівні, якщо вони мають однакові елементи.

$\{x\} = \{x, x\}$.

$\{x, y\} = \{y, x\}$.

Аксіома пари

SA2. $\exists D (\forall x \in D (x = A \vee x = B))$.

Існують множини лише з двома множинами всередині $D = \{A, B\}$.

Аксіома існування булеана (аксіома множини підмножин).

SA3. $\forall A \exists B (\forall x \in B (\forall y \in x (y \in A)))$.

$\text{Pow}(A)$ — power-set (множина підмножин).

$A = \{x, y\}, B = \text{Pow}(A) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$.

Аксіома об'єднання

SA4. $\forall A \exists B (\forall x \in B (\exists y \in A (x \in y)))$.

$A = \{\{1, 2\}, \{3, 4\}\}$,

Об'єднання всіх множин в A.

$\cup A$ - sum-set.

$$\cup A = \{1, 2, 3, 4\}.$$

Аксіома нескінченості

$$SA5. \exists A (\emptyset \in A \wedge \forall x \in A (x \cup \{x\} \in A)).$$

Існує нескінчена множина типу:

$$\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \dots\}.$$

Аксіома підмножин

$$SA6. \forall A \exists B (\forall x \in B (x \in A \wedge P(x))).$$

Аксіома підстановки

$$SA7. (\forall x \exists !y P(x,y)) \rightarrow \forall A \exists B (\forall y \in B (\exists x \in A (P(x,y)))).$$

Аксіома регулярності (аксіома фундування)

$$SA8. \forall A (\exists B (B \in A) \rightarrow \exists C (C \in A \wedge \neg \exists D (D \in A \wedge D \in C))).$$

Аксіома регулярності забороняє посилання на себе, автореференцію.

Аксіома вибору

$$SA9. (\forall u \in A (u \neq \emptyset) \wedge \forall w, v \in A (w \cap v = \emptyset)) \rightarrow \exists x \forall y (y \in A \rightarrow \exists z (x \cap y = \{z\})).$$

Для кожного непорожнього класу множин, що не перетинаються, існує множина, яка містить рівно по одному елементу зожної множини поточного класу.

Парадокс цирульника

У селі живе цирульник, який голить усіх жителів села, які не голяться самі, і тільки їх. Чи голить цирульник сам себе?

Розглянемо деталі з допомогою множин:

Село = {НеГолятьсяСамі, ГолятьсяСамі, НеГолятьсяВзагалі};

ГолитьЦирульник = { $x \mid x \in \text{НеГолятьсяСамі}$ };

(цирульник є ГолитьЦирульник) \rightarrow (цирульник є НеГолятьсяСамі).

(цирульник є НеГолятьсяСамі) \rightarrow (цирульник є ГолитьЦирульник).

Виходить парадокс, бо ми хочемо визначити одну множину через другу множину, яка саме визначена через першу. Ми хочемо побудувати об'єкт цирульника неможливим шляхом.

Цей парадокс не має розв'язку, оскільки такого цирульника не може існувати. Це запитання провокаційне оскільки воно припускає існування цирульника, який не міг існувати, що є порожнім твердженням, а отже, хибним.

Подібних парадоксів можна вигадати багато, наприклад: Пташка яка підкладає яйця в гніздо тільки тим, хто не кладе їх сам. Чи кладе пташка яйця у своє гніздо?

Така сама ситуація з **парадоксом про всемогутність**.

Відомий псевдопарадокс про всемогутність також є софізмом.

Софізм про всемогутність (який згадує Норберт Вінер в книзі 1964 року): "-Чи може всемогутня істота створити камінь, який не зможе підняти? -Так може. -Тоді вона не всемогутня".

Насправді всемогутня істота може створити камінь, що не зможе підняти, тільки, якщо відмовиться від всемогутності. А відмовитись від всемогутності й водночас залишитись всемогутньою не вийде. Таким чином розсуд в цьому софізмі вимагає виконання взаємозаперечних умов. Крім того, будь-який камінь можна підняти, як казав Архімед, дайте тільки точку опори. Тому псевдопарадокс про всемогутність також є софізмом.

Що таке функція?

Функція — це базове поняття в математиці, програмуванні та багатьох інших науках і галузях. Формально функція визначається як відображення, яке ставить у відповідність кожному елементу однієї множини (називається областю визначення) елемент іншої множини (називається областю значень). Функція описує залежність між цими двома множинами, вказуючи, як кожен вхідний елемент області визначення відповідає певному елементу області значень.

У контексті математики функцію зазвичай позначають як "f", і вона може бути представлена графічно на координатній площині, де область визначення відображається на осі x, а область значень — на осі y.

Загальна форма функції в математиці:

$$f(x) = y,$$

де:

"f" - назва функції.

"x" - вхідний аргумент (елемент області визначення).

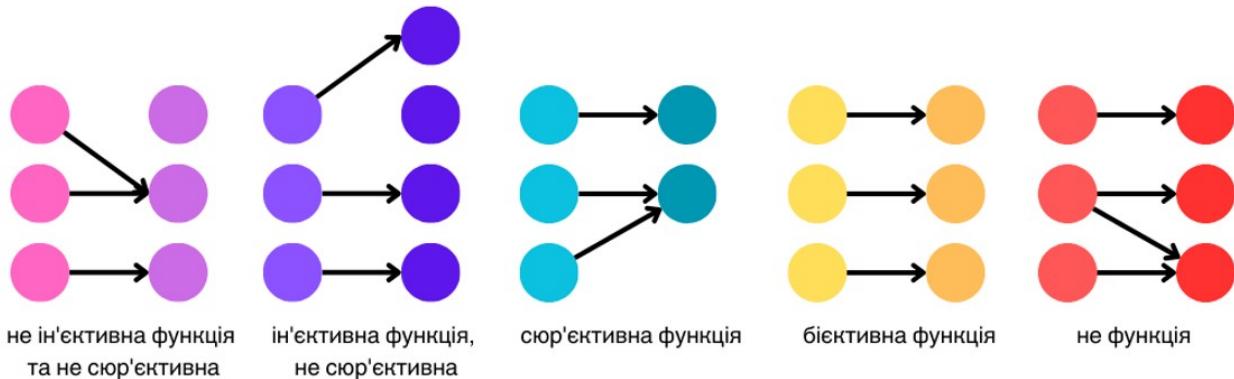
"y" - вихідне значення (елемент області значень), яке відповідає вхідному аргументу.

Функції використовуються для моделювання різних видів взаємозв'язків та розв'язання різних задач у математиці, науках, інженерії, програмуванні та інших галузях.

Біективна функція — це особливий вид функції, яка відповідає двом основним умовам:

1. Кожному елементу з області визначення відповідає рівно один елемент з області значень.
2. Кожен елемент з області значень має рівно один елемент в області визначення.

Це означає, що біективна функція встановлює взаємно-однозначну відповідність між елементами двох множин (область визначення та область значень).



Остання діаграма не зображає функцію, бо в ній існує елемент, якому відповідають два елементи з іншої множини.

Формально, відображення $f: X \rightarrow Y$ - ін'ективне тоді й тільки тоді, коли для кожного y з Y , існує не більш як один (або жодного) x в X такий, що $f(x) = y$. Інакше: f є ін'ективним, якщо для кожного x та x' з X , де $f(x) = f(x')$, виконується рівність $x = x'$.

Сюр'екція (сюр'ективна функція) — в математиці відповідність між двома множинами, в якій з кожним елементом другої множини асоціюється щонайменше один (або більше) елемент першої множини.

Формально, відображення $f: X \rightarrow Y$ є сюр'ективним, якщо для кожного y з Y , існує щонайменш один x з X такий, що $f(x) = y$.

Якщо задані дві математичні структури одного виду (групи, кільця, модулі, поля, векторні простори), то взаємно однозначне відображення (бієкція) елементів однієї математичної структури на іншу, що зберігає структуру, називається ізоморфізмом. Об'єкти, між якими існує ізоморфізм називаються ізоморфними. Формально, ізоморфізм є бієктивним морфізмом.

Швейцарський математик Леонард Ейлер дав одне з перших визначень функції. У своїй роботі “Введення в аналіз нескінченно малих” (1748) Ейлер пише: “Функція змінної z — це аналітичний вираз, який певним чином складається з цієї змінної величини z і постійних величин. Отже, $a + 3z$, $az - 4z^2$, $az + b\sqrt{a^2 - z^2}$, c^z є функціями z ”. Ейлер ще не розрізняє функцію, яка повинна бути однозначною, і будь-який аналітичний вираз, який інакше називають алгебраїчним виразом і який може бути неоднозначним. Оскільки квадратний корінь має два значення, позитивне і негативне. Функції, побудовані з допомогою неоднозначних алгебраїчних виразів, Ейлер називає багатозначними функціями, які не можуть бути функцією в сучасному визначенні. Згодом поняття функції було доведено до свого сучасного вигляду Коші, Діріхле і Ріманом, поки остаточно не було покладено на основу теорії множин, наприкінці 19 - початку 20 століття.

Сучасне визначення функції є в роботі французького математика Жака Шарля Франсуа Штурма. Жак Штурм у своїй праці “Курс аналізу Політехнічної школи, 1857—1863” пише: “Дві змінні є функціями одна від одної, коли відомо, що кожній величині однієї з них відповідає певна величина іншої, навіть якщо співвідношення між ними невідоме і навіть якщо воно не може бути виражене аналітично”.

Французька група математиків “Ніколя Бурбакі” активно поширювала розуміння функції як відображення між двома множинами.

Програмісти можуть розглядати функцію як чорну скриньку, яка приймає щось на вхід і дає відповідний вихід, тоді як сама чорна скринька реалізує якийсь алгоритм, зокрема, аналітичний вираз. Таким чином, може бути багато алгоритмів, що реалізують одну функцію. Розгляд функцій разом з їх алгоритмами, тобто реалізаціями, почали активно вивчати Давид Гільберт, Аллан Тюрінг, Алонзо Черч, Стівен Кліні, Гаскелл Каррі.

Чорна скринька (чорний ящик) — це термін, який використовується у техніці й кібернетиці для позначення об'єкта чи системи, про принципи дії яких нічого невідомо, крім того, що певному вхідному сигналу відповідає певний вихідний сигнал.



$$f(x) := x^2;$$

$$f(2) = 4;$$

Функція називається чистою, якщо вона відповідає функції в математичному сенсі, тобто вона пов'язує кожне можливе вхідне значення з вихідним значенням і нічого більше не робить. Зокрема, чиста функція не має побічних ефектів, тобто її виклик не дає жодного видимого ефекту, крім результату, який вона повертає. Чиста функція не залежить ні від чого, окрім її параметрів, тому, коли її викликають в іншому контексті або в інший час з тими самими аргументами, вона дасть той самий результат.

Існує багато математичних функцій, для яких при цьому не описаній алгоритм.

Аксіома вибору гарантує існування функцій для вибору елементів з непорожніх множин. Це дозволяє визначати функції, які відображають елементи з одних множин в інші.

Існує безліч математичних функцій, для яких можуть бути важко чи навіть неможливо побудувати аналітичний алгоритм, який описує їх поведінку. Це особливо стосується функцій, які виникають у результаті складних математичних операцій, не мають простого аналітичного виразу або їх поведінка залежить від набагато складніших математичних структур.

Наприклад, багато спеціальних функцій, таких як функція еліптичних інтегралів, функція Бесселя, функція Рімана зі структурою, що не може бути виражена аналітично, використовуються в різних областях математики та фізики.

У таких випадках, для обчислення значень цих функцій використовують чисельні методи, інтерполяцію, ряди Тейлора та інші обчислювальні підходи. Складніші функції можуть також бути визначені як розв'язки диференціальних рівнянь чи інтегральних рівнянь, і для їх обчислення використовують чисельні методи.

Отже, хоча існують функції, для яких немає аналітичного алгоритму, на практиці їх значення можуть бути обчислені чисельними методами з високою точністю.

Теорема: Множина всіх арифметичних функцій незліченна (тобто вона більша за множину всіх натуральних чисел).

Доказ: Припустимо, що існує деякий перерахунок набору всіх арифметичних функцій, нехай $f_m(n)$ це значення функції m у цьому перерахунку для аргументу n . Утворимо функцію g таку, що для будь-якого n функція $g(n) = f_n(n) + 1$. Нехай p — номер функції g у цьому перерахунку (списку), так що $g(n) = f_p(n)$. Тоді $f_p(p) = g(p) = f_p(p) + 1$. Це протиріччя (суперечність), тому припущення про перерахунок із використанням натуральних чисел множини всіх арифметичних функцій є невірним.

Що таке бієкція?

Бієкція — це поняття в математиці, яке використовується для опису особливого відображення між двома множинами. Бієкція є однією з трьох основних видів відображень, інші два — це ін'єкція та сюр'єкція.

Бієкція визначається як відображення, яке має наступні важливі властивості:

- Кожен елемент області визначення відображається в унікальний елемент області значень. Іншими словами, кожному елементу області визначення відповідає лише один елемент області значень.
- Всі елементи області значень відображення "використовуються" і мають принаймні одну пряму залежність від елементів області визначення. Жоден елемент не залишається "поза роботою".

Математично бієкцію можна позначити як $f: A \rightarrow B$, де:

- " f " - назва відображення або функції.
- " A " - область визначення (початкова множина).
- " B " - область значень (цільова множина).

Бієкції є важливими в математичних дисциплінах, таких як топологія, алгебра, аналіз. Вони мають застосування в різних контекстах, включаючи розв'язання рівнянь, обчислення інверсій функцій, побудову взаємовідображень між різними множинами та інше. Бієкція також є важливою концепцією в теорії множин та математичній логіці.

У теорії множин теорема Шредера–Бернштейна стверджує, що коли між множинами A і B існують ін'єктивні функції $f: A \rightarrow B$ і $g: B \rightarrow A$, тоді також існує бієктивна функція $h: A \rightarrow B$.

Що таке аксіома вибору?

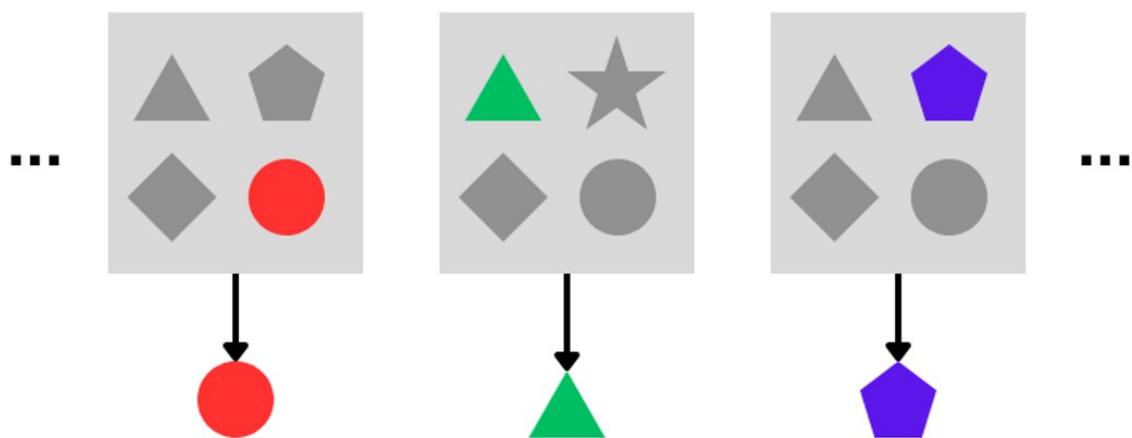
Аксіома вибору — це аксіома теорії множин. Аксіома вибору була сформульована та опублікована Ернстом Цермело в 1904 році.

Деякий час серед математиків точилася суперечка про правомірність так званої аксіоми вибору.

Як і фізики, математики мають деякі принципи. Аксіома вибору виражає інтуїтивний принцип, що елементи можна вибрати з кожного набору. Точніше, аксіома вибору говорить про те, що завжди можна сформулювати необхідну умову виділення (вибору) тих чи інших елементів із множин.

Припустимо, що у нас є множина в якій є певні елементи, чи впевнені ми, що для будь-якої множини елементів із цієї множини можна визначити умову їх виділення (фільтрації) з цієї множини? Багато математиків відповіли: так, це інтуїтивно зрозуміло і ввели аксіому вибору. Аксіома вибору не залежить від інших аксіом Цермело-Френкеля.

Аксіомою вибору є наступне твердження теорії множин: “Для будь-якого класу X непорожніх множин існує функція f , яка призначає кожній множині класу один з елементів цієї множини. Функцію f називають функцією вибору для даного класу”.



Ілюстрація аксіоми вибору, де кожен S_i та x_i представлені у вигляді коробки та фігури відповідно.

Для скінченної множини X аксіома вибору випливає з інших аксіом теорії множин. У цьому випадку це те саме, що сказати, якщо у нас є кілька коробок, кожна з яких містить одну ідентичну реч, то ми можемо вибрати рівно одну реч зожної коробки. Зрозуміло, що ми можемо це зробити: починаємо з першої коробки, вибираємо реч; переходимо до другої коробки, вибираємо реч; і так далі. Оскільки кількість коробок обмежена, то, діючи за нашою процедурою відбору, ми прийдемо до кінця. Результатом буде явна функція вибору: функція, яка зіставляє першу коробку з першим елементом, який ми вибрали, другу коробку з другим елементом, і так далі. (Щоб отримати формальний доказ для всіх скінчених множин, скористайтесь принципом математичної індукції.)

У випадку нескінченної множини X іноді аксіому вибору також можна обійти. Наприклад, якщо елементи X є наборами натуральних чисел. Кожна непорожня множина натуральних чисел має найменший елемент, тому, визначивши нашу функцію вибору, ми можемо просто сказати, що кожна множина пов'язана з найменшим елементом множини. Це дозволяє нам вибирати елемент із кожного набору, тож ми можемо написати явний вираз, який говорить нам, яке значення приймає наша функція вибору. Якщо таким чином можна визначити функцію вибору, то аксіома вибору не потрібна.

Труднощі виникають, коли неможливо здійснити природний вибір елементів з кожного набору. Якщо ми не можемо зробити явний вибір, то чому ми впевнені, що такий вибір можна зробити в принципі? Наприклад, нехай X — множина непорожніх підмножин дійсних чисел. По-перше, ми можемо спробувати діяти так, ніби X скінчений. Якщо ми спробуємо вибирати елемент з кожного набору, то, оскільки X є нескінченим, наша процедура вибору ніколи не закінчиться, і, як наслідок, ми ніколи не отримаємо функцію вибору для всього X . Тому вона не працює. Далі ми можемо спробувати визначити найменший елемент з кожного набору. Але деякі підмножини дійсних чисел не містять найменшого елемента. Наприклад, такою підмножиною є відкритий інтервал $(0, 1)$. Якщо x належить $(0, 1)$, то $x / 2$ також належить йому, і воно менше x . Тому вибирати найменший предмет теж не вийде. Причина, яка дозволяє вибирати найменший елемент з підмножини натуральних чисел, полягає в тому, що натуральні числа мають властивість бути повністю впорядкованими. Кожна підмножина натуральних чисел має унікальний найменший елемент завдяки природному впорядкуванню.

Неформально, в математиці, міра — це функція, що відображає множини на не від'ємні дійсні числа, при цьому, надмножини відображаються на більші числа, ніж підмножини.

У 1902 році французький математик Анрі Лебег у своїх лекціях сформулював теорію міри та гадав, що вона може бути застосована до довільної обмеженої множини. Але поява контрприкладів розвіяла ці сподівання. Побудова таких невимірних множин завжди спирається на аксіому вибору.

Множина Віталі — історично перший приклад множини, що не має міри Лебега (невимірна множина). Цей приклад опублікував 1905 року італійський математик Джузепе Віталі.

Парадокс Банаха-Тарського — теорема в теорії множин, яка стверджує, що тривимірна куля рівноскладена двом своїм копіям.

Парадокс (насправді теорема) Банаха-Тарського — це математична теорема, яка була сформульована і доведена у 1924 році польськими математиками Стефаном Банахом і Альфредом Тарським.

Зважаючи на її неправдоподібність, цю теорему часто використовують як аргумент проти прийняття аксіоми вибору, яка істотно використовується для побудови такого розбиття.

Припустимо, що ми маємо кулю в тривимірному просторі. За допомогою деякого математичного обчислення (перетворення), цю кулю можна розрізати на обмежену кількість окремих точок, а потім знову зібрати ці точки так, щоб утворилася дві однакові кулі, кожна з яких ідентична по розмірах та формі з початковою кулею. Збирання двох куль проводиться без розтягування, а просто з допомогою трансляції (переміщення) та ротації (обертання). Для плоского кола аналогічна властивість неправильна.

Ця формально доведена теорема суперечить фізичній інтуїції, тому й називається парадоксом.

Що таке математична індукція?

Математична індукція — це принцип математичного доведення тверджень, який складається з двох основних кроків:

1. Базовий крок: Перший крок полягає в доведенні твердження для певного початкового натурального числа (зазвичай для числа 1 або 0). Якщо твердження виконується для цього початкового числа, то використовується як базис для подальших розглядів.
2. Крок індукції: Другий крок передбачає доведення, що якщо твердження справджується для деякого натурального числа (наприклад, k), то воно також справджується для наступного числа (тобто $k + 1$). Це робиться за допомогою припущення про індукцію, тобто припущення, що твердження справджується для довільного числа k .

Якщо обидва кроки виконані коректно, то можна вважати, що твердження справджується для всіх натуральних чисел, починаючи з початкового числа, вказаного в базовому кроці.

Математична індукція дуже корисний метод для доведення тверджень, які мають відносну структуру або закономірність, такі як формули для арифметичних послідовностей, рекурентні рівняння та багато інших математичних тверджень.

Математична індукція — це принцип (або аксіома), згідно з яким, якщо для певного твердження відомо, що у разі його вірності для числа x , воно також буде вірне для $x + 1$ і відомо, що воно вірне для певного числа n , то це вірно для всіх (натуральних) чисел. Не слід плутати математичну індукцію з трансфінітною індукцією, оскільки математична індукція використовується лише для зліченної множини.

Зліченна множина — в теорії множин така множина, елементи якої можна занумерувати натуральними числами. Таким чином, будь-яка множина є або зліченою, або незліченою. Всі скінченні множини є зліченними.

Математична індукція: Якщо у разі вірності твердження P для будь-якого елементу x з набору M випливає вірність цього твердження P для $x+1$ (елемент наступним після x), тоді у разі вірності твердження P для будь-якого елементу з набору M , воно буде вірне для всіх елементів цього набору. Приклад: якщо з вірності формули $x < x+1$ випливає вірність формули $(x+1) < (x+1)+1$, тоді якщо формула вірна для певного елементу x , вона буде вірна для всіх елементів.

Трансфінітна індукція: Якщо у разі вірності твердження P для всіх елементів у менше x (тобто $y < x$) з набору M випливає вірність цього твердження P для самого x , тоді у разі вірності твердження P для всіх елементів менше x , воно буде вірне для всіх елементів цього набору.

Математичну індукцію часто називають принципом доміно або ланцюговою реакцією. Принцип доміно можна виразити так: якщо у нас є ряд доміно, в якому, якщо впаде одне доміно, впаде і наступне, то коли впаде перше доміно, впадуть всі інші. Принцип математичної індукції міститься в “Елементах” Евкліда (3 р. до н. е.), наприклад, у теоремі про нескінченність простих чисел. Він також зустрічається в індійського математика Бхаскари II (12 століття нашої ери), а саме, у його методі розв’язування рівнянь, який називається чакравала або циклічний метод. Математична індукція міститься в книзі “Дві книги з арифметики” (1575) італійського математика Франческо Мауроліко. Математична індукція була використана П'єром Ферма в “методі нескінченного спуску” (близько 1640 р.). Математична індукція досить точно визначена і використана Ріхардом Дедекіндом у його праці “Що таке числа?” (1888). У 1889 році італійський математик Джузеппе Пеано ввів математичну індукцію як аксіому арифметики у своїй роботі “Принципи арифметики, представлені новим методом” (1889). Найімовірніше, сучасна назва принципу (математична індукція) була введена британським математиком Августом де Морганом у 1838 році.

Принцип математичної індукції формальною мовою:

Мовою теорії множин: $((x \in K) \wedge (((x \in N \wedge x \in K) \rightarrow (x + 1 \in K))) \rightarrow (K \supseteq N)$.

Мовою предикатів: $(P(0) \wedge \forall x(P(x) \rightarrow P(x'))) \rightarrow \forall xP(x)$.

Загалом висловлення стверджує, що якщо властивість P справедлива для 0 і для кожного x , якщо вона справедлива для x , то вона справедлива для x' (тобто $x + 1$), то ця властивість справедлива для всіх x .

Які є види чисел?

Існує безліч видів чисел, і вони можуть класифікуватися за різними критеріями. Ось деякі основні види чисел:

1. Натуральні числа (N): Це числа, які починаються з 1 (або 0) і нескінченно збільшуються (1, 2, 3, 4, ...).
2. Цілі числа (Z): Вони містять всі натуральні числа, а також їх від'ємні аналоги і нуль (...,-3, -2, -1, 0, 1, 2, 3, ...).
3. Раціональні числа (Q): Це числа, які можна виразити у вигляді дробу p/q , де p і q - цілі числа і q не дорівнює нулю. Приклади раціональних чисел включають десяткові дроби та звичайні дроби ($1/2$, 0.75 , $-3/4$, ...).
4. Ірраціональні числа: Це числа, які не можуть бути виражені у вигляді простого дробу і мають нескінченну неперіодичну десяткову дробову частину. Приклади ірраціональних чисел включають $\sqrt{2}$, π , та e .
5. Дійсні числа (R): Вони включають в себе як раціональні, так й ірраціональні числа.
6. Комплексні числа (C): Це числа у вигляді $a + bi$, де a і b - дійсні числа, i - уявна одиниця ($i^2 = -1$). Комплексні числа використовуються в математиці та інженерії для розв'язання різних завдань.
7. Позитивні, від'ємні та нуль: Це класифікація чисел за їхнім знаком або величиною.
8. Прості числа: Це натуральні числа, які мають лише два дільники: 1 і себе. Приклади простих чисел включають 2, 3, 5, 7, 11 та інші.
9. Кратні числа: Це числа, які без залишку діляться на інше число. Наприклад, 6 є кратним числом 3.
10. Відсотки та десяткові дроби: Це спеціальні види чисел, які використовуються для вираження відсоткових значень та часток.

Це лише загальний перелік видів чисел. В математиці та наукових дисциплінах є багато інших спеціалізованих видів чисел і систем числення, але перелічені вище є основними та найпоширенішими.

Ідеальні числа (досконалі числа) — це конкретний вид натуральніх чисел, які в математиці мають дуже цікаві властивості. Ідеальне число визначається наступним чином:

Натуральне число "n" називається ідеальним, якщо сума всіх його дільників (крім самого числа "n") дорівнює самому числу "n".

Іншими словами, якщо позначити дільники числа "n" як " $d_1, d_2, d_3, \dots, d_k$ ", де "k" - кількість дільників (крім самого числа "n"), то ідеальне число виконує рівність:

$$n = d_1 + d_2 + d_3 + \dots + d_k$$

Прикладом ідеального числа є число 28, оскільки його дільники (крім самого числа 28) - це 1, 2, 4, 7, і 14, і сума цих дільників:

$$1 + 2 + 4 + 7 + 14 = 28$$

Отже, 28 - ідеальне число.

Нікомах Гераський наводить 4 приклади досконалих чисел: 6, 28, 496, 8128.

Він також дає загальне правило для знаходження таких чисел, доказ якого міститься в "Елементах" Евкліда, книга 9, пропозиція 36.

Якщо сума $1 + 2 + 2^2 + \dots + 2^n = p$, а p — просте число, то $2^n * p$ — досконале число.

Наприклад: $1 + 2 + 4 = 7$, а $7 * 4 = 28$, отже 28 є досконалим числом, тому що $1 + 2 + 4 + 7 + 14 = 28$.

Які основні арифметичні формули?

В 1923 році Джон фон Нейман запропонував цікавий метод опису натуральних чисел на основі теорії множин.

Кожне число — це набір унікальних елементів, але кожне більше число містить менше як власну підмножину ($0 \subset 1 \subset 2 \subset 3 \subset 4$). Якщо 0 є порожньою множиною $\{\}$, то $1 \in \{\{\}\}$ (множина, в якій є порожня множина),

$2 \in \{\{\}, \{\{\}\}\}$, або $2 = \{0, 1\}$, а $3 = \{0, 1, 2\}$ тощо.

Множина натуральних чисел (N) добре впорядкована і задовільняє наступні умови:

1. $\emptyset \in N$.

2. $(X \in N) \rightarrow (X' \in N)$.

Де $X' = X \cup \{X\}$.

Якщо $X = \emptyset = \{\}$, то $X' = \emptyset \cup \{\emptyset\}$, тобто $\{\emptyset\}$.

Якщо $X = \{\emptyset\}$, то $X' = \{\emptyset\} \cup \{\{\emptyset\}\}$, тобто $\{\emptyset, \{\emptyset\}\}$.

Це дуже зручне представлення натурального числа, якщо людина звикла працювати з множинами.

Цей метод представлення числа сумісний з аксіоматикою Пеано та конструктивною теорією натуральних чисел Давида Гільберта.

У 1884 році Готлоб Фреге публікує наступну, більш об'ємну, роботу: "Основи арифметики". Фреге був абсолютно правий, що число — це не набір одиниць, число — це набір унікальних елементів, але кожне більше число містить менше як підмножину. Множина може містити тільки унікальні об'єкти, не може бути двох абсолютно однакових об'єктів в множині, тобто $\{1, 1\} = \{1\}$, а $\{1, \{1\}\} \neq \{1\}$. Зокрема, це через те, що ви не зможете задати однозначно функцію на такій множині, наприклад, задамо функцію як множину $\{\{1, \{2\}\}, \{1, \{3\}\}, \{2, \{3\}\}\}$, тоді $f(2) = 3$, а $f(1) = 2$ або 3 , що заборонено, бо функція повинна бути однозначною.

Це: $\{\}$ - порожня множина, $0 = \{\}$, $1 = \{0\}$, $2 = \{0, 1\}$, $3 = \{0, 1, 2\}$, $4 = \{0, 1, 2, 3\}$. $2 + 1 = 2 \cup \{2\} = \{0, 1, 2\}$. $2 - 1 = 2 \cap \{1\} = 1$. (Тут арифметичні операції замінюються операціями над множинами, див. теорію множин).

Аксіоми натуральних чисел (деяка форма аксіом Пеано):

AN0. $x = x$;

AN1. 0 – натуральне число (0 – у множині натуральних чисел, $0 \in N$);

AN2. x' – натуральне число ($x' \in N$);

AN3. $x' \neq 0$;

AN4. якщо $x' = y'$, то $x = y$.

AN5. (Аксіома індукції)

Нехай дано набір M натуральних чисел з такими властивостями:

1) число x належить M .

2) Якщо x належить M , то x' також належить (елемент x' , наступний після x);

Тоді M містить x і всі натуральні числа після x .

Або

AN0. $x = x$;

AN1. $0 \in N$;

AN2. $x' \in N$;

AN3. $x' \neq 0$;

AN4. $(x' = y') \rightarrow (x = y)$.

AN5. (Аксіома індукції)

$$(P(0) \wedge \forall x(P(x) \rightarrow P(x'))) \rightarrow \forall xP(x).$$

Особливість чисел 0, 1 та -1 полягає в їхній важливості та ролі в математиці та числових операціях. Ось кілька важливих аспектів:

1. Число 0 (нуль):

- Додавання нуля до будь-якого числа не змінює його значення: $a + 0 = a$.
- Нуль також є нейтральним елементом множення, що означає, що множення будь-якого числа на нуль дає 0: $a * 0 = 0$.
- У багатьох математичних операціях та властивостях нуль відіграє важливу роль.
- Число в степені 0 дає 1, тобто $x^0 = 1$.
- Ділення на 0 не визначене.

2. Число 1 (одиниця):

- Множення будь-якого числа на одиницю не змінює його значення: $a * 1 = a$.
- Число 1 також важливе в багатьох математичних операціях, і воно виступає як основа великої кількості обчислень.

3. Число -1 (мінус один):

- Мінус один є числом, яке відповідає протилежному напрямку одиниці на числовій прямій.
- Мінус один також використовується у математичних операціях, наприклад, для визначення оберненого числа: якщо $a * b = 1$, то $b = 1/a$, і у випадку, коли $a = -1$, то $b = -1$.

Мінус один є оберненим числом для самого себе. Це означає, що множення -1 на -1 дає 1: $(-1) * (-1) = 1$. Ця властивість використовується для визначення оберненого числа в алгебрі. У геометрії число -1 представляє точку, яка лежить на протилежному від початку координат відстані на числовій прямій. Це відображення відоме як симетрія відносно початку координат.

$$-x * -1 = x;$$

$$x * -1 = -x;$$

Усі ці числа грають важливу роль в математиці та мають специфічні властивості, які роблять їх особливими в числовій системі.

Ідемпотентність — властивість об'єкта або операції при повторному застосуванні операції до об'єкта давати той самий результат, що й за першого.

Приклади ідемпотентних операцій: додавання з нулем, множення на одиницю.

Унарна операція чи функція називається ідемпотентною, якщо її застосування двічі до будь-якого значення аргументу дає таке ж значення, як і застосування один раз:

$$f(f(x))=f(x);$$

Бінарна операція називається ідемпотентною, якщо для довільного елемента x виконується: $x * x = x$;

Обернення множення

$$a * b * (c * (1/b)) = a * c, \text{ якщо } b > 0;$$

$$x * (y + z) = x * y + x * z, \text{ (Закон дистрибутивності);}$$

$$(x * y) * z = x * (y * z), \text{ (Закон асоціативності);}$$

$$x * y = y * x, \text{ (Закон комутативності);}$$

Визначення операцій для натуральних чисел (\mathbb{N})

Додавання (+)

Визначення (Definition):

- 1.1. $x + 0 = x$;
- 1.2. $(x + y) + 1 = x + (y + 1)$;
 $(x + 1 \text{ або } x')$;

Властивість:

- 1.3. $(x + y) + z = x + (y + z)$, (Асоціативність додавання);

Доказ:

$(x + y) + 0 = x + (y + 0)$; (Def. 1.1)
якщо $(x + y) + 1 = x + (y + 1)$ тому $(x + y) + 1' = x + (y + 1')$,
бо $(x + y) + 1' = x + (y + 1') = (x + y) + 1 + 1 = x + (y + 1) + 1$; (Def. 1.2)
якщо $(x + y) + z = x + (y + z)$ тоді $(x + y) + z' = x + (y + z')$; (Def. 1.2)
Значить, $(x + y) + z = x + (y + z)$; (AN5)

- 1.4. $x + y = y + x$, (Комутативність);

Доказ:

$(0 + y) + 0 = 0 + (y + 0)$; (Prop. 1.3)
 $(0 + y) + 0 = (0 + y)$; (Def. 1.1)
 $0 + (y + 0) = (y + 0)$; (Prop. 1.3)
 $(0 + y) = (y + 0)$;
 $0 + (y + 1) = (0 + y) + 1 = (y + 0) + 1$; (Def. 1.2)
 $y + (0 + 1) = (y + 0) + 1$; (Def. 1.2)
Якщо $x + y = y + x$ то $x + y' = y' + x$;
Значить, $x + y = y + x$ (AN5);

Множення (*)

Визначення:

- 2.1. $x * 0 = 0$;
- 2.2. $x * 1 = x$;
- 2.3. $x * (y + 1) = (x * y) + x$;

Властивості:

2.4. $x * (y + z) = x * y + x * z$, (Дистрибутивність множення);

Доказ.

$$x * (y + 1) = x * y + x * 1; \text{ (Def. 2.3)}$$

$$x * (y + 1)' = (x * y + x * 1) + x; \text{ (Def. 2.3)}$$

$$(x * y + x * 1) + x = (x * y + x * 2);$$

$$x * (y + z) = x * y + x * z; \text{ (AN5)}$$

2.5. $(x * y) * z = x * (y * z)$, (Асоціативність множення);

2.6. $x * y = y * x$, (Комутативність множення);

Порядок ($<$, $>$)

(функція, яка повертає true або false)

Визначення:

3.1. $x > y$, якщо існують деякі $z \neq 0$ і $z + y = x$.

3.2. $x < y$, якщо існують деякі $z \neq 0$ і $z + x = y$.

$a \geq b$ означає $a > b$ або $a = b$.

Віднімання (-)

(функція, яка повертає число)

визначення:

4.1. $x - x = 0$;

4.2. $x' - 1 = x$;

4.3. $x - y' = (x - y) - 1$;

Операція $x - y$ не допускається з натуральними числами, якщо $y > x$.

Ділення (/)

Визначення:

5.1. $x / y = z$, де $z * y = x$ і $x * z = y$;

Операція $0 / 0$ = заборонена.

Операція x / y не допускається для натуральних чисел, якщо $y > x$ і $x \neq y * z$.

Властивості:

5. 2. $x / y = (x * z) / (y * z)$;

Піднесення до степеня:

(функція, яка повертає число)

Визначення:

- 6.1. $\text{pow}(0, 0) = \text{undefined}$, не має сенсу (але часто визначається як константа 1);
- 6.2. $\text{pow}(x, 0) = 1$;
- 6.3. $\text{pow}(x, 1) = x$;
- 6.4. $\text{pow}(x, n + 1) = \text{pow}(x, n) * x$;

Властивості:

- 6.5. $\text{pow}(x, y) * \text{pow}(x, z) = \text{pow}(x, y + z)$;

Доказ.

$$\text{pow}(x, y) * \text{pow}(x, 1) = \text{pow}(x, y) * x = \text{pow}(x, y + 1), (\text{Def. 6.1-4});$$

$$\text{pow}(x, y) * \text{pow}(x, z') = \text{pow}(x, y) * (\text{pow}(x, z) * x) = (\text{pow}(x, y) * \text{pow}(x, z)) * x = \text{pow}(x, y + z'), (\text{Def. 2.5});$$

$$6.6. \text{pow}((x + y), n+1) = (\text{pow}((x + y), n) * x) + (\text{pow}((x + y), n) * y);$$

Корінь ($\sqrt{}$)

Визначення:

$$7.1. \text{root}(\text{pow}(a, n), n) = a.$$

Властивості:

$$7.2. \text{root}(\text{pow}(a, n), m) = \text{root}(\text{pow}(a, np), mp).$$

$$7.3. \text{root}(a * b, n) = \text{root}(a, n) * \text{root}(b, n), \text{якщо } a \geq 0 \text{ та } b \geq 0.$$

$$7.4. \text{root}(a / b, n) = \text{root}(a, n) / \text{root}(b, n), \text{якщо } a \geq 0 \text{ та } b > 0.$$

$$7.5. \text{root}(\text{pow}(a, m * n), n) = \text{pow}(a, m).$$

Примітка: $\text{root}(x, n)$ існує, якщо x додатне число (натуральне або раціональне), і корінь існує, якщо x від'ємне, а n парне число, інакше операція не визначена для натуральних чисел.

Індійський математик Брахмагупта ймовірно перший почав практично використовувати від'ємні числа.

Брахмагупта писав, що:

Майно особи, вилучене з майна особи, дає нуль, тобто нічого.

$$x - x = 0.$$

Ніщо (нуль), додане до майна особи, дає те саме майно особи.

$$x + 0 = x.$$

Майно особи, додане до рівного боргу, дає ніщо, тобто нуль.

$$x + -x = 0.$$

Брахмагупта записував свої математичні тексти словами, без спеціальних символів.

Беремо всі операції з натуральних чисел і розширюємо їх спеціальною операцією для цілих чисел.

Додавання і віднімання

Визначення:

$$8.1. x - (x + y) = -y;$$

$$8.2. -x - y = -x + -y = -(x + y);$$

$$8.3. -x - (-x) = -x + x = y - x;$$

Множення

Визначення:

$$9.1. -x * -y = x * y;$$

$$9.2. x * -y = -y * x = -(x * y);$$

“Плюс (+) помножений на плюс дає у добутку плюс (тобто додатне число). Мінус (-) помножений на мінус також дає у добутку плюс. Але плюс на мінус, або мінус, помножений на плюс, дає у добутку мінус” (Рене Декарт, Додатки до “Геометрії”, 1637)

Ділення

Визначення:

$$10.1. -x / -y = x / y.$$

$$10.2. -x / y = -(x / y).$$

$$10.3. x / -y = -(x / y).$$

Піднесення до степеня

Визначення:

$$11.1. \text{pow}(x, -y) = 1 / \text{pow}(x, y);$$

Визначення $\text{pow}(x, -y)$ як $1/\text{pow}(x, y)$ дає математик Леонард Ейлер у своїй книзі “Універсальна арифметика” (1768 року, російською).

Властивості:

$$11.2. \text{pow}(x, y) / \text{pow}(x, z) = \text{pow}(x, y-z).$$

Піднесення до степеня

Визначення:

12.1. $\text{pow}(0, 0)$ = невизначено, не має сенсу (але може задаватись як 1);

12.2. $\text{pow}(x, 0) = 1$;

12.3. $\text{pow}(x, 1) = x$;

12.4. $\text{pow}(x, n + 1) = \text{pow}(x, n) * x$;

12.5. $\text{pow}(x, -y) = 1 / \text{pow}(x, y)$;

Визначення $\text{pow}(x, -y)$ як $1/\text{pow}(x, y)$ дає математик Леонард Ейлер у своїй книзі “Універсальна арифметика” (1768 року, російською).

Властивості:

12.6. $\text{pow}(x, y) * \text{pow}(x, z) = \text{pow}(x, y + z)$;

Доказ.

$$\text{pow}(x, y) * \text{pow}(x, 1) = \text{pow}(x, y) * x = \text{pow}(x, y + 1),$$

$$\text{pow}(x, y) * \text{pow}(x, z') = \text{pow}(x, y) * (\text{pow}(x, z) * x) = (\text{pow}(x, y) * \text{pow}(x, z)) * x = \text{pow}(x, y + z'),$$

12.7. $\text{pow}((x + y), n+1) = (\text{pow}((x + y), n) * x) + (\text{pow}((x + y), n) * y)$; Відповідно до закону дистрибутивності множення в арифметиці.

Робота з дробами потребує однорідності речей, які ви рахуєте, наприклад, два банани та один апельсин разом складають три речі, але два банани та пів апельсина разом не складають дві з половиною речі, тому що дроби говорять про однорідні речі.

Раціональні числа (\mathbb{Q})

Ми розширюємо набір цілих чисел раціональними числами, щоб дозволити операцію поділу в усіх випадках, тобто визначити її без обмежень.

Ми представляємо раціональні числа (дроби) як частини прямої. Позначаємо їх як x/y .

1:1							
1:2				1:2			
1:4		1:4		1:4		1:4	
1:8	1:8	1:8	1:8	1:8	1:8	1:8	1:8

Беремо всі операції над цілими числами та розширюємо їх спеціальними операціями для раціональних чисел.

Визначення рівності:

13.1. $0/0$ = невизначений, але може означати 0.

13.2. $0/x$ = невизначений, але може означати 0.

13.3. $v/w = x/y$, якщо $(v * y) = (w * x)$;

Приклад:

$$2/4 = 1/2.$$

Додавання

Визначення:

14.1. $v/w + x/y = ((v*y)+(x*w))/(w*y)$;

Множення

Визначення:

15.1. $v/w * x/y = (v*x)/(w*y)$;

Ділення

Визначення:

$$16.1. v/w / x/y = (v*y)/(w*x);$$

Віднімання

Визначення:

$$17.1. x/y - z/y = (x-z)/y;$$

Властивості:

$$17.2. v/w - x/y = (vy/y - wx/y)/w;$$

Доказ:

$$\begin{aligned} v/w - x/y &= v/w - (w/y * x)/w = (v - w/y * x)/w = (v/1 - (w*x)/(y*1))/w = \\ &= ((v*y/1*y) - (w*x)/(y*1))/w = (vy/y - wx/y)/w; \end{aligned}$$

Порядок

Визначення:

$$18.1. v/w > x/y, якщо (v * y) > (w * x);$$

$$18.2. v/w < x/y, якщо (v * y) < (w * x);$$

Піднесення до степеня

Визначення:

$$19.1. \text{pow}(x, y/z) = \text{root}(\text{pow}(x, y), z);$$

Властивості:

$$19.2. \text{pow}(x/y, z) = (\text{pow}(x, z)/\text{pow}(y, z));$$

Коли у вас є дріб у показнику степеня, ви можете інтерпретувати це як отримання кореня з основи.

Наприклад, $a^{(m/n)}$ представляє корінь n-го ступеня з a у степені m.

$0,125 = 1/8$, отже, $5^{(0,125)}$ еквівалентно 8-му кореню з 5.

Модульна арифметика

Ділення з остачею (%)

$$a \% b = a - (b * \text{Floor}(a/b))$$

$\text{Floor}(x) = y$, де y — ціле число, де $0 \leq x-y < 1$.

Простий алгоритм роботи % такий:

Нам потрібно знайти c в $a \% b = c$.

Ми робимо:

Знайдіть мінімальне ціле d , що $b * d \geq a$, де $b \geq a$, інакше поміняйте a та b місцями.

Якщо $b * d = a$, то $c = 0$;

Якщо $b * d > a$, то $c = a - (b * (d-1))$;

Рівність за модулем

$$a = b \pmod{n}, \text{ якщо } a \% n = b \% n;$$

(Також $a = b \pmod{n}$, якщо існує d таке $a-b = dn$).

Додавання за модулем

$$a + b \pmod{n} = (a + b) \% n.$$

Операція $a + b \pmod{2}$ також називається Exclusive OR (\vee , XOR).

Віднімання за модулем

$$a - b \pmod{n} = (a - b) \% n.$$

Множення за модулем

$$a * b \pmod{n} = (a * b) \% n.$$

Що таке алгоритм?

Алгоритм — це послідовність конкретних інструкцій, які виконуються для вирішення певної задачі або досягнення певного результату. Алгоритми використовуються в інформатиці, математиці, інженерії та багатьох інших галузях для автоматизації процесів і вирішення різноманітних завдань.

Основні характеристики алгоритмів включають:

1. Визначеність: Кожна інструкція в алгоритмі має бути чіткою та однозначною, без суперечливостей або двозначностей.
2. Кінцевість: Алгоритм повинен завершувати свою роботу після скінченної кількості кроків.
3. Вхід і вихід: Алгоритми можуть приймати вхідні дані та генерувати вихідні дані.
4. Ефективність: Алгоритми повинні бути ефективними, тобто вони повинні виконувати завдання швидко та з обмеженими ресурсами, такими як пам'ять і обчислювальна потужність.

Алгоритми є основною частиною розробки програмного забезпечення, а також використовуються для вирішення різних завдань, таких як сортування даних, пошук інформації, оптимізація процесів та багато інших. Вони представляють собою абстрактні моделі для розв'язання конкретних завдань і можуть бути реалізовані у вигляді програмного коду.

Від латинізованого ім'я арабського вченого Аль-Хорезмі (9 століття) походить слово "алгоритм", що означає певний набір дій для розв'язання конкретної задачі. Також завдяки його роботам до європейських мов потрапили слова шифр і цифра, які походять від арабського слова "сіфр", яке означає "нуль".

Слово "алгоритм" вживає Іоанн де Сакробоско (1195 – 1256) у своїй праці "Мистецтво нумерації" (надрукована в 1692). Цим словом Сакробоско називає науку, яку виклав Аль-Хорезмі у своєму трактаті про числа. У 1240 році французький монах Олександр із Вільд'є (1175 – 1240) пише латинський текст під назвою "Кармен де Алгорісмо", на початку якого сказано, що "алгорісмус" означає мистецтво роботи з індійськими числами. Слово "алгорісмус" походить від латинізованої версії ім'я Аль-Хорезмі.

Французький математик Жан Лерон д'Аламбер у відомій енциклопедії 18 століття пише: "Алгоритм — арабський термін, який використовується деякими авторами, зокрема іспанцями, для позначення практики алгебри. Його також іноді використовують для арифметики, тобто операцій з числами.

Алгоритм, відповідно до сили слова, насправді означає мистецтво обчислення з точністю та легкістю". Вже з 19 століття слово алгоритм набирає значення чіткого набору кроків для отримання певного результату, зокрема результату обчислення. Англійський математик Чарльз Хаттон (1737 – 1823) в своєму математичному словнику (1795) писав: "Алгоритм — загальні правила обчислення в будь-якому мистецтві. Алгоритм також означає загальні правила для виконання операцій арифметики, або алгебри".

Зараз, алгоритм — це точний припис про порядок виконання певної системи операцій над вихідними даними для отримання бажаного результату.

Алгоритм є кінцевою та визначеною процедурою, яка дає визначений результат.

Для представлення алгоритмів ми будемо використовувати просту машину RASP, яка є деякою уявною машиною, але має багато функцій, які такі ж, як у реальних комп'ютерах, і її також можна сконструювати.

RASP (Random-access stored-program machine - машина зі збереженою програмою та довільним доступом, або рівнодоступна адресна машина).

1. RASP має пам'ять, яка поділена на клітинки (реєстри), які мають власні номери.
2. RASP має "блок управління пам'яттю", який може зберігати, читувати дані з реєстра за номером його позначки.
3. RASP має "блок керування програмою", який може читати нумерований список рядків, які є командами, і виконувати ці команди.
4. RASP має лічильник команд, які зберігають номер поточної команди, за замовчуванням він починається з 1.

"Програми — це конкретні формулювання абстрактних алгоритмів, заснованих на конкретному представленні та структурах даних" ("Алгоритми + Структури даних = Програми", Ніклайс Вірт, 1976).

В нашому випадку, програма — це список команд, кожна з яких має числове представлення.

Програма зберігається в частині пам'яті та може бути змінена "блоком управління пам'яттю", тоді "блок керування програмою" може прочитати оновлену команду.

Початкові дані для машини записують безпосередньо в реєстри, а результати роботи читаються безпосередньо з реєстрів (ми не визначаємо спеціальний інтерфейс для введення та виведення даних).

5. RASP має кнопку запуску, яка запускає "блок керування програмою" з першого рядка програми та після виконання лічильник команд збільшується на одиницю і виконує наступну команду в списку.

Команди RASP (Дуже проста мова асемблера, яка в пам'яті зберігається як машинний код):

r(i) - зміст реєстру i.

1. Load a, завантажте a до реєстру 0;
2. Store a, збережіть r(0), до реєстру a;
3. Add a, виконайте r(0) + a і збережіть до реєстру 0;
4. Sub a, виконайте r(0) - a і збережіть до реєстру 0;
5. Mult a, виконайте r(0) * a і збережіть до реєстру 0;
6. Div a, виконайте r(0) / a і збережіть до реєстру 0;
7. Jump a, лічильник команд до a.
8. JZERO a, якщо r(0) = 0, лічильник команд встановлюється до a, якщо ні до наступної команди.
9. Halt, припинити роботу.

Усі ці команди мають числові представлення, які зберігаються в пам'яті та можуть бути динамічно (в роботі) змінені.

Що не може бути описане алгоритмом?

1. Не існує алгоритму який би у всіх випадках міг вірно визначити складність певного алгоритму. Ця інформація відома як теорема про зупинку (Алана Тюрінга). Згідно з нею, неможливо створити універсальний алгоритм, який би міг визначити, чи конкретний алгоритм завершить свою роботу для всіх можливих вхідних даних.
2. Не існує універсального алгоритму, який би казав, чи програма видасть результат чи зависне. Ця проблема відома як проблема зупинки і є наслідком теореми про зупинку Алана Тюрінга. Немає загального алгоритму, який може точно передбачити, чи певна програма завершить виконання чи відомий результат для всіх можливих вхідних даних.
3. Не існує алгоритму, який би для будь-якої системи аксіом міг визначити, чи вона несуперечлива. Ця проблема відома як проблема рішення логічної консистентності. Немає загального алгоритму, який би міг визначити, чи система аксіом (логічні правила) несуперечлива, і це є наслідком теореми Курта Геделя про неповноту.
4. Не існує алгоритму, який би для всіх плиток міг встановити, чи можна ними замостити площину. Ця проблема відома як проблема теорії мозаїк. Немає загального алгоритму, який би міг вирішити, чи можна певними плитками замостили безперервну площину. Ця проблема включає в себе важкі математичні питання, такі як теорія фракталів.
5. Не існує універсального алгоритму, щоб встановити, чи рівень стиснення файлу є найкращим.

Колмогоровська складність — це концепція в теорії обчислень, яка визначає, наскільки складно представити конкретний об'єкт (наприклад, текстовий файл, зображення або будь-який інший об'єкт) за допомогою мінімальної програми чи алгоритму. Іншими словами, це показує, наскільки коротко можна описати цей об'єкт. Чим коротший опис, тим менша Колмогоровська складність.

Ідея Колмогоровської складності полягає в тому, що існує універсальний алгоритм (універсальна машина Тюрінга), який може генерувати будь-який об'єкт, якщо відомий його короткий опис (програма). Однак у багатьох випадках знайти найкоротший можливий опис для конкретного об'єкта є дуже складно або навіть неможливо.

Це веде до того, що немає універсального алгоритму, який би міг автоматично визначити найкращий рівень стиснення для будь-якого файлу, оскільки він вимагав би знань про найкоротший можливий опис кожного можливого файлу. Колмогоровська складність є теоретичною концепцією і демонструє обмеження універсальних алгоритмів у вирішенні деяких завдань, але на практиці вона не використовується для стиснення файлів через велику обчислювальну складність і відсутність універсального алгоритму для її реалізації.

Існують задачі, для яких не існує алгоритму, який би дозволив їх вирішити у всіх випадках. Такі задачі називаються "нерозв'язними" в рамках певної теорії або засобів, які вам доступні.

Давньогрецькі задачі про трисекцію кута і квадрату кола є прикладами таких нерозв'язних задач. Ці задачі були сформульовані в античних часах і стосувалися обмежень, які були накладені на їх вирішення. Для трисекції кута за допомогою циркуля і лінійки було доведено, що ця задача не має алгоритму, який би працював у всіх можливих випадках. Те ж саме стосується і задачі про квадрату кола, тобто побудову квадрата, площа якого дорівнює площі даного кола, використовуючи лише циркуль і лінійку.

Це стосується саме обмежень, які були накладені на методи вирішення цих задач. Варто відзначити, що вивчення нерозв'язних задач має важливе теоретичне значення в математиці, оскільки воно допомагає краще розуміти межі можливостей різних методів і обмежень в математичних задачах.

Проблема відповідності Поста (Post Correspondence Problem, PCP) є відомою проблемою в теоретичній комп'ютерній науці та математиці. Вона була введена Емілем Постом у 1946 році як проблема рішення.

У проблемі Поста вам надається набір плиток, кожна з яких має рядок, написаний зверху та знизу. Мета полягає в тому, щоб визначити, чи існує послідовність цих плиток, які, впорядковані в лінію, дають однакові рядки при об'єднанні рядків зверху та знизу. Кожну плитку можна використовувати кілька разів, і порядок їх використання має значення. Всі плитки з набору повинні бути використані, але їх можна дублювати.

Формально, задано набір пар плиток $\{(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)\}$, де a_i та b_i - рядки. Питання полягає в тому, чи існує послідовність така, що об'єднання a_1, a_2, \dots, a_k дорівнює об'єднанню b_1, b_2, \dots, b_k .

Відомо, що Проблема відповідності Поста є нерозв'язною проблемою, що означає, що не існує алгоритму, який завжди може визначити, чи існує рішення чи ні. Цей результат був доведений за допомогою редукції з проблеми зупинки, іншої нерозв'язної проблеми. Незважаючи на її нерозв'язність, це фундаментальна проблема в дослідженні формальних мов і теорії обчислюваності, що надає уявлення про обмеження алгоритмічного мислення. Для деяких наборів плиток можна сказати, чи існує (чи не існує) комбінація яка дає відповідність (match) між верхнім та нижнім рядами, але це не вийде зробити для будь-якого набору плиток.

Плитки: $\{(a, a), (ba, b), (c, ac), (c, c)\}$,

a	ba	c	c
a	b	ac	c

Що таке складність алгоритму?

Складність алгоритму — це характеристика, яка визначає, наскільки ефективно або ресурсозатратно (такі як час або пам'ять) працює алгоритм при розв'язанні конкретної задачі в залежності від розміру вхідних даних. Вона описує, як збільшення розміру вхідних даних впливає на продуктивність алгоритму.

Складність алгоритму може бути вимірювана відповідно до наступних аспектів:

1. Часова складність: Визначає, як алгоритм залежить від часу для виконання. Вона вказує на кількість кроків або операцій, які алгоритм повинен виконати, щоб обробити вхідні дані. Зазвичай вимірюється у величинах, які виражаються в обсягу операцій або в часі (наприклад, у секундах або мілісекундах).
2. Просторова складність: Вказує на обсяг пам'яті, який необхідний для виконання алгоритму. Це може бути кількість байтів пам'яті, необхідна для збереження даних та інших ресурсів, включаючи стек викликів та кеш-пам'ять. Подібно до часової складності, просторова складність алгоритму часто виражається асимптотично у О-нотації.
3. Складність опису алгоритму — яка довжина опису алгоритму. Може формалізуватися з допомогою визначення "Колмогоровська складність" (визначається мінімальною довжиною програми або опису, необхідного для його генерації або відтворення).
4. Цикломатична складність — це вимірювання, розроблене Томасом Маккейбом для визначення складності програми. Воно вимірює кількість лінійно-незалежних шляхів через програмний модуль. Програми з нижчою цикломатичною складністю легше зрозуміти та менш ризиковано модифікувати. Враховується кількість розгалужень в програмі (умовних операторів). Цикломатична складність обчислюється на основі графу, що відображає цикл роботи програми.

Складність алгоритму може бути різною в залежності від природи самого алгоритму та характеристик вхідних даних. Зазвичай метою є знаходження алгоритму з мінімальною можливою складністю, щоб ефективно вирішити поставлену задачу. Вивчення та аналіз складності алгоритмів є важливою частиною теоретичної та практичної інформатики для оптимізації ресурсів та покращення продуктивності програмних систем.

"О-нотація" (також відома як "асимптотична нотація") — це спосіб опису складності алгоритмів, який вказує на те, як часова чи просторова складність алгоритму змінюється з ростом розміру вхідних даних. Основна ідея О-нотації полягає в тому, щоб вказати верхню межу або границю того, наскільки швидко зростає використовуваний ресурс (час чи пам'ять) зі збільшенням обсягу даних.

Найпоширеніші символи О-нотації включають:

1. O(1): Ця нотація вказує на те, що часова чи просторова складність алгоритму залишається постійною незалежно від розміру вхідних даних. Іншими словами, алгоритм вимагає постійної кількості операцій чи пам'яті.
2. O(n): Це означає лінійну складність, де час або пам'ять зростають лінійно з розміром вхідних даних. Наприклад, якщо n подвоюється, то і ресурси також подвоюються.

3. $O(n^2)$: Ця нотація вказує на квадратичну складність, де час або пам'ять зростають квадратично з розміром вхідних даних. Якщо n подвоюється, то ресурси збільшуються в чотири рази.
4. $O(\log n)$: Це означає логарифмічну складність, де зростання ресурсів сповільнюється з ростом n . Це дуже ефективна форма складності. Алгоритм бінарного пошуку має складність $O(\log n)$.
5. $O(n \log n)$: Ця нотація зустрічається в багатьох ефективних алгоритмах сортування та інших операціях, де зростання ресурсів є під-лінійним, але все ж досить ефективним.

О-нотація дозволяє абстрагуватися від конкретних констант та дрібних деталей і зосередитися на загальній тенденції росту ресурсів у залежності від розміру вхідних даних. Вона корисна для порівняння та аналізу алгоритмів з точки зору їхньої ефективності.

Часова складність алгоритму являє собою кількість часу, необхідного для виконання алгоритму, тобто час для отримання результату.

Часова складність простого алгоритму може бути представлена у вигляді суми: $(s_1 * t_1) + (s_2 * t_2) + (s_3 * t_3) + \dots + (s_n * t_n)$, де кожен s_i – кількість конкретної команди в поточному алгоритмі, а t_i – час, необхідний для виконання цієї команди. Цей спосіб вимірювання ми можемо використовувати, тільки якщо ми знаємо всі кроки алгоритму і визначаємо час для кожного кроку, але зазвичай ми будемо використовувати більш універсальний і формальний спосіб позначення часової складності.

$O(f(x))$ означає $(f(x) * c)$, тобто часову складність поточного алгоритму, де $f(x)$ ми можемо розуміти як кількість усіх кроків поточного алгоритму в гіршому випадку з довжиною вхідних даних x , а c , $c > 0$ — це час виконання найгіршого кроку в алгоритмі (означає найповільнішу команду). Це спрощений опис О нотації, який дає нам функцію часової складності, яка буде асимптотою для фактичної функції часової складності.

Якщо ж інтерпретувати О як нотацію Едмунда Ландау, тоді:

$f(x) = O(g(x))$, якщо існує позитивне дійсне число M і дійсне число x_0 , такі що:

$$|f(x)| \leq Mg(x) \text{ для всіх } x \geq x_0.$$

Це означає, що функція $f(x)$ при $x < x_0$ може бути більшою за $Mg(x)$. Тобто ми говоримо, що для малих значень x нам відхилення не дуже важливі, а для великих значень x функція $f(x)$ завжди повинна бути менше, або рівна $Mg(x)$.

Постійний час.

Якщо f постійна функція $f(x) = y$, то $O(f(x)) = O(f(x+n)) = O(y)$, що означає, що збільшення x не впливає на складність часу.

Якщо час алгоритму статичний, тобто йому завжди потрібен одинаковий час для роботи, навіть якщо вхідні дані будуть збільшенні, ми можемо представляти його часову складність як постійну. $O(1) = O(f(n)=1)$. Символи $O(1), O(2), O(3), O(4), O(5)\dots$ означають, що алгоритм має постійний час роботи.

Зазвичай, в О-нотації всі константи, які призводять до лінійного приросту, скороочуються. Тобто, $O(n * m) = O(n + m) = O(n)$.

Якщо функція $f(x)$ задається лінійним рівнянням (або поліномом без оператора степеня) тоді часова складність алгоритму $O(f(x))$ буде лінійною. В такому випадку кажуть, що складність алгоритму лінійна. Якщо функція $f(x)$ задається поліномом другого степеня, тоді часова складність алгоритму квадратна, наприклад, $f(x) = x^2$, тобто $O(n^2)$.

Приклад алгоритму який отримує на вхід несортований масив чисел, і повертає мінімальну кількість пар чисел, так що сума чисел в парі не була більше за limit. Наприклад, ви хочете розподілити людей в пари, щоб посадити їх в човни. Човни вміщують тільки дві людини та можуть витримати вагу не більше ніж limit. Вам потрібно визначити мінімальну кількість човнів для групи людей.

Цей алгоритм має часову складність у найгіршому випадку $O(n^*log(n))$, де n кількість чисел в початковому масиві. Саме сортування масиву забирає час $O(n^*log(n))$ у найгіршому випадку. Функція sort використовує комбінацію алгоритмів, зокрема, merge sort. Складність функції sort в JS $\leq O(n^*log(n))$. Алгоритм написаний на мові JavaScript. Шляхом техніки двох курсорів (головок зчитування) ми досягли швидкості $O(n^*log(n))$, хоча наївний підхід дав би швидкість $O(n^2)$.

```
function minPairs(arr, limit) {
    arr.sort((a, b) => a - b); //merge sort, O(n^*log(n))

    let left = 0,
        right = arr.length - 1;
    let result = [];

    while (left <= right) {
        let cur = arr[left] + arr[right];

        if (cur <= limit) {
            if (left === right) {
                result.push([arr[left], null]);
            } else {
                result.push([arr[left], arr[right]]);
            }
            left++;
            right--;
        } else {
            result.push([null, arr[right]]);
            right--;
        }
    }

    return result;
}

minPairs([1,4,6,,3,3,2,2,2,1,5,5], 6) // [ [null, 6], [1, 5], [1, 5], [2, 4], [2, 3], [2, 3] ]
```

Що таке логарифм?

Шотландський математик Джон Непер видав у 1614 році невелику книгу “Опис дивовижної таблиці логарифмів”, в якій описав логарифми та тригонометричні функції в таблицях.

Визначення логарифма:

$$\log(a, b) = c, \text{ де } \text{pow}(a, c) = b \text{ і } b > 0.$$

Приклад:

$$\log(2, 8) = 3, \text{ бо } \text{pow}(2, 3) = 8.$$

Властивості:

$$\log(a, b*c) = \log(a, b) + \log(a, c);$$

Доказ

$$\log(a, b) = m; a^m = b;$$

$$\log(a, c) = n; a^n = c;$$

$$b * c = a^m * a^n = a^{m+n};$$

$$\log(a, b*c) = m + n = \log(a, b) + \log(a, c);$$

Логарифм дозволяє прискорити множення багатозначних чисел шляхом складання їхніх логарифмів.

$$32 * 16 = 2^{\log(2, 32)} * 2^{\log(2, 16)} = 2^{(\log(2, 32) + \log(2, 16))} = 2^{(5+4)} = 2^9 = 512.$$

Логарифм числа менше одиниці є від'ємним. Наприклад, логарифм бази 10 числа 0.1 дорівнює -1, бо $10^{-1} = 0.1$;

Англійський математик Генрі Бріггс (1561 — 1630), розвиваючи ідеї Джона Непера, склав і опублікував перші довідкові таблиці десяткових логарифмів. Англійський математик Вільям Отред (1575—1660) зробив важливий внесок у розвиток простоти у використанні логарифмічної лінійки, запропонувавши використовувати дві однакові шкали, які рухаються одна відносно іншої. Сама ідея логарифмічної шкали була раніше опублікована валлійцем Едмундом Гюнтером, але для виконання обчислень цю шкалу потрібно було ретельно вимірюти двома циркулями. Подвійна шкала Отреда дала миттєві результати. Логарифмічна лінійка добре служила інженерам і математикам понад 200 років.

Що таке моном і поліном?

Моном і поліном (або одночлен і многочлен) — це поняття з алгебри, пов'язані з алгебраїчними виразами.

Моном:

Моном — це алгебраїчний вираз, який складається з одного доданку. Зазвичай мономи мають наступний вигляд:

$$cx^n$$

де:

c - це коефіцієнт (число).

x - це змінна або літерал.

n - це невід'ємний ціличисельний показник, який показує ступінь змінної x .

Наприклад, $3x^2$ — це моном, де $c = 3$, x — змінна, і 2 — показник.

Поліном:

Поліном — це алгебраїчний вираз, який складається з суми або різниці декількох мономів. Зазвичай поліноми мають наступний вигляд:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

де:

$P(x)$ — поліном.

$a_n, a_{n-1}, \dots, a_1, a_0$ — це коефіцієнти, кожен з яких є числом.

x — змінна або літерал.

n — найвищий показник ступеня полінома.

Наприклад, $2x^3 - 3x^2 + 5x - 1$ — це поліном третього ступеня.

Основна теорема алгебри: Будь-який поліном додатного степеня (тобто ≥ 1) з комплексними коефіцієнтами має принаймні один комплексний корінь.

Два одночлени (мономи) рівні, якщо вони мають рівні коефіцієнти та вони складаються з однакових букв з однаковими показниками. Одночлени називаються подібними, якщо вони рівні або відрізняються лише коефіцієнтами.

Приклад: Одночлени $2a^2b^3$ і $(6:3)a^2b^3$ рівні, одночлени $2a^3, -3a^3, (1:2)a^3$ називаються подібними.

Що означає $P = NP$?

$P = NP$ — це одна з найважливіших відкритих проблем у теорії обчислень та інформатики. Ця проблема стосується класів обчислювальних задач і має прямий вплив на сферу криптографії та багатьох інших областей.

У термінах теорії обчислень:

1. P — це клас обчислювальних задач, які можуть бути вирішенні за поліноміальний час на детермінованих обчислювальних машинах. Іншими словами, це клас ефективно розв'язуваних задач.
2. NP — це клас задач, для яких, якщо у вас є пропозиція рішення, ви можете перевірити правильність цього рішення за поліноміальний час на детермінованій машині. Іншими словами, це клас задач, для яких можливо визначити правильність рішення досить швидко.

Проблема $P = NP$ полягає в тому, чи є P рівним NP , тобто чи всі задачі, які можна перевірити за поліноміальний час, також можна розв'язати за поліноміальний час. Це питання залишається нерозв'язаним і є однією з самих важливих відкритих проблем в інформатиці. Якщо P дорівнює NP , це може мати серйозні наслідки для безпеки криптографії, оскільки багато сучасних криптографічних протоколів базуються на тому, що деякі задачі в NP надзвичайно важкі для розв'язання.

Сьогодні проблема $P = NP$ залишається невирішеною, і це одна з основних проблем в теорії обчислень.

Теорема Кука-Левіна, також відома як теорема Кука, стверджує, що задача задовільнення булевої формулі в кон'юнктивній нормальній формі (КНФ), коротше задача SAT (Satisfiability Problem), є NP -повною. Це одна з фундаментальних теорем теорії обчислювальної складності і має важливе значення для класифікації складності обчислювальних задач.

Якщо задача SAT є NP -повною, це означає, що вона належить класу NP (клас обчислювальних задач, для яких можна перевірити правильність рішення за поліноміальний час) і що будь-яка інша задача з класу NP може бути зведена до неї за поліноміальний час. Іншими словами, SAT є "складністю" NP -повних задач, оскільки вона може використовуватися для доведення NP -повноти інших задач.

Теорема Кука-Левіна була важливим кроком у розвитку теорії обчислювальної складності та мала великий вплив на багато галузей інформатики, включаючи розробку алгоритмів та криптографію.

Можливо, колись ваш начальник доручить вам розробити алгоритм вирішення конкретної проблеми. Звичайно, ці алгоритми мають бути більш ефективними та розумнішими, ніж просте перерахування варіантів, тобто ніж метод "проб і помилок". Вивчаєте такий алгоритм день, два, три, потім тиждень, місяць, і розумієте, що незабаром начальник зажадає рішення, але ви його поки не знайшли. Після невдалих спроб знайти алгоритми, ефективніші за простий пошук, не хочеться йти до начальника і говорити, що у вас не вистачає інтелекту для вирішення такого алгоритму. В принципі, можна піти й сказати: "Я занадто тупий, щоб розв'язати цю проблему", але навіщо себе завчасно обмовляти.

Можливо, такого алгоритму взагалі немає, тобто вирішити цю проблему можна лише "грубою силою". Але річ у тому, що довести відсутність розв'язання проблеми може бути так само складно, як і знайти на неї відповідь. Ви можете зробити це простіше, якщо сказати, що я поки не можу вирішити цю проблему, але вся група іменитих спеціалістів також не може, тому не поспішайте мене звільнити. Зіткнувшись із такими проблемами, математики та програмісти думали, як добре було б, якби ми знали, чи всі проблеми можна ефективно розв'язати, чи є багато проблем, які неможливо ефективно розв'язати.

Встановлено, що всі обчислювані задачі поділяються на два типи, а саме, на задачі, які можна розв'язати алгоритмічно за поліноміальний час, і ті, які неможливо розв'язати за поліноміальний час, тобто час, необхідний для їх розв'язання, зростає експоненціально залежно від даних. Відомі задачі, розв'язання яких вимагає експоненціального часу, але не доведено, що їх неможливо розв'язати за поліноміальний час. Коли ми говоримо, що час є експоненційним, ми повинні розуміти його як час, більший за будь-який поліноміальний час складності, тобто не заданий поліномом.

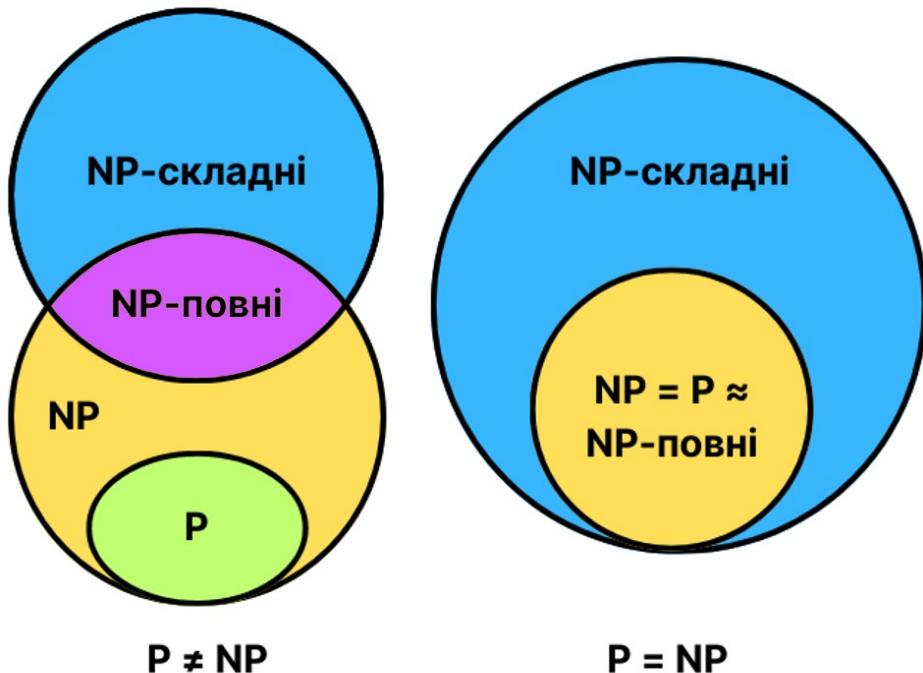
Задачі класу P (P, тобто поліноміальні) — це задачі, для вирішення яких існує алгоритм, часова складність якого є деяким поліномом від n, де n — довжина вхідних даних.

Клас задач NP (NP, недетерміновані поліноміальні) — це клас, який включає клас P, і надана відповідь на будь-яку задачу може бути перевірена на правильність за поліноміальний час, перевірка відбувається за певними правилами, відповідно до сертифіката перевірки. Для задач класу P ви можете надати алгоритм для детермінованої (тобто стандартної) машини Тюрінга, яка працює за поліноміальний час. Для класу NP може бути представлений алгоритм, який дає розв'язок задачі за поліноміальний час для недетермінованої машини Тюрінга, і якщо відповідь знайдено, її можна перевірити за поліноміальний час на детермінованій машині Тюрінга. Поки що всі відомі задачі класу NP, за винятком підкласу P, мають неполіноміальну, експоненційну часову складність. Питання в тому, чи можливо для інших задач класу NP знайти алгоритм, який обчислює відповідь за поліноміальний час на детермінованій машині Тюрінга? (Детермінована машина Тюрінга по суті те саме, що комп'ютер без можливості паралельних обчислень). Тобто $P = NP$ чи ні?

NP повна задача — це задача, яка належить до класу NP і всі задачі з класу NP можна звести до неї за поліноміальний час, тобто звести всі задачі з класу NP до NP-повної задачі й обчислити її як еквівалент у складності.

Прикладом NP повної задачі є задача на знаходження гамільтонового циклу в графі, тобто циклу, який проходить усі вершини графа.

Задача належить до NP-складного класу, якщо вона NP-повна або якщо не належить до NP-класу, тобто не існує відомого алгоритму для недетермінованої машини Тюрінга, який би розв'язав її за поліноміальний час. Задача відноситься до класу NP-hard (NP-складного), якщо вона є NP-повною або невідомий недетермінований алгоритм, що розв'язує її за поліноміальний час, тобто взагалі не належить класу NP.



Якщо P не дорівнює NP , то поліноміального алгоритму розв'язування NP -повної задачі не існує. У теорії складності обчислень теорема Кука–Левіна, також відома як теорема Кука, стверджує, що булева проблема виконання є NP -повною. Тобто вона знаходитьться в класі NP , і будь-яку задачу в NP можна звести за поліноміальний час детермінованою машиною Тюрінга до проблеми булевого виконання. У логіці та інформатиці проблема булевого виконання (B -SAT) — це проблема визначення того, чи існує інтерпретація, яка задовольняє заданий булевий формулі, тобто чи можна підібрати значення змінних так, щоб булева формула була вірною. У своїй фундаментальній роботі 1971 року “Складність процедур доказу теорем” американський вчений-комп’ютерник Стівен Артур Кук формалізував поняття скорочення (зведення) за поліноміальним часом (також відоме як скорочення Куха) і NP -повноти та довів існування NP -повної проблеми, показавши, що булева задача на виконання (зазвичай відома як SAT) є NP -повною. У статті також сформульована широковідома проблема в інформатиці, проблема “ P дорівнює NP ”.

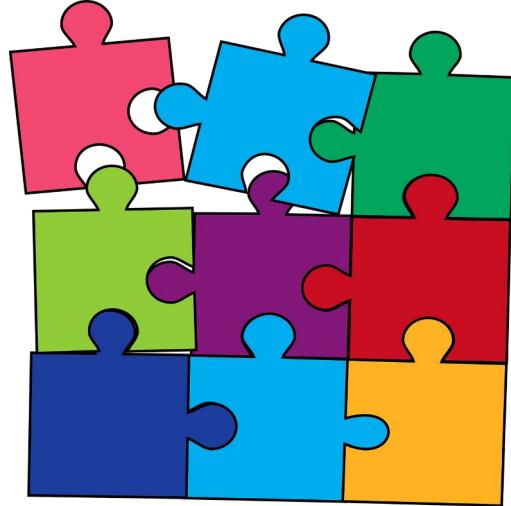
Часова складність алгоритму $O(n^n)$ не є поліноміальною, а є експоненційною. Натомість складність $O(n^2)$ та $O(n^3)$ є поліноміальними, тобто описуються поліномом (многочленом).

Алгебраїчний вираз, що складається з кількох мономів (одночленів), з'єднаних знаками + або -, називається поліномом (многочленом). У математиці моном (одночлен) — це алгебраїчний вираз, що складається з коефіцієнта та буквеної частини, де між літерами стоять множення та піднесення до степеня з натуральними показниками. Алгебраїчний вираз, у якому дія останнього порядку не є додаванням або відніманням, називається одночленом. Множник, виражений цифрами, поставлений перед буквеними множниками, називається коефіцієнтом при одночлені. Отже, одночлен — це або одне число, виражене буквою, або цифрами, наприклад, $-a$, 10 , або добуток чи частка, наприклад, $(a + b) * c$, $(a / b) * c$, a / b , $a / (b * c)$, a^3 .

У загальному випадку, складність задачі збору пазлів може бути NP -складною, що означає, що немає відомого ефективного алгоритму для її вирішення в загальному випадку.

Задача збору пазлів з картинкою виглядає як завдання відновлення зображення, розрізаного на частини (пазли), які потрібно правильно скласти, щоб отримати повне зображення. У випадку картинок, важливо також враховувати візуальну відповідність кожного пазла і його місця в кінцевому зображенні.

Пазли

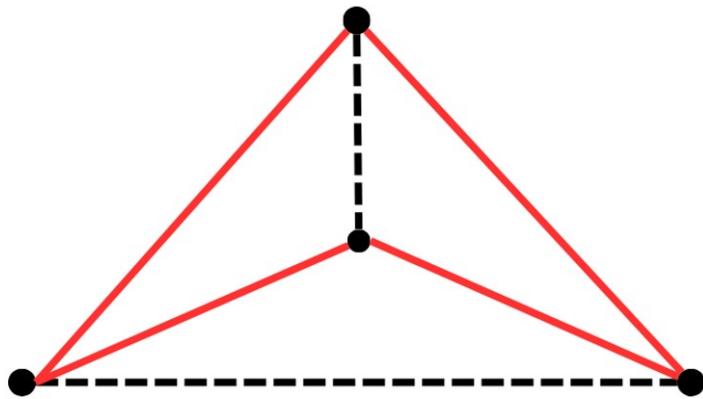


У загальному випадку, задача збору пазлів є NP-повною, що означає, що немає відомих поліноміальних алгоритмів для вирішення її у всіх випадках. Таким чином, часова складність може значно зростати зі збільшенням розміру та складності завдання.

Гамільтонів граф — в математиці це граф, що містить гамільтонів цикл.

Гамільтонів шлях — шлях, що містить кожну вершину графу рівно один раз. Гамільтонів шлях, початкова і кінцева вершини якого збігаються, називається гамільтоновим циклом.

Визначення наявності таких шляхів і циклів у графах є NP-повною задачею. Граф може мати декілька гамільтонових шляхів.



Гамільтонів цикл

У теорії графів, шлях Ейлера — шлях у графі, який проходить кожне ребро рівно один раз. Схожим чином, цикл Ейлера — шлях Ейлера, який розпочинається та завершується в одній вершині. Вперше такі шляхи розглянуті Леонардом Ейлером (Ойлером) під час розв'язання відомої задачі кенігсберзьких мостів в 1736.

Недетермінована машина Тюрінга (НМТ) — це розширення класичної детермінованої машини Тюрінга (ДМТ), де для кожного стану і символу на стрічці існує декілька можливих переходів.

Тобто на відміну від ДМТ, яка має єдиний "шлях обчислень", НМТ має "дерево обчислень" (у загальному випадку — експоненціальне число шляхів).

В теоретичній інформатиці недетермінована машина Тюрінга — машина Тюрінга, функція переходу якої являє собою недетермінований скінчений автомат.

Клас алгоритмів, виконуваних за поліноміальний час на недетермінованих машинах Тюрінга, називається класом NP.

Можна імітувати недетерміновану машину Тюрінга (НМТ) за допомогою набору звичайних машин Тюрінга (ДМТ). Ідея полягає в тому, щоб використовувати кілька ДМТ паралельно та обробляти всі можливі варіанти переходів в кожному кроці.

Один із підходів до цього — це використання "гілок" або "вилок". Кожен крок робиться в кількох копіях ДМТ, де кожна копія відповідає одному можливому переходу. Якщо в якій-небудь копії відбувається успішний перехід, інші копії припиняють свою роботу. Таким чином, можна імітувати недетермінованість.

Що таке жадібний алгоритм?

Жадібний алгоритм — це алгоритм для розв'язання оптимізаційних задач, який працює шляхом прийняття локально найкращого вибору на кожному кроці з надією досягнути глобальної оптимізації. Основна ідея жадібного алгоритму полягає в тому, що він обирає найкращий можливий варіант на кожному етапі, не зважаючи на те, як це вплине на майбутні рішення (вибори).

Основні риси жадібних алгоритмів:

1. **Локальний вибір:** Алгоритм робить вибір, який найкращий для поточного стану, не розглядаючи можливі наслідки для майбутніх кроків.
2. **Невідновлювальність:** Жадібний алгоритм не перевіряє всі можливі варіанти, а тільки той, який виглядає оптимальним на даному кроці.
3. **Не завжди оптимальний:** Хоча жадібний підхід простий і швидкий, він не завжди гарантує знаходження глобальної оптимізації і може привести до підбору підоптимального рішення.

Важливо враховувати, що жадібний алгоритм не завжди є оптимальним для всіх задач, і його використання вимагає ретельного аналізу конкретної задачі та доведення його ефективності.

Жадібний алгоритм — простий і прямолінійний евристичний алгоритм, який приймає найкраще рішення, виходячи з наявних на кожному етапі даних, не зважаючи на можливі наслідки, сподіваючись урешті-решт отримати оптимальний розв'язок.

Наприклад, використання жадібної стратегії для задачі комівояжера породжує такий алгоритм: "На кожному етапі вибирати найближче з невідвіданих міст".

Жадібний алгоритм добре розв'язує деякі задачі, а інші — ні. Більшість задач, для яких він спрацьовує добре, мають дві властивості: по-перше, до них можливо застосувати принцип жадібного вибору, по-друге, вони мають властивість оптимальної підструктури. Необхідно, щоб жадібний вибір на першому етапі не унеможливлював шлях до оптимального розв'язку.

Кажуть, що принцип жадібного вибору застосовується до проблеми оптимізації, якщо послідовність локально оптимальних варіантів дає глобально оптимальне рішення. Як правило, доказ оптимальності відбувається за цією схемою:

Доведено, що жадібний вибір на першому кроці не закриває шляху до оптимального рішення: для кожного рішення існує інше, узгоджене з жадібним вибором і не гірше первого.

Показано, що підзадача, яка виникає після жадібного вибору на першому кроці, схожа на вихідну. Аргумент завершується індукцією.

Задача комівояжера — це комбінаторна оптимізаційна задача, яка полягає в пошуку найкоротшого шляху, який би проходив через всі дані точки (зазвичай це міста або вузли), починаючи з одного пункту і закінчуючи в тому ж пункті, і відвідуючи кожен пункт тільки один раз.

Формально задачу комівояжера можна визначити наступним чином:

Дано:

1. Набір точок (міст або вузлів), які потрібно відвідати.
2. Відстані (або вартості) між кожною парою точок, які вказують, скільки коштує подорожувати з однієї точки до іншої.

Знайти:

Найкоротший можливий шлях, який пройде через всі точки один раз і завершиться в початковій точці (цикл).

Задача комівояжера є важливою у багатьох галузях, таких як логістика, транспорт, маршрутизація, виробництво і багато інших. Це також є однією з класичних задач комбінаторної оптимізації.

Однак задача комівояжера відома своєю NP-складністтю, що означає, що для великої кількості точок знаходження оптимального розв'язку може бути вкрай складним обчислювальним завданням. Вона зазвичай вирішується за допомогою різних евристичних методів та алгоритмів, таких як жадібні алгоритми, методи генетичного програмування, методи гілок та меж та інші.

Яка мова слугує для опису алгоритмів?

Для опису алгоритмів використовують різні мови програмування та псевдокод. Мови програмування, такі як Python, C++, Java, а також багато інших, можуть бути використані для реалізації алгоритмів і їх подальшого опису.

Псевдокод — це структурований спосіб запису алгоритмів, який наближений до мов програмування, але часто використовується для пояснення алгоритмів без прив'язки до конкретної мови програмування. Псевдокод може бути використаний для узагальненого опису алгоритму та його логіки, щоб інші розробники зрозуміли його безпосередньо.

Отже, для опису алгоритмів можна використовувати мови програмування або псевдокод, залежно від потреби та цілей опису.

Ви можете використовувати рекурсивні функції та лямбда-числення для опису алгоритмів, особливо якщо алгоритм потребує рекурсивного або функціонального підходу.

Рекурсивні функції — це функції, які викликають самі себе. Вони часто використовуються для опису алгоритмів, які можуть бути рекурсивними за природою, наприклад, обходи дерева, розбиття задачі на менші підзадачі тощо.

Лямбда-числення — це математична система для вираження обчислень, яка використовує анонімні функції та абстракції. Вона може бути використана для опису алгоритмів на більш абстрактному рівні, що дозволяє зосередитися на математичних властивостях алгоритму.

Як правило, використання рекурсивних функцій і лямбда-числення більше підходить для теоретичного опису алгоритмів та математичного моделювання, ніж для практичної реалізації в реальних програмах. У реальних програмах для опису алгоритмів частіше використовують мови програмування та псевдокод.

Ось простий приклад рекурсивної функції із використанням мови програмування JavaScript для обчислення факторіалу числа:

```
function factorial(n) {
    if (n === 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

// Приклад використання:
var result = factorial(5); // Результат буде 120
console.log(result);
```

У цій функції factorial, якщо n дорівнює 0, функція повертає 1 (базовий випадок). У протилежному випадку, вона викликає саму себе з аргументом n - 1 і множить результат на n. Це є типовим прикладом рекурсивної функції.

Тепер подивимося на простий приклад лямбда-числення для обчислення квадрата числа:

У лямбда-численні може бути використана наступна абстракція для обчислення квадрата числа:

$\lambda x. x * x$

Ця абстракція приймає аргумент x і повертає його квадрат. У лямбда-численні вирази подаються у вигляді функцій і абстракцій, які використовуються для обчислень.

Це лише прості приклади, але вони ілюструють, як можна використовувати рекурсивні функції і лямбда-числення для опису алгоритмів.

Факторіал числа може бути виражений у лямбда-численні, використовуючи рекурсивну абстракцію. Ось спрощений приклад того, як ви могли б визначити факторіал числа n у лямбда-численні:

fact = $\lambda n. (\text{iszzero } n) 1 (\text{mult } n (\text{fact } (\text{pred } n)))$

У цій абстракції:

- fact - це ім'я функції, яка обчислює факторіал.
- n - це вхідний аргумент, для якого обчислюється факторіал.
- iszzero - це лямбда-функція, яка перевіряє, чи n дорівнює нулю.
- 1 - це базовий випадок, коли n дорівнює нулю. У цьому випадку факторіал рівний 1.
- mult - це лямбда-функція, яка виконує множення двох чисел.
- pred - це лямбда-функція, яка віднімає 1 від числа.

Ця абстракція використовує рекурсію для обчислення факторіалу числа n . Вона перевіряє, чи n дорівнює нулю. Якщо так, вона повертає 1. У протилежному випадку, вона викликає саму себе з $\text{pred } n$ і множить результат на n . Це визначення лямбда-функції дозволяє обчислювати факторіал будь-якого натурального числа.

Ось так ми б писали код, якби не було умовних операторів if-else, switch.

```
const a = 5;  
const b = 2;  
  
const state = 1;  
  
const result = (a * (1-state) + b * state);  
  
console.log(result); //2, or 5 if state 0
```

До 20 століття алгоритми записувалися неформально, бо їх повинні були розуміти люди. Але вчені захотіли створити мову, яка б дозволяла записувати алгоритми так, щоб елементарна машина могла їх зрозуміти. Так була створена мова програмування, яка включала один вічний цикл, умовний оператор, операцію зупинки циклу, арифметичні операції.

Якщо людину повернути в часи аналогової техніки, вона одразу зрозуміє чому любить програмування. Програмування дозволяє робити універсальні пристрой. В людини у смартфоні може бути цілий оркестр. Мова програмування дозволяє записувати алгоритми для цифрових пристрой. До винайдення мов програмування алгоритми записували неформально, бо розуміти їх мали не машини, а людська голова. Відомий алгоритм Евкліда, ще до нашої ери створений, потрібно було якось комп'ютеру пояснити, от й створили машину Тюрінга з її елементарною мовою, потім лямбда-числення, мову асемблера, Cobol, Fortran, Algol, Lisp, Simula, Pascal, C, C++. Лейбніц й Беббідж хотів створити комп'ютери, щоб не навантажувати математиків рутинною роботою. Тюрінг хотів універсальний прилад створити. Виходить замість того, щоб розробляти новий хардвер, можна просто софтвер запрограмувати й отримати навіть кращий результат.

Який мінімальний набір функцій для опису будь-якого алгоритму?

Теорія рекурсивних функцій стосується математичної теорії обчислюваності та формалізує поняття обчислювальних функцій та алгоритмів. За класичною теорією рекурсивних функцій, існує набір примітивних функцій і конструкцій, з яких можна побудувати будь-яку обчислювальну функцію.

Цей набір включає такі основні математичні функції та конструкції:

1. Початкові функції: До них відносяться базові арифметичні операції, такі як додавання, віднімання, множення і ділення. (Операція множення може бути побудована на основі простіших функцій, вона не є базовою).
2. Умовні конструкції: Умовні оператори дозволяють обирати одну з альтернативних гілок обчислення в залежності від заданої умови.
3. Рекурсія: Рекурсія дозволяє функціям викликати самих себе. Це дуже потужна конструкція для вирішення завдань, які можуть бути розділені на менші аналогічні підзадачі.
4. Мінімізація: Мінімізація використовується для знаходження найменшого значення, при якому задана функція приймає певне значення (це пов'язано з поняттям часткової рекурсивної функції).
5. Комбінація функцій: Можна поєднувати функції і конструкції для створення складних функцій.

Ці п'ять основних елементів дозволяють створювати рекурсивні функції, які можуть вирішувати різноманітні обчислювальні завдання. Теорія рекурсивних функцій є однією з теорій, що лежить в основі обчислювальної теорії та теорії алгоритмів, і вона використовується для формалізації обчислювальних процесів.

Рекурсивна функція

Функція $f(n_1, \dots, n_k)$, яка відображає натуральні числа в натуральні числа, називається примітивно-рекурсивною, якщо вона побудована з допомогою наступних (і тільки цих) функцій:

1. Функції константи

Усі функції типу:

$$Cnst(x) = c, Cnst(x_1, \dots, x_n) = c, \text{ де } c \text{ — постійне значення (стала, константа).}$$

2. Функція слідування

$$S(x) = x + 1.$$

3. Функція вибору

$$Sel(y, x_1, x_2, x_3, \dots, x_n) = x_y.$$

Наприклад,

$$Sel(1, x_1, x_2, x_3) = x_1.$$

$$Sel(2, x_1, x_2, x_3) = x_2.$$

$$Sel(3, x_1, x_2, x_3) = x_3.$$

4. Функція підстановки

$$Sbt(x_1, \dots, x_n) = g(h_1(x_1, \dots, x_n), \dots, h_m(x_1, \dots, x_n)).$$

Наприклад,

$$Sbt(x) = Sel(2, S(x), S(x+1), S(x+2)) = S(x+1).$$

$$Sbt(x) = S(S(x)).$$

5. Функція рекурсії

$$Rec(x_1, x_2, \dots, x_n, 0) = g(x_1, x_2, \dots, x_n).$$

$$Rec(x_1, x_2, \dots, x_n, y+1) = h(x_1, x_2, \dots, x_n, y, Rec(x_1, x_2, \dots, x_n, y)).$$

Частково рекурсивна функція визначається аналогічно примітивно рекурсивній, тільки двом операторам, суперпозиції та примітивній рекурсії, додається ще третій оператор — мінімізації аргументу.

5. Функція мінімізації

$Min(f, x)$ означає:

Перший випадок: повернути найменше y для якого $f(x, y) = 0$.

Другий випадок: повернути $undefined$ (невизначено), якщо немає y , що задовольняє перший випадок.

Функція $f(x_1, \dots, x_k)$ називається загальнорекурсивною, якщо існує скінчений набір рівнянь, такий, що для будь-якого вибору чисел n_1, \dots, n_k можна вивести лише один m , $f(n_1, \dots, n_k) = m$.
 Загальнорекурсивна функція — частково рекурсивна функція, визначена для всіх значень аргументів.
 Завдання визначення того, є частково рекурсивна функція з даним описом загальнорекурсивної чи ні, алгоритмічно нерозв'язне.

Твердження про те, що загальні рекурсивні функції, як ми їх визначили, вичерпують клас усіх ефективно обчислюваних функцій, відоме як теза Черча. Оскільки це твердження по суті є філософським, то про його доказ чи спростування не може бути й мови. Проте є вагомі аргументи на користь цієї тези: це в основному той факт, що всі відомі обчислювані функції є загальнорекурсивними, а також те, що зовсім інші спроби визначити загальні рекурсивні функції (наприклад, абстрактна машина Тюрінга) дали той самий результат.
 Множина S називається рекурсивною, якщо її характеристична функція є загальнорекурсивною.
 Характеристична функція $x(n)$ для множини S — це функція, яка дає відповідь 1, якщо об'єкт n знаходиться у множині S , і відповідь 0, якщо об'єкт n не входить до множини S .

Набір S є рекурсивно перерахованим, якщо S порожній або якщо S є діапазоном деякої загальної рекурсивної функції.

Одним із популярних прикладів загальнорекурсивної, але не примітивно рекурсивної функції є функція Аккермана.

Функція Аккермана.

Розглянемо перехресну рекурсію:

$$\Phi(a, 0) = 2 * a + 1,$$

$$\Phi(0, n+1) = \Phi(1, n),$$

$$\Phi(a', n') = \Phi(\Phi(a, n+1), n).$$

Ця рекурсія має властивості формалізованої обчислювальної процедури, але визначена з її допомогою функція $\Phi(a, n)$ не може бути визначена з допомогою примітивних рекурсій, оскільки вона росте швидше, ніж будь-яка примітивна рекурсивна функція.

Теорема. Існують ефективно обчислювальні функції, крім примітивно рекурсивних.

Усі схеми генерування примітивних рекурсивних функцій з однієї змінної можна ефективно перерахувати, наприклад, у вигляді $f_m(n)$, таким чином отримати ефективно обчислювальну функцію двох змінних m і n . Якщо ми поставимо тепер $g(n) = f_n(n) + 1$, то g може не бути примітивно рекурсивною, але, звичайно, вона все одно буде ефективно обчислюваною функцією.

Щоб довести, що множина всіх арифметичних функцій незліченна (тобто вона більша за множину всіх натуральних чисел), використовується подібний метод. Припустимо, що існує деякий перерахунок набору всіх арифметичних функцій, нехай $f_m(n)$ це значення функції m у цьому перерахунку для аргументу n . Утворимо функцію g таку, що для будь-якого n функція $g(n) = f_n(n) + 1$. Нехай p — номер функції g у цьому перерахунку (списку), так що $g(n) = f_p(n)$. Тоді $f_p(p) = g(p) = f_p(p) + 1$. Це протиріччя (суперечність), тому припущення про перерахунок із використанням натуральних чисел множини всіх арифметичних функцій є невірним.

Що таке лямбда-числення?

Лямбда-числення (λ -числення) — це формальна система, розроблена американським математиком Алонзо Черчом для формалізації та аналізу концепції обчислюваності. Це обчислення було розроблено Алонзо Черчом у 1933 році в рамках програми створення математичного фундаменту з допомогою функцій, а не множин, щоб уникнути перешкод, таких як парадокс Рассела, і створити конструктивну основу для математики. Однак у першій версії лямбда-числення був виявлений парадокс Кліні-Россера (відкритий Кліні в 1935 році в теорії лямбда-числення Черча, пізніше усунений Черчом), який показав вразливість до парадоксів навіть таких конструктивних теорій, як лямбда-числення. Алонзо Черч у варіанті лямбда-числення, описаному в роботі 1941 році, описав лямбда-числення без парадоксів, тобто усунув парадокси. Лямбда-числення виявилося зручним інструментом у вивченні рекурсії та обчислюваності функцій і лягло в основу парадигми функціонального програмування. Основне правило програмування функціональної парадигми: Чистота функцій, тобто відсутність побічних ефектів, та імутабільність змінних, тобто всі змінні є константами. Функція повинна приймати значення і повернати, а не змінювати стан змінних в процесі своєї роботи, тобто вона повинна бути закрита в собі.

Лямбда-числення можна розглядати як ідеалізовану мінімалістичну мову програмування, у цьому сенсі лямбда-числення подібне до машини Тюрінга, іншої мінімалістичної абстракції, здатної виконувати будь-який відомий зараз алгоритм. Різниця між ними полягає в тому, що лямбда-числення відповідає функціональній парадигмі програмування, а машина Тюрінга — імперативній. Тобто машина Тюрінга має певний “стан” — список символів, які можуть змінюватися з кожною наступною інструкцією. На відміну від цього, лямбда-числення уникає станів, воно має справу з функціями, які отримують значення параметрів і повертають результати обчислень (можливо, інші функції), але не змінюють вхідні дані.

Основи лямбда-числення (λ -числення).

Лямбда-вирази складаються з таких символів:

1. Змінні: A,B,C,a,b,c,....
2. Дужки: (,).
3. Оператор абстракції: λ .

Набір λ -виразів можна визначити індуктивно так:

1. будь-яка змінна є λ -виразом;
2. абстракція $\lambda x.M$ — λ -вираз, якщо x — змінна, а M — λ -вираз;
3. аплікація MN є λ -виразом, якщо M і N є λ -виразами.

Лямбда-числення засноване на двох фундаментальних операціях:

Аплікація означає застосування або виклик функції щодо заданого значення. Зазвичай його позначають fa , де f — функція, і a — аргумент. Це відповідає загальноприйнятому позначенням $f(a)$ в математиці, яке також іноді використовується, але для Лямбда-числення важливо, щоб f розглядався як алгоритм, який обчислює результат із заданого вхідного значення.

Абстракція, або λ -абстракція, буде функції за заданими виразами. А саме, якщо $t[x]$ є виразом, який вільно містить x , то позначення $\lambda x.t[x]$ означає: λ є функцією аргументу x , яка має вигляд $t[x]$. Таким чином, нові функції можна побудувати з допомогою абстракції. Приклад: $\lambda x.x^2$, $(\lambda x.x^2)2 = 4$.

Бета-редукція, або β -редукція

Оскільки вираз $\lambda x.2 * x + 1$ позначає функцію, яка присвоює кожному x значення $2 * x + 1$, то для обчислення виразу $(\lambda x.2 * x + 1)3$, який включає як аплікацію, так і абстракцію, необхідно виконати заміну числа 3 в доданку $2 * x + 1$ замість змінної x . Результатом є $2 * 3 + 1 = 7$. Це міркування загалом записується як $(\lambda x.t)a = t[x := a]$ і називається бета-редукцією.

У лямбда-численні теорема Черча-Россера стверджує, що при застосуванні правил редукції до термінів порядок, у якому обираються скорочення, не впливає на кінцевий результат. Теорема була доведена в 1936 році Алонзо Черчом і Дж. Барклі Россером, на честь якого вона названа.

$$\begin{array}{ccc} (\lambda x. ax)((\lambda y. by)c) & \xrightarrow{\hspace{2cm}} & a((\lambda y. by)c) \\ \downarrow & & \downarrow \\ (\lambda x. ax)(bc) & \xrightarrow{\hspace{2cm}} & a(bc) \end{array}$$

Основною формою еквівалентності, визначеною в термінах лямбда, є альфа-еквівалентність (α -еквівалентність). Наприклад, $\lambda x.x$ і $\lambda y.y$ є альфа-еквівалентними лямбда-термінами, і обидва представляють ту саму функцію (функцію ідентичності). Терміни x і y не є альфа-еквівалентними, оскільки вони не входять до лямбда-абстракції. Усе, що можна отримати на вході функції в лямбда-численні, є терміном. У лямбда-численні $y + 2 \neq x + 2$, оскільки x і y можуть приймати різні значення. Але якщо змінні закриті $\lambda y.y + 2 = \lambda x.x + 2$, то вирази рівні. Загалом, функції $f_1(x)$ і $f_2(x)$ рівні, якщо для будь-якого x буде $f_1(x) = f_2(x)$.

Іта-редукція (η -редукція) виражає ідею про те, що дві функції ідентичні тоді й тільки тоді, коли вони, застосовані до будь-якого аргументу, дають однакові результати. η -редукція перетворює формули $\lambda x.fx$ і f одна в одну, тільки якщо x не має вільних входів у f . Мета іта-редукції (η -редукції) полягає в тому, щоб застосувати або скасувати абстракції над функцією, щоб її спростити. Це можливо, коли функція більше нічого не може зробити зі своїм аргументом. Наприклад, уявімо, що у нас є проста функція $fx = gx$. Припустимо, g та f приймають один і той самий аргумент x , і аплікація функції дає те саме значення. Оскільки f і g беруть одинаковий аргумент і дають одинаковий результат, ми можемо спростити рівняння до $f = g$. У лямбда-численні це спрощення називається іта-редукція (η -редукція).

Каррування функцій багатьох змінних може бути виражене в λ -численні як функції однієї змінної, яка є “синтаксичним цукром”, тобто спрощенням позначення. Описаний процес перетворення функцій багатьох змінних у функцію однієї змінної називається карруванням або каррінг на честь американського математика Гаскелла Каррі (1900 – 1982), хоча його також самостійно винайшов Мойсей Шейнфінкель (1889 – 1942).

“Синтаксичний цукор” дає програмісту альтернативний спосіб запису іншої, вже наявної в мові синтаксичної конструкції та при цьому є більш зручним, більш коротким.

λ -позначення можна поширити на функції більш ніж однієї змінної. Наприклад, вираз x -у визначає дві функції h і k двох змінних, які визначаються:

$$h(x, y) = x - y,$$
$$k(y, x) = x - y.$$

Їх можна позначити як

$$h = \lambda xy.x-y,$$
$$k = \lambda yx.x-y.$$

Однак ми можемо уникнути потреби в спеціальному позначенні функцій кількох змінних, використовуючи функції, значеннями яких є не числа, а інші функції. Наприклад, замість двомісної функції h вище розглянемо одномісну функцію h_c , визначену як

$$h_c = \lambda x.(\lambda y.x-y).$$

Для кожного числа a маємо

$$h_c(a) = \lambda y.a-y;$$

отже, дляожної пари чисел a, b ,

$$(h_c(a))(b) = (\lambda y.a-y)(b) = a - b = h(a,b).$$

Використання функції h_c замість h зазвичай називають карруванням.

Каррування в мові JavaScript:

```
const curryAdd = (a) => (b) => a + b;  
const add5 = curryAdd(5);  
console.log(add5(3)); // Outputs 8
```

Що таке комп'ютерна програма?

Комп'ютерна програма — це послідовність інструкцій, які вказують комп'ютеру або іншому обчислювальному пристрою, що робити. Програма складається з набору команд, які вказують, як обробляти дані, виконувати обчислення, взаємодіяти з користувачем та виконувати інші завдання.

“Програми — це конкретні формулювання абстрактних алгоритмів, заснованих на конкретному представленні та структурах даних” (“Алгоритми + Структури даних = Програми”, Ніклаус Вірт, 1976)

В загальному, “Програма раніше означала лист, скріплений королівською печаткою (тобто вказівку, припис)” (“Енциклопедія Дідро та Д'Аламбера”, 1765)

Що таке універсальний комп'ютер?

Універсальний комп'ютер — це тип комп'ютера, який може виконувати різні обчислення та завдання, завдяки можливості програмування його операцій та функцій. Термін "універсальний" означає, що цей комп'ютер може бути налаштованим для виконання різних завдань, від обчислення математичних операцій до обробки тексту, графіки, аудіо, інтернет-перегляду та багатьох інших функцій.

Можливість створення комп'ютерів загального призначення для виконання всіх можливих алгоритмів базується на фундаментальних принципах і концепціях інформатики та обчислювальної техніки, зокрема:

Теорія обчислюваності: Алонзо Черч і Аллан Тюрінг запропонували математичні моделі обчислювальних пристрій. Ці моделі можуть реалізувати будь-який обчислювальний алгоритм і є універсальними обчислювальними пристроями. Тобто, якщо існує алгоритм, який може бути виконаний на будь-якому іншому обчислювальному пристрої, то існує еквівалентний алгоритм, який може бути виконаний на машині Тюрінга, або описаний з допомогою лямбда-числення Черча.

Теза Черча-Тюрінга — це фундаментальне поняття в теорії обчислень, яке висловлює ідею, що будь-яке обчислювальне завдання, яке може бути виконане, може бути виконане за допомогою машини Тюрінга або іншого обчислювального пристроя, який є еквівалентним машині Тюрінга. Іншими словами, теза Черча-Тюрінга підтверджує, що машини Тюрінга є універсальними обчислювальними пристроями, які здатні виконувати будь-яку обчислювальну задачу.

Аллан Тюрінг і Алонзо Черч були дослідниками, які незалежно відкрили свої моделі обчислень. Черч розробив λ -числення (лямбда-числення), а Тюрінг працював над машиною Тюрінга. Обидва вони прийшли до однакового висновку: усі обчислення можуть бути представлені за допомогою їхніх власних моделей, і ці моделі є рівносильними в термінах обчислювальної сили.

Теза Черча-Тюрінга не є математичною теоремою, але це припущення, яке підтримується загальним прийняттям серед вчених у галузі теорії обчислень. Вона вказує на те, що немає жодних більш потужних обчислювальних моделей, ніж машина Тюрінга. Тому, якщо щось може бути обчислено, це також може бути зроблено за допомогою машини Тюрінга.

Тезу Черча-Тюрінга можна розглядати як аксіому або природний відповідник обчислюваності, але вона сама не може бути математично доведена, оскільки вона є основним принципом теорії обчислюваності, на якому ґрунтуються вся теорія.

Принципи функціонального програмування: Концепції функціонального програмування також важливі для загального призначення комп'ютерів. Функціональне програмування базується на математичних функціях і використовує рекурсію для вирішення завдань. Воно дозволяє створювати алгоритми, які можуть бути застосовані до різних типів даних і завдань.

Завдяки цим концепціям і моделям, комп'ютери стають універсальними засобами для виконання різних обчислювальних завдань та алгоритмів. Ця універсальність лежить в основі розвитку сучасних обчислювальних систем і додала нові можливості для обчислень у різних галузях, включаючи науку, інженерію, медицину, бізнес і багато інших.

Перший збудований універсальний комп'ютер називався EDSAC. Комп'ютер EDSAC мав архітектуру фон Неймана та умовні оператори (оператор goto). Хоча слово комп'ютер було використано в абрівіатурі комп'ютера ENIAC 1945 року, ENIAC не мав програмного забезпечення, а перекомутувався.

У другому томі Оксфордського словника 1893 року за редакцією Джеймса Мюррея (1837—1915) слово "комп'ютер" означало людину, яка виконує математичні обчислення, переважно астрономів.

Що таке аналогова обчислювальна машина?

Аналогова обчислювальна машина використовує різні аналогові компоненти, такі як операційні підсилювачі, конденсатори та резистори, щоб моделювати електричні параметри інших систем. Вона працює з неперервними аналоговими сигналами, представляючи напругу та струм у фізичних величинах.

Сьогодні під словом комп'ютер мають на увазі цифровий комп'ютер, який може зберігати програму та є універсальним. Проте, існують так звані аналогові комп'ютери (обчислювальні машини), які мають суттєву відмінність від цифрових і є малоцікавими для теорії обчислень.

В аналогового комп'ютера немає етапу збору та формування даних, крім того, вони не універсальні. Коли цифровий комп'ютер обробляє якісь вхідні сигнали, його задача перетворити цей сигнал в інформацію, набір змістовних даних, зазвичай у двійковому форматі. Саме цей етап дає можливість застосовувати математичні принципи для опрацювання сигналів, вхідної інформації в цифрових комп'ютерах. Аналоговий комп'ютер натомість намагається прогнати вхідний сигнал через певний набір радіодеталей, щоб на виході отримати якийсь інший сигнал. Приклад простого аналогового комп'ютера це металевий стрижень, який при нагріванні змінює розмір. Цей стрижень репрезентує функцію розширення певного металу залежно від певної температури. Нагрійте стрижень до температури t і отримаєте значення функції $f(t) = l$, де l довжина стрижня при t . Цифрові комп'ютери набагато точніші й дають можливість застосовувати алгоритми до даних. В цифрових комп'ютерах обробка сигналу відбувається так: Вхідний сигнал → оцифрування → трансформація цифрових даних → вивід (відтворення цифрових даних)

Цифро-аналоговий перетворювач (ЦАП; DAC — Digital-to-Analog Convertor) — електронний пристрій для перетворення цифрового (як правило двійкового) сигналу на аналоговий. Пристрій, що виконує зворотну дію, називається аналогово-цифровим перетворювачем (АЦП).

Як правило ЦАП отримує на вхід цифровий сигнал в імпульсно-кодовій модуляції PCM (pulse-code modulation). Перетворення різних стиснутих форматів в PCM виконується відповідними кодеками. Кодек (кодер/декодер) — це пристрій або комп'ютерна програма, яка кодує або декодує потік даних або сигнал.

Імпульсно-кодова модуляція (ІКМ або PCM — Pulse Code Modulation) — процес перетворення аналогового сигналу на цифровий сигнал, коли через певні інтервали часу беруться відліки рівня аналогового сигналу і незалежно один від одного квантується і далі кодуються числами.

Теорема відліків (або теорема Найквіста) важлива для розуміння імпульсно-кодової модуляції (ІКМ) та інших систем передачі сигналів. Ця теорема встановлює зв'язок між частотою дискретизації сигналу та його максимальною частотою. Зазвичай формулюється так: частота дискретизації повинна бути не менше ніж двічі більша за максимальну частоту сигналу.

Терміни “аналоговий комп'ютер” та “цифровий комп'ютер” ввів американський інженер Джордж Стібіц (1904 – 1995).

Що таке машина Тюрінга?

Машина Тюрінга — це абстрактна математична модель, яка була запропонована британським математиком та логіком Алланом Тюрінгом в 1936 році. Вона є основною моделлю для дослідження обчислювальних процесів та визначення обчислювальної властивості проблем.

Машина Тюрінга складається з нескінченної стрічки, яка розділена на клітинки, де кожна клітинка може містити символ з деякого обмеженого алфавіту, та головки, яка може переміщатися вліво або вправо по стрічці та читувати або записувати символи. Машина Тюрінга також має внутрішній стан, який може змінюватися відповідно до правил, що задаються для конкретного обчислювального завдання.

Машина Тюрінга може виконувати різні види обчислень, включаючи арифметичні обчислення, логічні операції та багато інших. Вона може слугувати як уявна модель для розв'язання будь-якого обчислювального завдання та дослідження обчислювальної складності проблем.

Теорія машин Тюрінга була важливою для розвитку теорії обчислень та комп'ютерної науки, і вона є основою для розуміння того, що можливо імітувати за допомогою комп'ютерів та обчислювальних систем.

Математично описати машину Тюрінга можна так:

1. Алфавіт: Машина Тюрінга використовує алфавіт, який складається з обмеженого набору символів. Цей алфавіт може включати цифри, літери або інші символи.
2. Стрічка: Машина Тюрінга має нескінченну стрічку, розділену на клітинки. Кожна клітинка на стрічці може містити один символ з алфавіту.
3. Зчитувальна головка: Машина Тюрінга має головку, яка може читувати символи на поточній клітинці стрічки та записувати нові символи. Головка також може переміщатися вліво або вправо на одну клітинку за кожен такт.
4. Внутрішній стан: Машина Тюрінга має внутрішній стан, який визначає її поточну конфігурацію та поведінку. На кожному кроці обчислення, машина може змінювати свій внутрішній стан відповідно до правил, які задаються для конкретного завдання.
5. Правила переходу: Машина Тюрінга працює відповідно до правил переходу, які визначають, як вона повинна реагувати на символ, який вона знаходить на поточній клітинці та на свій поточний внутрішній стан. Правила переходу вказують, який символ записати на поточну клітинку, яким чином перемістити головку та який новий внутрішній стан встановити.

Алан Тюрінг у 1936 році розробив уявну обчислювальну машину, яку можна збудувати і яка здатна виконувати всі відомі сьогодні алгоритми. Машина Тюрінга являє собою нескінченну стрічку, розділену на клітинки. Ця стрічка виконує роль регістра, тобто пам'яті. У ці клітинки можна записувати символи з певного набору. Машина має блок управління та зчитувач, який може зчитувати одну клітинку за раз і переміщатись на одну клітинку вліво або вправо за один рух. Машина має регістр станів, який зберігає стан з певного набору станів. Машинна програма виконується блоком управління, вона також переміщує зчитувач. Програма машини Тюрінга складається з інструкцій, кожна з яких виглядає наступним чином: якщо машина знаходиться в стані r і читач бачить на стрічці символ n , то в цю клітинку запиши символ m , зроби крок вправо (або ліворуч), переведи стан машини в стан q або зупинися. З допомогою цієї машини Аллан Тюрінг довів негативну відповідь на проблему

зупинки та на проблему розв'язання. Тюрінг показав, що неможливо створити правильний алгоритм, використовуючи методи машини Тюрінга або лямбда-числення (читай будь-які мови програмування), які, аналізуючи програму для машини Тюрінга, могли б сказати, чи дастъ вона відповідь, чи зациклиться, тобто чи працюватиме вона нескінченно, безрезультатно. Тобто неможливо створити парсер, компілятор, який буде говорити для всіх програм, дастъ вона відповідь за скінчену кількість часу чи ні. Машину Тюрінга можна побудувати, якщо використовувати кінцеву стрічку, до того ж таких машин уже створено чимало, у різних варіаціях, механічних, пневматичних, гіdraulічних, електричних. Машина Тюрінга може мати кілька головок і навіть кілька стрічок, в залежності від творчих здібностей програміста, головне, щоб зберігався загальний принцип і строгість.

Скінчений автомат — це обчислювальний пристрій, який має скінченну кількість станів. Найпростіший скінчений автомат можна представити як вмікач, що має два стани: “увімкнений”, “вимкнений”, та одну команду яка змінює його стан. Якщо вмікач був у стані “вимкнений”, тоді команда зміни стану увімкне, наприклад, світло, а якщо він спочатку був у стані “увімкнений”, тоді навпаки, вимкне.

Складнішим скінченним автоматом була шифрувальна машина “Енігма”, яка мала клавіатуру для вводу, але результат вводу залежав від попередніх вводів, тобто якщо ви натискаєте літеру А на клавіатурі цієї машини декілька разів, то результат виводу може бути різний в залежності під попередніх натисків на клавіші (тобто в залежності від поточного стану машини Енігма). Скінчений автомат Мілі можна використати для опису машини Енігма.

Сучасні комп'ютери та машина Тюрінга не є скінченними автоматами, бо мають значно більші можливості, зокрема, можливість змінювати свою конфігурацію в процесі роботи. Проте, скінчений автомат може бути реалізований на машині Тюрінга.

Скінчений автомат можна описати математично за допомогою п'ятірки $(Q, \Sigma, \delta, q_0, F)$, де:

Q (множина станів): Q є скінченою множиною станів, які може приймати автомат.

Σ (алфавіт вхідних символів): Σ є скінченою множиною вхідних символів, які приймає автомат.

δ (функція переходів): Функція переходів $\delta: Q \times \Sigma \rightarrow Q$ визначає, який стан автомата переходить в наступний стан при отриманні певного вхідного символу. Для кожної пари (стан, символ) функція δ визначає новий стан.

q_0 (початковий стан): q_0 є одним зі станів з множини Q і вказує початковий стан автомата.

F (множина заключних станів): F є підмножиною множини станів Q і вказує, які стани є заключними (або приймаючими) станами автомата.

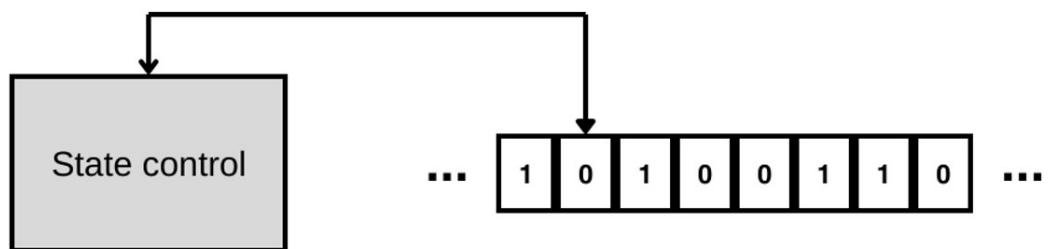
Зазвичай скінчений автомат представляється графічно, де вершини графа відповідають станам, ребра позначають переходи між станами з врахуванням вхідних символів, і позначені стани показують початковий та заключні стани.

Скінчений автомат виконує послідовність переходів з одного стану в інший відповідно до вхідних символів. Якщо автомат після обробки вхідної послідовності перебуває в одному зі станів, що належать до множини F , то він приймає вхідну послідовність; інакше він її відхиляє.

“Вступ до теорії автоматів, мов і обчислень” (1979) - впливовий підручник формальних мов та теорії алгоритмів написаний Джоном Гопкрофтом та Джейффрі Ульманом.

Складність кібернетичної системи визначається кількістю станів, які система може приймати та кількістю параметрів (зв'язків), які впливають на зміну стану.

Turing machine



Інтуїтивно машину Тюрінга можна описати так:

1. Дано стрічка (або реєстр), яка розділена на клітинки, в які можуть бути записані певні символи. (Стрічка може мати обмежену кількість клітинок, або бути теоретично нескінченною).
2. Дано зчитувальну головку (пристрій), яка рухається по стрічці й зчитує або записує символ в тій чи іншій клітинці на стрічці.
3. Для зберігання певних станів машини надається реєстр, тобто місце, в якому зберігається конкретний стан машини з набору доступних станів.
4. Дано модуль (блок), в якому зберігається та виконується дана програма.
5. Програма може лише:
 - а) змінити реєстр стану,
 - б) записати символ у клітинку, біля якої розташована головка для читання,
 - в) перемістити головку для читання на одну клітинку вправо або вліво.
- г) Зупини машину, стоп.
5. Вказано кінцевий набір символів, доступних для запису в клітинки стрічки.
6. Задано скінченну множину станів машини.
7. Встановлено програму, яка на основі поточного стану машини та символу, який зараз зчитує головка, визначає подальші дії машини (зміна станів, запис на стрічку, переміщення вправо, вліво). Конфігурація машини Тюрінга — це її поточний стан, положення головки та стан стрічки. Поточний рядок символів на стрічці називається аргументом машини Тюрінга.

Приклад налаштування машини Тюрінга.

Напишемо просту програму, яка змінить всі нулі, що на стрічці, на одиниці.

Символи на стрічці: #, 0, 1;

Стани машини: s_1, s_2 ;

Початкова конфігурація машини: $\langle s_1, |#[0]0|0|0|#| \rangle$, де '|#[0]0|0|0|#' символи на стрічці, [] - поточне положення головки;

Програма:

$s_1, 0 \rightarrow 1, L, s_1$;

$s_1, # \rightarrow #, R, s_2$;

$s_2, 1 \rightarrow 1, R, s_2$;

$s_2, # \rightarrow STOP$;

Де L – головка наліво, R – головка направо.

$(s_1, 0) \rightarrow (1, L, s_1)$ - означає, якщо поточний стан машини - s_1 , а поточний символ на стрічці - 0, то запишіть символ 1 замість 0 в клітинку, перемістіть вліво L (ліворуч) і змініть стан машини з s_1 на s_2 . Потім виконайте команду s_2 .

Програму можна встановити з допомогою табличного методу:

	0	1	#
s_1	$1, L, s_1$	STOP	$#, R, s_2$
s_2	STOP	$1, R, s_2$	STOP

Етапи виконання програми

1) $s_1: |#[0]0|0|0|#|$,

- 2) $s_1: \#|1|[0]|0|0|\#$,
- 3) $s_1: \#|1|1|[0]|0|\#$,
- 4) $s_1: \#|1|1|1|[0]|\#$,
- 5) $s_1: \#|1|1|1|1|[\#]$,
- 6) $s_2: \#|1|1|1|[1]|\#$,
- 7) $s_2: \#|1|1|[1]|1|\#$,
- 8) $s_2: \#|[1]|1|1|1|\#$,
- 9) $s_2: \#[1]|1|1|1|\#$,
- 10) $s_2: |[\#]|1|1|1|1|\#$,

Таким чином $\#|0|0|0|0|\#$ стало $\#|1|1|1|1|\#$. Це проста програма.

Приклад машини Тюрінга з двома зчитувальними головками.

На стрічці у нас є набір символів:

#,#,0,0,0,#,#.

Наша машина матиме дві зчитувальні головки, які разом зчитують два символи одночасно.

Налаштуємо програму на машині таким чином.

Програма 1:

Почніть справа: ##00[00]##.

00>01>стоп,
01>10>стоп,
10>11>стоп,
11>00>перемістіть (две головки) на комірку праворуч і перезапустіть програму.
##>01>стоп.

00>01 — тож якщо число з обох клітинок дорівнює 00, то запишіть 01. У нашій машині буде тільки один стан.

Ця програма збільшує двійкове число на 1.

Почніть справа: ##00[00]##.

00[00]>00[01], (0>1). 0000 = 2, 0001 = 1.

00[01]>00[10], (1>2).

00[10]>00[11], (2>3).

00[11]>[01]00, (3>4).

01[00]>01[01], (4>5).

01[01]>01[10], (5>6).

01[10]>01[11], (6>7).

01[11]>[10]00, (7>8).

$1000 > 10[01], (8 > 9)$.

$1001 > 10[10], (9 > 10)$.

$10[10] > 1011, (10 > 11)$.

$10[11] > [11]00, (11 > 12)$.

$11[00] > 11[01], (12 > 13)$.

$11[01] > 11[10], (13 > 14)$.

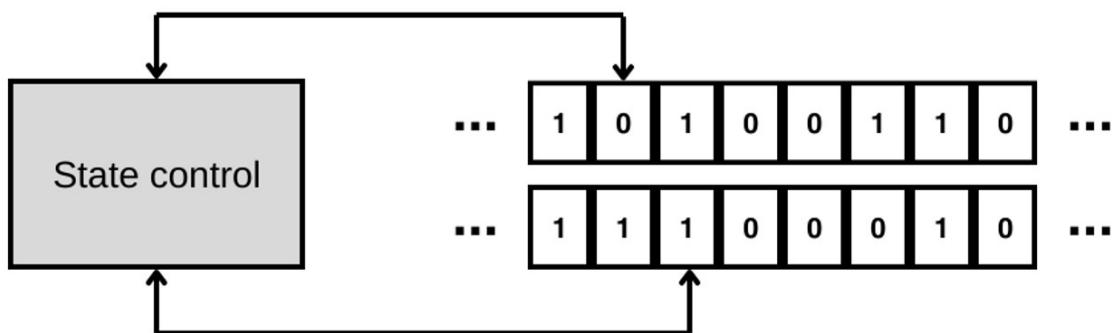
$11[10] > 11[11], (14 > 15)$.

$11[11] > [01]0000, (15 > 16)$.

В принципі, машина Тюрінга може мати кілька стрічок і одну головку для кожної стрічки. Також машина Тюрінга може мати кілька головок для однієї стрічки. Такі машини будемо називати покращеними або розширеними машинами Тюрінга.

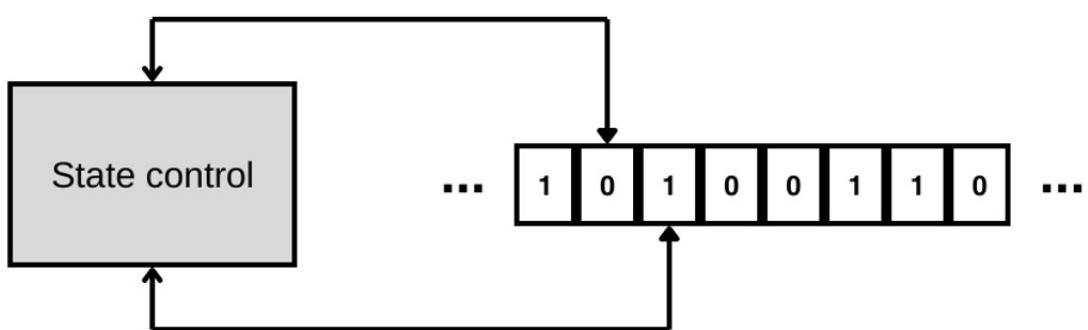
Багатострічкова машина Тюрінга

Multi-type Turing machine



Багатоголовкова машина Тюрінга

Multi-head Turing machine



Розглянемо програму для машини Тюрінга з двома зчитувальними головками та однією стрічкою.

Програма для перевірки дужок.

Суть програми полягає в тому, що на вхід машини Тюрінга ми даємо певний вираз із дужками (), і машина визначить, чи правильно розташовані дужки.

Правильно поставлені дужки у виразах:

0, (0), [0], {[][], 0}, {[0]}, {}.

Якщо після того, як машина Тюрінга обробить рядок введення, на її стрічці залишиться хоча б одна дужка, це означатиме, що вихідний вираз неправильний. Тобто, якщо на вхід машини подати рядок ()[], то вихід буде таким, тобто вихідний вираз неправильний, якщо буде, тоді все правильно.

Ми не будемо встановлювати саму програму, а лише опишемо хід (логіку) її виконання.

Припустимо, на машині Тюрінга задана послідовність символів:

{	()	([])	}
---	---	---	---	---	---	---	---

Машина використовує дві головки: <>, <>.

Якщо дві голови мають символи:

1. <[>, <]>,
2. <{>, <}>,
3. <(>, <)>,

тоді замініть їх точками та перемістіть обидві зчитувальні головки праворуч.

Якщо на головах щось інше, то просто перемістіть обидві голови вправо.

Робимо такі проходи (обробляємо), стільки разів, скільки символів на стрічці, і щоразу збільшуємо відстань між головами на одну клітинку.

Таблиця виконання:

1	$\langle \{ \rangle$	$\langle (\rangle$))	([]))	}
2	{	$\langle (\rangle$	$\langle) \rangle$	([])))	}
3	{	.	$\langle . \rangle$	$\langle (\rangle$	[])))	}
4	{	.	.	$\langle (\rangle$	$\langle [\rangle$])))	}
5	{	.	.	($\langle [\rangle$	$\langle] \rangle$)))	}
6	{	.	.	(.	$\langle . \rangle$	$\langle) \rangle$))	}
7	{	.	.	(.	.	$\langle) \rangle$	$\langle \} \rangle$	$\langle \} \rangle$	

Починаємо все спочатку, розсугаючи зчитувальні головки на одну клітинку.

8	$\langle \{ \rangle$.	$\langle . \rangle$	(.	.	.))	}
---	----------------------	---	---------------------	---	---	---	---	---	---	---

Ще кілька проходів зі збільшенням кількості клітинок між зчитувальними головками.

n	{	}
m

Довести правильність цього алгоритму можна методом математичної індукції.

Машина Тюрінга — це математична модель елементарної обчислювальної машини.

Дамо формальне визначення машини Тюрінга:

Машину Тюрінга ми визначаємо як п'ять об'єктів $[A, S, v, z, d]$, де:

- 1) A — скінчений алфавіт символів, які можна записати в клітинки, тобто $A = \{a_1, \dots, a_n\}$.
- 2) S — скінчена множина внутрішніх станів машини, тобто $S = \{s_1, \dots, s_n\}$.
- 3) v — функція від $S \times A$ до S , тобто $v: S \times A \rightarrow S$, де $S \times A$ декартів добуток множин S і A .
- 4) z — функція від $S \times A$ до A , тобто $z: S \times A \rightarrow A$, де $S \times A$ декартів добуток множин S і A .
- 5) d — функція від $S \times R$ для встановлення {Left (L), Right (R), STOP}.

Декартів добуток

$$A \times B := \{<x, y> : x \in A \wedge y \in B\}.$$

$$A \times B \times C := (A \times B) \times C.$$

$$A \times B \times C \times D = ((A \times B) \times C) \times D.$$

Якщо $A = \{1, 2\}$, $B = \{3, 4\}$, тоді $A \times B = \{\langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle\}$.

$A \times B \neq B \times A$.

Приклад.

Описана нижче машина Тюрінга читує вхідну послідовність нулів і одиниць, виводить E, якщо кількість одиниць парна, і O, якщо вона непарна. Перед рядком нулів і одиниць (тобто початкова конфігурація, або якщо ми маємо на увазі лише стрічку, то початковий аргумент) передують порожні клітинки, позначені #. Е або О друкуються в першій порожній клітинці після рядка введення. Таким чином, алфавіт має вигляд $A = \{\#, 0, 1, E, O\}$. Внутрішні стани: $S = \{s_0, s_1, s_2\}$; s_0 — початковий стан. Машина зупиняється за командою STOP.

v	z	d
$(s_0, 0) \rightarrow s_1,$	$(s_0, 0) \rightarrow 0,$	$(s_0, 0) \rightarrow L,$
$(s_0, 1) \rightarrow s_2,$	$(s_0, 1) \rightarrow 1,$	$(s_0, 1) \rightarrow L,$
$(s_1, 0) \rightarrow s_1,$	$(s_1, 0) \rightarrow 0,$	$(s_1, 0) \rightarrow L,$
$(s_1, 1) \rightarrow s_2,$	$(s_1, 1) \rightarrow 1,$	$(s_1, 1) \rightarrow L,$
$(s_2, 0) \rightarrow s_2,$	$(s_2, 0) \rightarrow 0,$	$(s_2, 0) \rightarrow L,$
$(s_2, 1) \rightarrow s_1,$	$(s_2, 1) \rightarrow 1,$	$(s_2, 1) \rightarrow L,$
$(s_0, \#) \rightarrow s_0,$	$(s_0, \#) \rightarrow \#,$	$(s_0, \#) \rightarrow L,$
$(s_1, \#) \rightarrow s_1,$	$(s_1, \#) \rightarrow E,$	$(s_1, \#) \rightarrow STOP,$
$(s_2, \#) \rightarrow s_2$	$(s_2, \#) \rightarrow O$	$(s_2, \#) \rightarrow STOP$

Приклад виконання.

Початкова конфігурація: $s_0 = |#[1]|0|1|0|\#|$.

Виконання програми:

- 1) $s_0: |#[1]|0|1|0|\#|$,
- 2) $s_2: |#[1]|0|1|0|\#|$,
- 3) $s_2: |#[1]|0|1|0|\#|$,

4) $s_1: \#|1|0|1|[0]\#$,

5) $s_1: \#|1|0|1|0|\#$,

6) $s_1: \#|1|0|1|0|[E]$,

Ми отримуємо:

$\#|1|0|1|0|\# \rightarrow \#|1|0|1|0|[E]$,

тобто кількість одиниць парна.

Алан Тюрінг дав відповідь на проблему зупинки, з якої слідує що: Не можливо створити алгоритм, чи програму, в класичному розумінні цих слів, які б могли аналізувати програмний код (алгоритм) й давати відповідь чи цей програмний код, чи алгоритм, дасть результат для певних даних, чи зациклиться, зависне на вічно. Отже, також не можна створити програму яка буде завжди правильно аналізувати алгоритм (програмний код) й визначати його часову складність, тобто складність алгоритму, й валідність коду. Сучасний аналізатор ChatGPT не може точно оцінювати валідність й складність алгоритмів, адже доведена ціла низка теорем, які заперечують таку можливість. Хоча проблема зупинки вказує на існування обмежень універсальних алгоритмів, це не означає, що неможливо створити програми, які здатні аналізувати код і визначати деякі його характеристики.

Проблема зупинки

Чи існує алгоритм, який визначає для будь-якої даної конфігурації машини Тюрінга, завершиться її виконання коли-небудь чи ні?

Конфігурація машини Тюрінга — це налаштування машини Тюрінга, тобто її програма, початковий стан та вхідні дані.

Теорема. Проблема зупинки нерозв'язна.

Немає жодного алгоритму (програми), який може визначити, чи зупиняється дана машина Тюрінга, чи ні.

Доказ.

Ми не можемо просто запустити машину Тюрінга і чекати, поки вона зупиниться, тому що вона може зациклитись та працювати вічно, тому ми повинні визначити її зупинку, аналізуючи її програми та вхідні дані (аргумент).

Припустимо, що у нас є алгоритм, який визначає, чи зупиниться машина Тюрінга з певною конфігурацією чи ні, позначимо це як $p(x, y)$, де замість змінної x задається конкретна програма машини, а замість y — конкретні вхідні дані машини. Ми теоретично можемо призначити цей алгоритм p машині Тюрінга, тому ми вкажемо конфігурацію машини для тестування як рядок (на стрічці).

Припустимо, що у нас є машина Тюрінга з програмою, яка реалізує алгоритм p , який перевіряє, чи зупиняється (інша) машина Тюрінга з заданою програмою та вхідними даними. Наприклад, якщо ми хочемо перевірити, чи зупинить машину Тюрінга з програмою t і початковими даними g , позначимо її конфігурацію як (t, g) . Якщо виконання (t, g) зупинено, то пишемо $p(t, g) = 1$, якщо ні, то $p(t, g) = 0$.

Визначимо таку програму:

$q(x) := \text{if}(p(x, x)=0) \text{ стоп; інакше робити щось назавжди;}$
 $p(q, q);$

Функція $p(q, q)$ не зможе дати правильний результат, а це означає, що наше початкове припущення про існування універсального алгоритму визначення зупинки машини Тюрінга є невірним. Оскільки сучасний комп'ютер є універсальною машиною Тюрінга, для нього не існує алгоритму, щоб визначити, чи він зупиняється. Універсальна машина Тюрінга — це машина Тюрінга програму і вхідні дані якої можна змінювати.

Теорема зупинки передбачає наступну теорему: Існує рекурсивно перерахована множина, яка не є рекурсивною.

Множина називається розв'язуваною або рекурсивною, якщо існує алгоритм, який, отримавши елемент на вхід, завершує виконання через кінцеву кількість кроків і визначає, чи належить цей елемент даній множині.

Що таке клітинний автомат?

Клітинний автомат — це математична модель, яка складається з сітки або набору клітинок, кожна з яких може перебувати в одному з обмежених станів. Клітини спостерігають за станом своїх сусідніх клітин і відповідають на цей стан згідно з певними правилами. Клітинний автомат розглядається в дискретному часі, тобто визначені кроки часу, під час яких змінюється стан клітин.

Найвідоміший клітинний автомат — це "Гра життя" (Game of Life), який був розроблений математиком Джоном Конвеєм у 1970 році. У цій моделі клітини розташовані на двовимірній решітці, і вони можуть бути у двох станах: живі або мертві. Стан клітини змінюється в залежності від кількості сусідніх живих клітин за певних правил.

Клітинні automati використовуються в різних областях, таких як обробка зображень, криптографія, симуляція природних явищ (взаємодія мікроорганізмів), моделювання складних систем, і штучний інтелект. Вони є потужним інструментом для вивчення емерджентної поведінки та комплексних системних властивостей.

Математичний опис клітинного автомата містить:

1. Сітку клітин: Це просторова структура, в якій розташовані клітини. Сітка може бути одновимірною, двовимірною або багатовимірною, в залежності від конкретної моделі клітинного автомата.
2. Стани клітин: Кожна клітина може перебувати в одному з обмежених станів. Зазвичай це два стани: "живий" і "мертвий", але існують моделі з більшою кількістю можливих станів.
3. Правила переходу: Це набір правил, які визначають, як змінюються стани клітин в кожен момент часу. Правила вказують, як клітини реагують на стани своїх сусідніх клітин. Ці правила можуть бути виражені у вигляді логічних виразів або таблиць переходів.
4. Початковий стан: Початковий стан сітки клітин визначається перед початком симуляції. Це вихідний стан, з якого розпочинається розвиток системи.
5. Крок часу: Клітинний автомат розглядається в дискретному часі, тобто визначені кроки часу, на кожному з яких відбуваються зміни стану клітин.

Математичний опис полягає в тому, щоб формалізувати всі ці компоненти, а також визначити, як клітини взаємодіють між собою та як вони еволюціонують з часом. Математичні рівняння або алгоритми можуть бути використані для опису цих процесів.

У книзі “Клітинні автомати” (1968) Едгар Кодд пояснює, як комп’ютер, який здатний обчислювати всі обчислювані функції, може бути сконструйований з великої кількості простих ідентичних комірок, кожна з яких взаємодіє лише зі своїми безпосередніми сусідами.

Клітинний автомат — це клітинний автомат, розроблений британським вченим Едгаром Коддом у 1968 році. Він був розроблений для відтворення обчислювальної та конструкційної універсальності клітинного автомата фон Неймана, але з меншою кількістю станів: 8 замість 29.

Клітинний автомат — це дискретна модель обчислень, що вивчається в теорії автоматів.

Двовимірний клітинний автомат — це набір однакових комірок, які можуть перебувати в певних станах. Комірка, залежно від її поточного стану та стану оточуючих комірок, може перейти до наступного стану відповідно до визначеного програми. Принцип роботи такого автомата схожий на той, що використовується в машині Тюрінга. Поле клітинного автомата може складатися з квадратних або шестикутних осередків. Клітинний автомат складається з регулярної сітки комірок, кожна з яких знаходиться в одному з кінцевої кількості станів, наприклад увімкнено та вимкнено. Сітка може бути будь-якої кінцевої кількості вимірів. Для кожної комірки визначено набір комірок, які називають її околицями, відносно вказаної комірки. Початковий стан вибирається шляхом призначення стану для кожної комірки. Створюється нове покоління згідно з деяким фіксованим правилом, яке визначає новий стан кожної клітинки з точки зору поточного стану клітинки та станів клітинок, що знаходяться поблизу. Зазвичай, правило оновлення стану комірок однакове для кожної клітинки та не змінюється з часом, і застосовується до всієї сітки одночасно, хоча відомі винятки, наприклад, стохастичний клітинний автомат і асинхронний клітинний автомат.

Концепція клітинних автоматів була спочатку відкрита в 1940-х роках Станіславом Уламом і Джоном фон Нейманом, коли вони були працівниками Лос-Аламосської національної лабораторії.

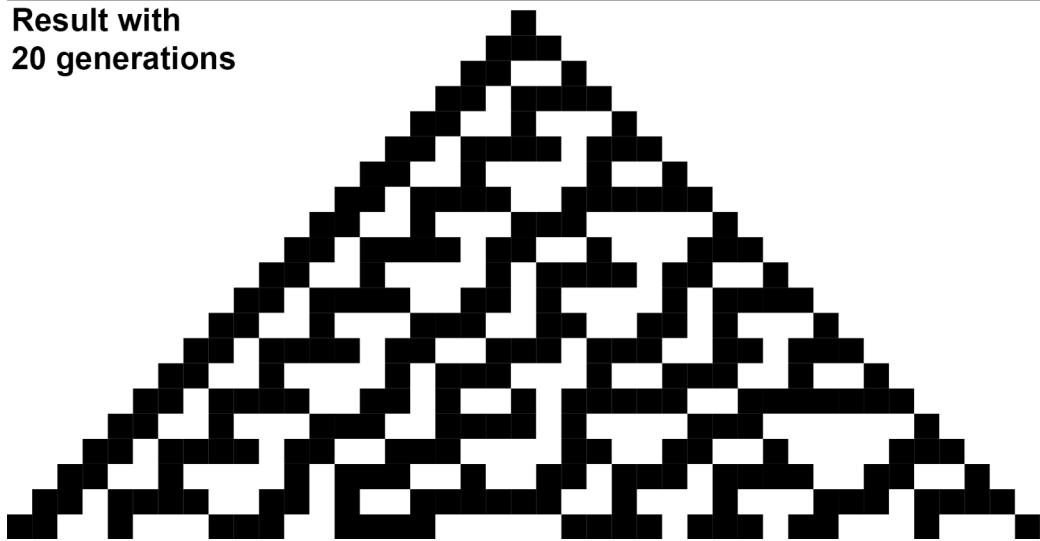
“Гра життя” — це клітинний автомат, розроблений британським математиком Джоном Хортоном Конвеєм у 1970 році. Це гра з нульовими гравцями, а це означає, що її еволюція визначається її початковим станом і не потребує подальшого введення. Людина взаємодіє з “Грою життя”, створюючи початкову конфігурацію та спостерігаючи, як вона розвивається. Клітинний автомат “Гра життя” повний за Тюрінгом і може моделювати машину Тюрінга. Гра вперше з’явилася публічно в жовтневому номері журналу “Саентіфік амерікан” за 1970 рік у колонці Мартіна Гарднера “Математичні ігри”. Теоретично, “Гра життя” має силу універсальної машини Тюрінга: все, що можна обчислити алгоритмічно, можна обчислити на ній. У 1980-х британський програміст Стівен Вольфрам зайнявся систематичним вивченням елементарних клітинних автоматів. Його асистент Метью Кук показав, що одне з правил Вольфрама для клітинних автоматів є повним за Тюрінгом.

Стівен Вольфрам — британський програміст, єврейського походження. Розробник системи комп’ютерної алгебри “Вольфрам математика”.

Rule 30

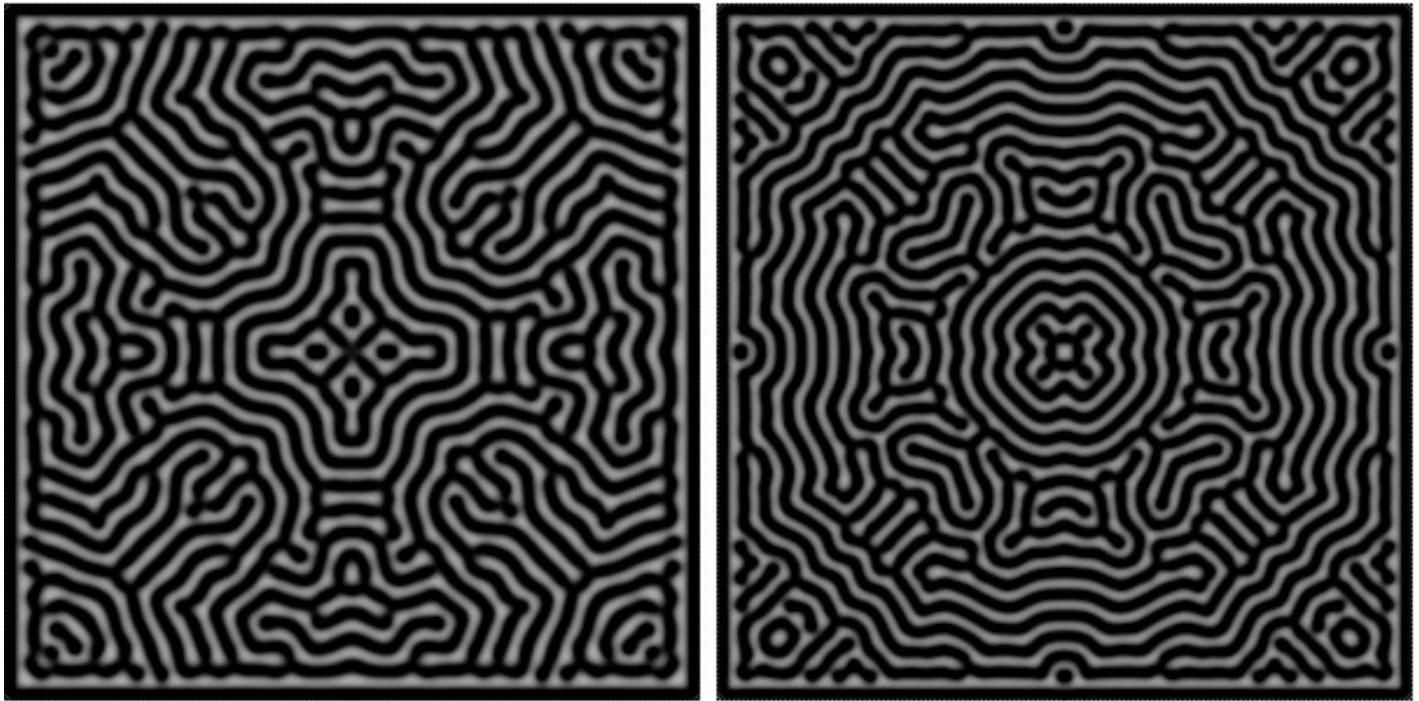
								
0	0	0	1	1	1	1	0	

**Result with
20 generations**



На малюнку зображений одновимірний клітинний автомат. На початку було задано правило та 1 заповнена клітинка, з якої утворилось 20 поколінь клітинок. Перевірка відбувається по три клітинки за раз вздовж одного ряду. Правило каже як має фарбуватися клітинка (знизу) відповідно до трьох клітинок попереднього покоління (зверху).

Візерунки Тюрінга



Англійський математик Алан Тюрінг вивчав утворення візерунків (патернів) в живій природі, зокрема рибах, ящірках тощо.

Процес утворення візерунків Тюрінга:

Уявіть собі плоску поверхню, як аркуш паперу. Спочатку ви рівномірно розливаєте два різних чорнила (які представляють хімічні речовини A і B) по всьому паперу. З часом чорнила взаємодіють і розповсюджуються (дифундують). Якщо чорнила слідуватимуть правилам утворення тюрінгових візерунків (одне розповсюджується швидше за інше та їхні реакції взаємно регулюються), ви можете побачити плями, смуги або інші складні візерунки, які з'являються на папері.

- На початку два чорнила рівномірно змішані по всій поверхні паперу. Невеликі випадкові коливання або флуктуації в концентрації чорнил з'являються через незначні нерівності в розподілі.

- Активатор (чорнило A) починає стимулювати виробництво активатора й інгібітора (чорнила B). Тим часом інгібітор (чорнило B) пригнічує виробництво активатора (чорнила A).

- Інгібітор дифундує (розповсюджується) швидше, ніж активатор.

- Завдяки швидкій дифузії інгібітора, він швидко розповсюджується і пригнічує активатор в його околицях.

- Це призводить до локальних піків концентрації активатора в областях, де його концентрація спочатку була трохи вищою за середню, оскільки повільна дифузія дозволяє йому залишатися більш концентрованим.

- Швидка дифузія інгібітора створює западини навколо цих піків, не дозволяючи активатору занадто далеко розповсюджуватись.

- Зі зростанням цього процесу початкові малі коливання посилюються і стабілізуються у чіткі візерунки.

В оригінальній публікації Тюрінга 1952 року йшлося про хімічні речовини X та Y, які перетікають від клітини з меншою концентрацією до клітини з більшою концентрацією відповідно до коефіцієнтів дифузії.

Що не може універсальний комп'ютер?

Універсальний комп'ютер, такий як загальний комп'ютер Тюрінга, не може виконати завдання, для якого не існує алгоритму, який би його розв'язав. Це випливає з обмежень теорії обчислюваності та теореми Тюрінга.

10-та проблема Гільберта є дуже хорошим прикладом такої ситуації. Ця проблема полягає в тому, щоб визначити, чи має діофантове рівняння загальний розв'язок (розв'язки в цілих числах). Важливо підкреслити, що для цієї проблеми не існує загального алгоритму, який би дозволив вирішити її для будь-якого можливого діофантового рівняння.

Теорема про зупинку також важлива, оскільки вона демонструє, що не існує загального алгоритму для визначення, чи завершиться виконання будь-якого даного алгоритму. Це робить деякі питання про обчислювальну складність нерозв'язними. Наприклад, не можна завжди точно визначити, чи буде алгоритм працювати нескінченно довго або завершиться.

Таким чином, обмеження універсальних комп'ютерів вказують на існування обмежень у самій теорії обчислюваності та показують, що не для всіх завдань існують загальні алгоритми, які б їх вирішували.

Машина Тюрінга — це математична модель елементарної обчислювальної машини.

Дамо формальне визначення машини Тюрінга:

Машину Тюрінга ми визначаємо як п'ять об'єктів $[A, S, v, z, d]$, де:

- 1) A — скінчений алфавіт символів, які можна записати в клітинки, тобто $A = \{a_1, \dots, a_n\}$.
- 2) S — скінчена множина внутрішніх станів машини, тобто $S = \{s_1, \dots, s_n\}$.
- 3) v — функція від $S \times A$ до S , тобто $v: S \times A \rightarrow S$, де $S \times A$ декартів добуток множин S і A .
- 4) z — функція від $S \times A$ до A , тобто $z: S \times A \rightarrow A$, де $S \times A$ декартів добуток множин S і A .
- 5) d — функція від $S \times R$ для встановлення {Left (L), Right (R), STOP}.

Декартів добуток

$$A \times B := \{<x, y> : x \in A \wedge y \in B\}.$$

$$A \times B \times C := (A \times B) \times C.$$

$$A \times B \times C \times D = ((A \times B) \times C) \times D.$$

Якщо $A = \{1, 2\}$, $B = \{3, 4\}$, тоді $A \times B = \{\langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle\}$.

$A \times B \neq B \times A$.

Приклад.

Описана нижче машина Тюрінга читує вхідну послідовність нулів і одиниць, виводить E, якщо кількість одиниць парна, і O, якщо вона непарна. Перед рядком нулів і одиниць (тобто початкова конфігурація, або якщо ми маємо на увазі лише стрічку, то початковий аргумент) передують порожні клітинки, позначені #. Е або О друкуються в першій порожній клітинці після рядка введення. Таким чином, алфавіт має вигляд $A = \{\#, 0, 1, E, O\}$. Внутрішні стани: $S = \{s_0, s_1, s_2\}$; s_0 — початковий стан. Машина зупиняється за командою STOP.

v	z	d
$(s_0, 0) \rightarrow s_1,$	$(s_0, 0) \rightarrow 0,$	$(s_0, 0) \rightarrow L,$
$(s_0, 1) \rightarrow s_2,$	$(s_0, 1) \rightarrow 1,$	$(s_0, 1) \rightarrow L,$
$(s_1, 0) \rightarrow s_1,$	$(s_1, 0) \rightarrow 0,$	$(s_1, 0) \rightarrow L,$
$(s_1, 1) \rightarrow s_2,$	$(s_1, 1) \rightarrow 1,$	$(s_1, 1) \rightarrow L,$
$(s_2, 0) \rightarrow s_2,$	$(s_2, 0) \rightarrow 0,$	$(s_2, 0) \rightarrow L,$
$(s_2, 1) \rightarrow s_1,$	$(s_2, 1) \rightarrow 1,$	$(s_2, 1) \rightarrow L,$
$(s_0, \#) \rightarrow s_0,$	$(s_0, \#) \rightarrow \#,$	$(s_0, \#) \rightarrow L,$
$(s_1, \#) \rightarrow s_1,$	$(s_1, \#) \rightarrow E,$	$(s_1, \#) \rightarrow STOP,$
$(s_2, \#) \rightarrow s_2$	$(s_2, \#) \rightarrow O$	$(s_2, \#) \rightarrow STOP$

Приклад виконання.

Початкова конфігурація: $s_0: \#|1|0|1|0|\#$.

Виконання програми:

- 1) $s_0: \#|1|0|1|0|\#$,
- 2) $s_2: \#|1|0|1|0|\#$,
- 3) $s_2: \#|1|0|1|0|\#$,

4) $s_1: \#|1|0|1|[0]\#$,

5) $s_1: \#|1|0|1|0|\#$,

6) $s_1: \#|1|0|1|0|[E]$,

Ми отримуємо:

$\#|1|0|1|0|\# \rightarrow \#|1|0|1|0|[E]$,

тобто кількість одиниць парна.

Алан Тюрінг дав відповідь на проблему зупинки, з якої слідує що: Не можливо створити алгоритм, чи програму, в класичному розумінні цих слів, які б могли аналізувати програмний код (алгоритм) й давати відповідь чи цей програмний код, чи алгоритм, дасть результат для певних даних, чи зациклиться, зависне на вічно. Отже, також не можна створити програму яка буде завжди правильно аналізувати алгоритм (програмний код) й визначати його часову складність, тобто складність алгоритму, й валідність коду. Сучасний аналізатор ChatGPT не може точно оцінювати валідність й складність алгоритмів, адже доведена ціла низка теорем, які заперечують таку можливість. Хоча проблема зупинки вказує на існування обмежень універсальних алгоритмів, це не означає, що неможливо створити програми, які здатні аналізувати код і визначати деякі його характеристики.

Проблема зупинки

Чи існує алгоритм, який визначає для будь-якої даної конфігурації машини Тюрінга, завершиться її виконання коли-небудь чи ні?

Конфігурація машини Тюрінга — це налаштування машини Тюрінга, тобто її програма, початковий стан та вхідні дані.

Теорема. Проблема зупинки нерозв'язна.

Немає жодного алгоритму (програми), який може визначити, чи зупиняється дана машина Тюрінга, чи ні.

Доказ.

Ми не можемо просто запустити машину Тюрінга і чекати, поки вона зупиниться, тому що вона може зациклитись та працювати вічно, тому ми повинні визначити її зупинку, аналізуючи її програми та вхідні дані (аргумент).

Припустимо, що у нас є алгоритм, який визначає, чи зупиниться машина Тюрінга з певною конфігурацією чи ні, позначимо це як $r(x, y)$, де замість змінної x задається конкретна програма машини, а замість y — конкретні вхідні дані машини. Ми теоретично можемо призначити цей алгоритм r машині Тюрінга, тому ми вкажемо конфігурацію машини для тестування як рядок (на стрічці).

Припустимо, що у нас є машина Тюрінга з програмою, яка реалізує алгоритм r , який перевіряє, чи зупиняється (інша) машина Тюрінга з заданою програмою та вхідними даними. Наприклад, якщо ми хочемо перевірити, чи зупинить машину Тюрінга з програмою t і початковими даними g , позначимо її конфігурацію як (t, g) . Якщо виконання (t, g) зупинено, то пишемо $r(t, g) = 1$, якщо ні, то $r(t, g) = 0$.

Визначимо таку програму:

$q(x) := \text{if}(r(x, x)=0) \text{ стоп; інакше робити щось назавжди;}$
 $p(q, q);$

Функція $p(q, q)$ не зможе дати правильний результат, а це означає, що наше початкове припущення про існування універсального алгоритму визначення зупинки машини Тюрінга є невірним. Оскільки сучасний комп'ютер є універсальною машиною Тюрінга, для нього не існує алгоритму, щоб визначити, чи він зупиняється. Універсальна машина Тюрінга — це машина Тюрінга програму і вхідні дані якої можна змінювати.

Теорема зупинки передбачає наступну теорему: Існує рекурсивно перерахована множина, яка не є рекурсивною.

Теорема Гьоделя про неповноту

Визначимо формальну систему Z для арифметики натуральних чисел.

Аксіоми для Z :

1. $\forall x, y \exists! z (x + y = z)$;
2. $\forall x, y \exists! z (x * y = z)$;
3. $\forall x (x + 0 = x) \wedge (x * 1 = x)$;
4. $\forall x, y (x + (y + 1) = (x + y) + 1)$;
5. $\forall x, y (x * (y + 1) = (x * y) + x)$;
6. $\forall x, y ((x + 1 = y + 1) \rightarrow x = y)$;
7. $\forall x (\neg x + 1 = 0)$.
8. $\forall t_1, \dots, t_k ((A_m(0, t_1, \dots, t_k) \wedge \forall y (A_m(y, t_1, \dots, t_k) \rightarrow A_m(y + 1, t_1, \dots, t_k))) \rightarrow \forall x A_m(x, t_1, \dots, t_k))$.

Формула 8 — це схема математичної індукції, тобто схема аксіом, яка визначає нескінчений набір аксіом, кожна з яких є окремим випадком математичної індукції, для певної формули.

Перша теорема Гьоделя про неповноту.

У системі Z є така пропозиція A , що ні A , ні $\neg A$ не можуть бути доведені з допомогою аксіом з Z , якщо система Z несуперечлива (узгоджена, послідовна).

(Це справедливо для будь-якої системи, яка включає систему Z , тобто для будь-якої системи з виразною силою Z).

Ескіз доведення.

Суть методу доведення першої теореми про неповноту Геделя полягає в тому, що ми перекладаємо (гіпотетично) всі твердження, які можна сформулювати в рамках системи Z , тобто набору рядків символів мови логіки (наприклад, квантори, змінні), у набір рядків чисел. Таким чином, ми однозначно кодуємо весь список значущих рядків у системі Z у список рядків чисел. Ці рядки самі є натуральними числами. Кожне таке число є закодованим твердженням, зробленим методами системи Z . Ми можемо визначити рекурсивні функції на машині Тюрінга, яка перевірить, чи виводиться конкретне число зі списку закодованих висловлювань системи Z з аксіом цього формальної системи чи ні. На додаток, твердження, сформульовані методами мови системи Z , не обов'язково доводяться з її аксіом. Системна мова Z — це арифметична мова першого порядку, яка включає логіку предикатів і здатна виражати рекурсивні функції. Але якщо машина Тюрінга перевіряє закодовані рядки з допомогою рекурсивних функцій, тоді в одному з рядків з'являється пропозиції про ту саму рекурсивну функцію, яку виконує сама машина. І ми отримаємо самопосилання, як і в попередній теоремі про зупинку, тобто ми отримаємо, що у випадку якщо твердження є доказовим, тоді воно також є недоказове, і навпаки. Припустимо, ми хочемо встановити машину Тюрінга, яка визначатиме, чи виводиться даний рядок символів (тобто формула) із системи аксіом Z чи ні. Системи Z достатньо для вираження рекурсивних функцій, в рамках яких можна формально задати ту саму машину Тюрінга, що приведе до протиріччя, як у випадку функції $q(x)$, що фігурує в теоремі зупинки. Трохи змінивши функцію $q(x)$, ми отримаємо функцію, яка якщо доказова, то недоказова, і навпаки. Ця функція виступає як недоказова пропозиція A . Оскільки вся перевірка заснована на рекурсивних функціях, які працюють лише з натуральними числами (і нулем), то ці функції, виконавцем яких є машина Тюрінга або комп'ютер, можуть розпізнавати рядки логічних символів, вони, як уже згадувалося, повинні бути однозначно закодовані в натуральні числа. Гьодель запропонував власний метод кодування, а саме: кожному символу мови нашої формальної системи ми призначаємо просте число ($2, 3, 5, 7, 11, \dots$), оскільки формулі складаються з набору цих логічних знаків, формулі будуть закодовані як добутки простих чисел і розшифровані методом розкладання на прості множники.

Теорема Черча про нерозв'язність логіки предикатів.

Універсального алгоритму (процедури) для визначення того, чи є формула обчислення предикатів істинною (або хибною), не існує.

Доказ. У теоремі про зупинку машини Тюрінга ми показали, що можна створити такий вислів, який неможливо розв'язати, такий вислів також можна сформувати в рамках логіки предикатів.

Теорема Матіясевича–Робінсона–Девіса–Патнема

Універсального алгоритму для визначення того, чи можна розв'язати Діофантове рівняння в цілих числах (додатних і від'ємних), не існує.

Доказ.

Попередні поняття.

Діофантове рівняння — це поліноміальне рівняння, яке зазвичай складається з двох або більше невідомих, таке, що вивчаються лише цілі розв'язки (ціличисельне рівняння таке, що всі невідомі приймають цілі значення). Найпростіше лінійне діофантове рівняння має вигляд $ax + by = c$, де a, b і c задані цілі числа. Загальне Діофантове рівняння є рівнянням вигляду:

$$P(x_1, \dots, x_m) = 0,$$

де P — ціла функція (наприклад, поліном з цілими коефіцієнтами), а змінні x_i приймають цілі значення.

Множина всіх елементів, які можуть бути згенеровані деяким точно заданим алгоритмом (який можна запустити на машині Тюрінга), називається перелічуваною множиною.

Теорема. Існують перераховані множини, які не рекурсивні (обчисленні).

Доказ. Доказ цього факту ми навели в теоремі зупинки, де показали, як можна побудувати таку множину.

Множина називається розв'язуваною або рекурсивною, якщо існує алгоритм, який, отримавши елемент на вході, завершує виконання через кінцеву кількість кроків і визначає, чи належить цей елемент даній множині. Іншими словами, множина розв'язувана (обчислена), якщо її характеристична функція обчислюється. Характеристична функція множини A - це функція, яка може для даного елемента точно визначити, належить він до множини A чи ні.

Множина зліченна, якщо вона скінчена або її можна поставити у однозначну відповідність множині натуральних чисел.

Рекурсивно перерахована множина — множина конструктивних об'єктів (наприклад, натуральних чисел), всі елементи якої можуть бути отримані за допомогою деякого алгоритму.

Теорема. Обчисленна множина (рекурсивна множина) завжди зліченна і перерахована.

Доказ. Те, що обчисленна множина є зліченою і перерахованою, випливає з методу її побудови, а саме, з використання зліченої кількості рекурсивних функцій.

Діофантова множина — це множина, що складається з упорядкованих множин з n цілих чисел, для яких існує алгебраїчне діофантове рівняння:

$P(a_1, \dots, a_n, x_1, \dots, x_m) = 0$, яка розв'язна тоді й тільки тоді, коли множина чисел (a_1, \dots, a_n) належить до цієї множини. (a_1, \dots, a_n) — параметри, (x_1, \dots, x_m) — корені.

Теорема Девіса (Наведено без доведення).

Будь-який перерахований набір цілих чисел є діофантовим, а будь-який діофантовий набір є перерахованим.

\Leftrightarrow - еквівалентність.

Доказ теореми Матіясевича–Робінсона–Девіса–Патнема

Припустимо, що у нас є програма (алгоритм) S , яка визначає за будь-яким діофантовим рівнянням M_k , чи є воно розв'язним у цілих числах чи ні. Якщо рівняння не розв'язується, то програма S виводить номер цього рівняння k , інакше вона нічого не друкує. Таким чином, проходячи через усі рівняння, програма видасть певну множину M_S з номерами всіх нерозв'язних рівнянь (якщо таких рівнянь немає, то порожня множина). Але оскільки ця множина M_S є перерахованім набором цілих чисел, то,

за теоремою Девіса, існує діофантове рівняння M_{ks} , яке визначає цю множину M_s . Отже, рівняння M_{ks} розв'язне, оскільки існує певний його набір. Але що дасть програма S , якщо застосувати до її входу те саме рівняння M_{ks} ? Припустимо, рівняння не розв'язне і програма S виводить число ks , але тоді множини M_s будуть містити число ks , отже, рівняння M_{ks} розв'язне для цього числа, тобто ми отримуємо протиріччя. Якщо програма S нічого не друкує, то рівняння M_{ks} розв'язне, але оскільки програма нічого не друкує, множина M_{ks} порожня, отже, Діофантове рівняння M_{ks} не розв'язується, знову ж таки протиріччя. Отже, алгоритм не існує.

$M_k(k,x)$ - рівняння з індексом k , у списку всіх діофантових рівнянь. У цьому випадку у ролі параметра передається число k , x – корінь для розв'язування рівняння, який необхідно знайти.

S — алгоритм, який визначає, чи розв'язане рівняння.

$S(M_k(k,x) = 0) \Leftrightarrow k \in M_s$ - означає, що якщо рівняння M_k не має цілого кореня x , при якому воно дорівнює 0, то програма S каже 0, тобто що рівняння не розв'язується в цілих числах, і виводить номер рівняння k , тобто, встановлює набір M_s .

Якщо:

$$(S(M_k(k,x) = 0) = 0) \Leftrightarrow (k \in M_s) \Leftrightarrow \exists x(M_s(k,x) = 0) \Leftrightarrow \exists x(M_{ks}(k,x) = 0),$$

тоді:

$(S(M_{ks}(k,x) = 0) = 0) \rightarrow (k \in M_s) \rightarrow \exists x(M_{ks}(k,x) = 0))$; Протиріччя, тобто є конкретне Діофантове рівняння, яке не можна розв'язати з допомогою алгоритму.

Теорема Матіясевича–Робінсона–Девіса–Патнема доведена.

Складність за Колмогоровом, позначена $K(x)$, — це довжина найкоротшої можливої комп'ютерної програми (на деякій фіксованій мові програмування), яка виводить певний рядок x , а потім зупиняється. Вона представляє кількість інформації, необхідної для опису рядка в найбільш стислий спосіб.

В алгоритмічній теорії інформації складність об'єкта за Колмогоровом, наприклад фрагмента тексту, — це довжина найкоротшої комп'ютерної програми (на заздалегідь визначеній мові програмування), яка виводить об'єкт (текст) на вихід, тобто генерує його. Аксіоматичний підхід до складності Колмогорова розробив радянський математик Андрій Колмогоров (1903 — 1987) в 1964 році.

Розглянемо наступні два рядки з 15 малих літер:

"абвабвабвабв", і "аквбтсабвцбврпл".

Перший рядок має короткий англомовний опис, а саме "write abv 5 times", який складається з 17 символів. Другий не має очевидного простого опису, окрім запису самого рядка, тобто "write akvbtsabvcbvrpl", який містить 21 символ. Тому можна сказати, що операція запису первого рядка має "меншу складність", ніж запис другого.

Алгоритмічне обчислення колмогорівської складності довільного рядка є нерозв'язним завданням. Важливі практичні наслідки цього твердження для області програмування:

неможливість створення ідеального архіватора, який формує для будь-якого вхідного файлу програму найменшого можливого розміру, що друкує цей файл,

неможливість створення ідеального компілятора, що оптимізує за розміром початковий код.

Завдання обчислення алгоритмічної складності (складність об'єкта за Колмогоровом) довільного рядка є алгоритмічно нерозв'язним — не існує універсальної програми, яка приймала б на вхід рядок і видавала б на виході ціле число, яке вказує на його алгоритмічну складність. Показати це можна за допомогою суперечності шляхом створення програми, яка створює рядок, створити який можливо лише довшою програмою.

Подумайте про наявність набору квадратних плиток, які можна використовувати для покриття площини, але для з'єднання плиток між собою потрібно використовувати специфічні взаємодії, аналогічні пазлам (тобто дві плитки з набору можна об'єднати тільки при наявності в них відповідних сторін). Питання таке: чи існує алгоритм, який може визначити, чи можливо використовувати даний набір плиток для покриття площини? Математик Роберт Бергер в 1966 році довів, що не існує загального алгоритму для розв'язання цього завдання, продемонструвавши аперіодичні плитки. Роберт Бергер, народився в 1938 році, є відомим прикладним математиком і здобув славу завдяки відкриттю першої аперіодичної плитки.

Наразі наші комп'ютери не можуть створювати візуальні ефекти, які ідеально копіюють реальність, оскільки їм не вистачає швидкості обробки та обчислювальної потужності. Навіть невеликий матеріальний об'єкт складається з мільярдів атомів, які рухаються й випромінюють хвилі, що потребує величезного обсягу обчислень для точного відтворення їхньої поведінки.

Зараз створюють комп'ютерні ігри, які являють собою цілі віртуальні світи, по яких можна бродити, але необхідно зауважити, що якщо віртуальний світ побудований на основі скінченої кількості універсальних фізичних законів, тоді потрібно бути впевненим, що вони несуперечливі, бо інакше можуть виникнути помилки та аномалії в цьому світі. Довести несуперечливість системи законів може бути важко, або навіть неможливо (теорема Гьоделя). Створити великий віртуальний світ на основі універсальних законів важко, бо необхідно знати чи ці закони консистентні.

Лейбніц був дуже впевнений, що алгоритм є майже для всього, але це не так. Проте Готфрід Лейбніц казав, що алгоритм має бути простіший ніж задача, яку він вирішує, інакше з нього ніякої користі. Виявилось так, що існує багато алгоритмічно нерозв'язних проблем. Наприклад, Лейбніц надіявся, що буде знайдена загальна формула для розв'язання рівнянь будь-якого степеня, але як пізніше довід

Абель, такої формули не існує (Теорема Абеля—Руффіні). Алан Тюрінг, Альфред Тарський, Курт Гедель, Юрій Матіясевич та інші математики довели багато алгоритмічно нерозв'язних проблем.

Що таке Рівнодоступна адресна машина?

Random-access stored-program machine (RASP, машина зі збереженою програмою та довільним доступом до реєстрів, або рівнодоступна адресна машина) — це абстрактна модель обчислювальної машини, яка використовується для теоретичних досліджень в області обчислювальної теорії.

Основні характеристики RASP включають:

1. Зберігання програми: RASP може зберігати програми, які визначають послідовність операцій для виконання.
2. Випадковий доступ до пам'яті: RASP може звертатися до будь-якої комірки пам'яті безпосередньо, використовуючи адресу цієї комірки.
3. Універсальність: RASP може виконувати будь-який обчислювальний алгоритм, який може бути описаний в термінах послідовних інструкцій, які визначаються в програмі.

RASP використовується для розв'язання різних теоретичних завдань, пов'язаних з обчисленнями, і служить базовою моделлю для аналізу обчислювальної складності, теорії алгоритмів та інших аспектів теорії обчислень. Вона допомагає дослідникам розуміти обчислювальну потужність і обмеження обчислювальних систем та алгоритмів.

Random-Access Stored-Program Machine (RASP) базується на архітектурі, яка подібна до архітектури фон Неймана. Це включає в себе концепцію зберігання як даних, так і програм у пам'яті, де вони представлені у вигляді послідовностей бітів. Основні риси архітектури фон Неймана, які спільні з RASP, включають:

1. Зберігання програми: Програма та дані зберігаються в одній пам'яті, що дозволяє програмі визначати та модифікувати свої власні інструкції.
2. Виконання інструкцій: Програми виконуються послідовно, кожна інструкція обробляється процесором в порядку їх збереження в пам'яті.
3. Загальний доступ до пам'яті: Програма може звертатися до будь-якої комірки пам'яті за її адресою, яка прописана в коді.
4. Універсальність: Обчислювальний пристрій може виконувати будь-які алгоритми, які можна виразити послідовністю інструкцій, включених в програму.
5. Операції зчитування/запису та арифметичні/логічні операції: Обчислювальний пристрій може виконувати операції зчитування/запису даних, арифметичні операції (наприклад, додавання, віднімання), а також логічні операції.

Ці концепції архітектури фон Неймана і RASP надають теоретичну основу для багатьох моделей обчислювальних машин і допомагають в розумінні та аналізі обчислювальних систем та алгоритмів.

Random-Access Stored-Program Machine (RASP) може виконувати різні базові обчислювальні операції, подібні до операцій, які виконує звичайний комп'ютер. Основні операції, які RASP може виконувати, включають:

1. Зчитування з пам'яті: RASP може зчитувати дані з пам'яті, вказавши адресу пам'яті, з якої потрібно прочитати дані.
2. Запис до пам'яті: RASP може записувати дані в пам'ять, вказавши адресу пам'яті та значення, яке потрібно записати.
3. Арифметичні операції: RASP може виконувати операції додавання, віднімання, множення та ділення над числами, які зберігаються в пам'яті.
4. Логічні операції: RASP може виконувати операції порівняння, логічне I (AND), логічне АБО (OR) та інші операції над бітовими даними.
5. Умовні операції: RASP може виконувати розгалуження, перевіряючи умови та виконуючи інструкції в залежності від результату умови.
6. Пересування даних: RASP може переміщати дані з однієї частини пам'яті в іншу або між реєстрами.
7. Виклик та завершення підпрограм: RASP може викликати підпрограми для виконання певних завдань і повернутися до основної програми після завершення підпрограми.
8. Завантаження та збереження програми: RASP може зчитувати програмні інструкції з пам'яті та виконувати їх послідовно.

Ці операції дозволяють RASP виконувати різні обчислювальні завдання та розв'язувати різні обчислювальні проблеми, використовуючи програми, що визначають послідовність операцій.

Архітектура RASP (Random-Access Stored-Program Machine) — це абстрактна архітектура, яка надає теоретичний фреймворк для дослідження обчислювальних процесів і алгоритмів. Вона містить наступні ключові характеристики:

1. Пам'ять: RASP має пам'ять, у якій зберігаються дані та програма. Ця пам'ять розділена на комірки, і кожна комірка має свою унікальну адресу, що дозволяє звертатися до даних за їхніми адресами.
2. Регістри: RASP може мати деяку кількість реєстрів, які використовуються для зберігання проміжних результатів обчислень і адреси пам'яті.
3. Арифметичні та логічні операції: RASP підтримує базові операції, такі як додавання, віднімання, множення, ділення, порівняння та логічні операції, які виконуються над даними в пам'яті або в реєстрах.
4. Виконання програми: RASP читає інструкції з пам'яті та виконує їх послідовно. Інструкції включають в себе операції над даними, умовні оператори та інші керуючі конструкції.
5. Умовні переходи: RASP може виконувати розгалуження в програмі, перевіряючи умови та визначаючи, яку гілку програми слід виконати.
6. Виклик підпрограм: RASP може включати можливість виклику і виконання підпрограм, що розширяє можливості виконання різних завдань.

Архітектура RASP є теоретичним інструментом для дослідження обчислювальних процесів і визначення обчислювальної потужності. Вона надає базовий фреймворк для розуміння обчислювальних алгоритмів і їхньої виконавчої моделі. Реальні комп'ютери мають свої власні

конкретні архітектури, такі як архітектура x86 для багатьох комп'ютерів, але RASP використовується для абстрактного аналізу і теоретичних досліджень.

RASP (Random-access stored-program machine - машина зі збереженою програмою та довільним доступом).

1. RASP має пам'ять, яка поділена на клітинки (реєстри), які мають власні номери.
2. RASP має "блок управління пам'яттю", який може зберігати, перезберігати, читувати дані з реєстра за номером його позначки.

Регістр 0 зарезервований і називається "суматор", оскільки він використовується за замовчуванням для деяких команд.

3. RASP має "блок керування програмою", який може читати нумерований список рядків, які є командами, і виконувати ці команди.
4. RASP має лічильник команд, які зберігають номер поточної команди, за замовчуванням він починається з 1.

"Програми — це конкретні формулювання абстрактних алгоритмів, заснованих на конкретному представленні та структурах даних" ("Алгоритми + Структури даних = Програми", Ніклаус Вірт, 1976).

В нашому випадку, програма — це список команд, кожна з яких має числове представлення.

Програма зберігається в частині пам'яті та може бути змінена "блоком управління пам'яттю", тоді "блок керування програмою" може прочитати оновлену команду.

Початкові дані для машини записують безпосередньо в реєстри, а результати роботи читаються безпосередньо з реєстрів (ми не визначаємо спеціальний інтерфейс для введення та виведення даних).

5. RASP має кнопку запуску, яка запускає "блок керування програмою" з першого рядка програми і після виконання лічильник команд збільшується на одиницю і виконує наступну команду в списку.

Команди RASP (Дуже проста мова асемблера, яка в пам'яті є як машинний код):

$r(i)$ - зміст реестру i .

1. Load a, завантажте a до реєстру 0;
2. Store a, збережіть $r(0)$, до реєстру a;
3. Add a, виконайте $r(0) + a$ і збережіть до реєстру 0;
4. Sub a, виконайте $r(0) - a$ і збережіть до реєстру 0;
5. Mult a, виконайте $r(0) * a$ і збережіть до реєстру 0;
6. Div a, виконайте $r(0) / a$ і збережіть до реєстру 0;
7. Jump a, лічильник команд до a.
8. JZERO a, якщо $r(0) = 0$, лічильник команд встановлюється до a, якщо ні до наступної команди.
9. Halt, припинити роботу.

Усі ці команди мають числові представлення, які зберігаються в пам'яті та можуть бути динамічно (в роботі) змінені.

Що таке архітектура фон Неймана?

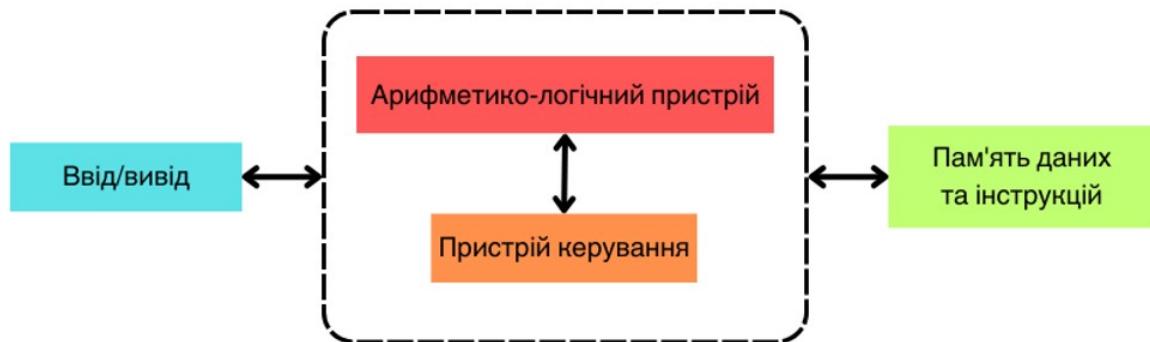
Архітектура фон Неймана (також відома як Прінstonська архітектура) — це стандартна архітектура комп'ютера, яка була розроблена та описана у 1945 році математиком та фізиком Джоном фон Нейманом. Ця архітектура визначає базову структуру сучасних цифрових комп'ютерів і є відомою своєю простотою та універсальністю.

Основні характеристики архітектури фон Неймана включають такі елементи:

1. Центральний процесор (ЦП): Всі обчислення виконуються центральним процесором. Він взаємодіє з іншими частинами комп'ютера і виконує інструкції, які зберігаються у пам'яті.
2. Пам'ять: Програми та дані зберігаються в спільній пам'яті. Це означає, що як програми, так і дані представлені у вигляді послідовності байтів, і до них можна звертатися однаковим чином.
3. Керування: Програми зберігаються у пам'яті та виконуються послідовно, інструкція за інструкцією. Центральний процесор читає інструкції з пам'яті та виконує їх.
4. Програмно-керований: Комп'ютери на основі архітектури фон Неймана використовують програми для керування своєю діяльністю. Це означає, що ви можете змінювати функціональність комп'ютера, змінюючи програми, які виконуються на ньому.
5. Загальний доступ до пам'яті: Центральний процесор може читати та записувати дані в будь-яку частину пам'яті, а не обмежувати доступ до пам'яті окремим регіонам.

Ця архітектура стала основою для більшості сучасних комп'ютерів і є стандартом у світі обчислювальної техніки. Архітектура фон Неймана дозволяє комп'ютерам виконувати різноманітні завдання, включаючи обчислення, зберігання та обробку даних, і вона залишається основною архітектурою для багатьох застосувань.

Архітектура фон Неймана (Прінstonська архітектура) — комп'ютерна модель, в якій застосований принцип спільного зберігання інструкцій і даних у пам'яті комп'ютера.



Що таке Гарвардська архітектура?

Гарвардська архітектура — це одна з двох основних архітектур, що використовуються в архітектурі комп'ютерів, іншою є архітектура фон Неймана (von Neumann architecture). Гарвардська архітектура відрізняється від архітектури фон Неймана тим, яким чином обробляються та зберігаються дані та команди обчислень.

Основні риси Гарвардської архітектури:

Розділення пам'яті: У Гарвардській архітектурі використовуються окремі пам'яті для зберігання інструкцій (пам'ять команд) і даних (пам'ять даних). Це означає, що програмні інструкції і дані зберігаються в різних фізичних просторах пам'яті.

Різні шини: Гарвардська архітектура використовує окремі шини для передачі даних і команд між процесором і пам'яттю команд (шлях команд) і пам'яттю даних (шлях даних). Це дозволяє одночасно виконувати читання і записи в пам'ять.

Швидший доступ до команд: Оскільки пам'ять команд і пам'ять даних розділені, то процесор може одночасно звертатися до пам'яті команд і пам'яті даних, що дозволяє підвищити швидкодію програм.

Інструкції та дані в різних форматах: У Гарвардській архітектурі інструкції і дані можуть бути представлені в різних форматах та кодуваннях.

Ця архітектура особливо підходить для застосувань, де важлива швидкодія та одночасний доступ до даних і команд. Вона зазвичай використовується в високопродуктивних обчислювальних системах, таких як мікроконтролери, процесори сигналів та суперкомп'ютери.

У контрасті з цим, архітектура фон Неймана використовує одну спільну пам'ять для команд і даних, і вона більш поширенна в загальних цілях комп'ютерів.

В Гарвардській архітектурі пам'ять розділена на два окремі блоки: пам'ять команд і пам'ять даних. Тобто програмні інструкції (команди) зберігаються в одній частині пам'яті, а дані, з якими працює програма, зберігаються в іншій частині пам'яті.

Для доступу до цих двох частин пам'яті використовуються окремі шляхи (або шини). Одна шина призначена для передачі команд (інструкцій) з пам'яті команд до процесора, а інша - для передачі даних з пам'яті даних до процесора.

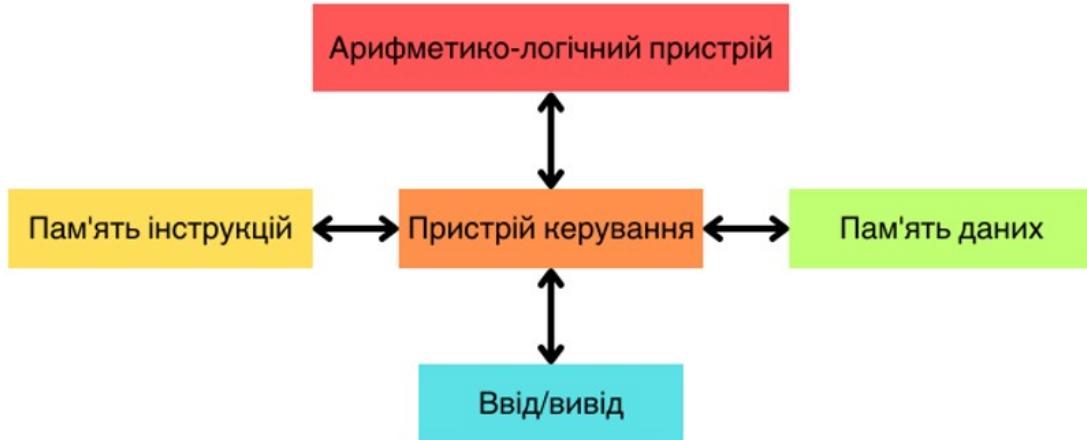
Система керування процесором і система керування пам'яттю роблять так, щоб процесор правильно взаємодіяв із цими двома частинами пам'яті. Вони визначають, які команди виконувати і які дані читувати чи записувати, так щоб програма працювала коректно.

В результаті такої організації, команди та дані можуть бути доступні процесору одночасно і без конфліктів. Це сприяє покращенню швидкодії обчислень в Гарвардській архітектурі.

У Гарвардській архітектурі, програми і дані зберігаються в окремих частинах пам'яті, і ця архітектурна особливість може вплинути на можливість зміни програми під час її виконання. Зазвичай програми зберігаються в пам'яті команд, і доступ до неї для зміни може бути обмежений.

Принцип розділення каналу даних та інструкцій відомий як Гарвардська архітектура. Гарвардська архітектура була розроблена Говардом Ейкеном наприкінці 1930-х років у Гарвардському університеті.

Гарвардська архітектура — це архітектура комп'ютера, характерними ознаками якої є: 1. Сховище інструкцій і сховище даних є окремими фізичними пристроями; 2. канал команд і канал даних також фізично розділені.



Типові операції (додавання та множення) вимагають кількох дій від будь-якого обчислювального пристрою:

1. отримання двох операндів;
2. вибір інструкції та її виконання;
3. збереження результату.

Ідея Говарда Ейкена полягала в тому, щоб фізично розділити командний рядок і рядки даних. Перший комп'ютер Ейкена, Марк I, використовував перфоровану стрічку для зберігання інструкцій та електромеханічні регистри для маніпулювання даними. Це дало змогу одночасно надсилати й обробляти команди та дані, що значно підвищило загальну продуктивність комп'ютера.

У класичній архітектурі фон Неймана процесор у будь-який момент часу може або прочитати інструкцію, або прочитати/записати одиницю даних з/в пам'ять. Обидві дії не можуть виконуватися одночасно, оскільки інструкції та дані використовують один і той самий канал (шину). У комп'ютері з Гарвардською архітектурою процесор може читати наступну інструкцію та працювати з пам'яттю даних одночасно і без використання кеш-пам'яті. Таким чином, комп'ютер з Гарвардською архітектурою, з певною складністю схеми, швидше комп'ютера з архітектурою фон Неймана, оскільки потоки інструкцій і даних розташовані на окремих, фізично не пов'язаних між собою апаратних каналах. Виходячи з фізичного розділення шин команд і даних, розміри біт цих шин можуть відрізнятися і не можуть фізично перетинатися.

Random-Access Machine (RAM), або машина з випадковим доступом, є абстрактною моделлю обчислення, яка використовується в теорії обчислювальної складності і аналізі алгоритмів. Ця модель допомагає дослідникам аналізувати та порівнювати алгоритми за їх продуктивністю та ресурсами, не звертаючи увагу на конкретні архітектури комп'ютерів або мови програмування.

Основні характеристики Random-Access Machine включають:

1. Зберігання даних: RAM має незмінну кількість пам'яті, поділену на окремі комірки або регистри. Кожен регистр може зберігати ціле число.

2. Інструкції: RAM має набір базових інструкцій, які можуть виконуватися, такі як операції додавання, віднімання, множення, ділення, порівняння і переміщення даних між реєстрами.
3. Виконання послідовних команд: Інструкції виконуються послідовно, одна за одною, і виконання може переходити в іншу команду залежно від умовних операторів.
4. Випадковий доступ до даних: RAM може безпосередньо звертатися до будь-якого реєстру за допомогою його адреси. Це означає, що доступ до даних не залежить від їх розташування в пам'яті.

RAM використовується для теоретичного аналізу алгоритмів та розрахунку їх складності, оскільки вона надає спрощену модель обчислювальної системи, яка не враховує деталей конкретних архітектур. Ця модель допомагає дослідникам робити загальні висновки про швидкість та ефективність алгоритмів без необхідності враховувати деталі конструкції реальних комп'ютерів.

RAM використовує Гарвардську архітектуру, оскільки в ній програма і дані розділені. Це відрізняється від архітектури фон Ноймана, де програма та дані розміщені в одній пам'яті.

У загальному, RAM — це модель обчислювальної машини, яка використовується для аналізу алгоритмів та обчислювальних проблем. Вона використовує Гарвардську архітектуру, де програма і дані зберігаються окремо, що надає моделі деяку гнучкість у виконанні обчислень. RAM - це важливий інструмент для теоретичного дослідження алгоритмів та обчислювальних завдань у комп'ютерній науці.

Що таке архітектура програмного забезпечення?

В загальному, в програмній інженерії під архітектурою розуміють загальний дизайн системи, тобто дизайн зв'язків між модулями системи та їх взаємодією.

Архітектура додатка — це його основна структура (*модулі та зв'язки між ними*) і правила взаємодії між її частинами (*модулями, компонентами*).

Визначення архітектури програмного забезпечення є у SWEBOK (IEEE), також в книзі "Чиста архітектура" Роберта Мартіна, або Роджера Пресмана "Програмна інженерія: підхід практика".

Існують архітектури:

- інформації (Site map, ER model),
 - програми (Model-View-Controller, Monolithic, Microservices architecture),
 - операційної системи (Microkernel),
 - процесора (RISC),
 - комп'ютера (von Neumann architecture),
 - мережі (Client-Server, Peer-to-Peer).
- та інші.

Що таке булева алгебра?

Булева алгебра — це математична галузь, яка вивчає логічні операції та вирази на основі двох можливих значень: правда (1) та хибність (0). Ця алгебра названа на честь англійського математика та логіка Джорджа Буля, який створив її в середині 19 століття. Булева алгебра є основою для роботи з логічними виразами та булевими змінними в інформатиці та електроніці.

Основні логічні операції в булевій алгебрі включають:

1. Логічне "І" (AND): Позначається звичайно як " \wedge " або " $&$ ". Результат операції "І" дорівнює 1, лише якщо обидва операнди дорівнюють 1, в інших випадках результат дорівнює 0.
2. Логічне "АБО" (OR): Позначається звичайно як " \vee " або " $|$ ". Результат операції "АБО" дорівнює 1, якщо хоча б один із операндів дорівнює 1.
3. Логічне "НЕ" (NOT): Позначається як " \neg " або " $!$ ". Результат операції "НЕ" - це обернення значення операнду, тобто 0 стає 1, а 1 стає 0.

Булева алгебра має широке застосування в логіці, математиці, інформатиці, електроніці, програмуванні, автоматизації та багатьох інших галузях. Вона є важливою для створення та аналізу логічних виразів і алгоритмів, що використовуються в сучасних технологіях і обчисленнях.

Система Буля доступно описана в книзі “Код” Чарльза Петцольда для програмістів початківців, а саме:

Коти можуть бути самцями та самками. Для зручності позначатимемо кота-самця буквою M, а самку — F. Букви M і F символізують множини (класи) котів з певними властивостями. Щоб позначити множину усіх котів чоловічої статі, просто скажіть M. Букви також можуть вказувати на колір кота. Наприклад, буква G може відповісти рудим котам, літера B — чорним, W — білим. У множині O можна визначити котів усіх інших кольорів, тобто, які не входять до множин G, B і W. У звичайній (числовій) алгебрі оператори + і * символізують додавання і множення. Знаки + і * в булевій алгебрі мають дещо інше значення. Символ + в булевій алгебрі позначає об’єднання двох множин, тобто множину, що складається з усіх елементів першої множини та всіх елементів другої множини. Наприклад, набір B + W включає всіх чорних і білих котів. Символ * в булевій алгебрі відповідає перетину двох множин, тобто означає множину, яка містить лише елементи, що входять як до першої, так і до другої множини. Наприклад, набір M * G складається з самок рудих кішок. Як і множення у звичайній алгебрі, перетин двох множин можна записати як M * G або просто MG. Щоб уникнути плутанини між звичайними та булевими алгебраїчними операціями, замість + і * об’єднання та перетину іноді позначаються \cup та \cap . У булевій алгебрі також виконуються закони асоціативності, комутативності та дистрибутивності. Символ 1 в булевій алгебрі означає не число, а універсальну множину, тобто множину всього, про що ми говоримо, наприклад, у нашому випадку це множина всіх без винятку котів. Таким чином:

$$M + F = 1.$$

Іншими словами, коти будь-якої статі взагалі потрапляють в об’єднання котів і кішок. У поєднанні зі знаком мінус 1 означає все, крім того, що віднімається.

Отже, набір: 1 - M містить усіх котів, крім котів-самців.

Зрозуміло, що в цей набір входять самки: 1 - M = F.

Останній символ, який нам потрібен, — 0. У булевій алгебрі це означає порожню множину, тобто множину, яка не містить жодного елемента.

Порожня множина є результатом перетину множин, що не перекриваються, наприклад, множин кішок і самців: F * M = 0.

Оскільки множина F містить усіх кішок, а множина (1 - F) містить усіх кішок протилежної статі, об’єднання цих двох множин дорівнює 1: $F + (1 - F) = 1$, а їх перетин дорівнює 0: $F * (1 - F) = 0$.

Цей вираз відіграв важливу роль в історії алгебри логіки, тому має свою назву — закон протиріччя. Він говорить, що ніщо не може бути собою і своєю противідністю одночасно.

Але в наступному виразі різниця між булевою алгеброю і звичайною алгеброю вже проявляється: $F * F = F$.

У булевій алгебрі цей вираз сповнений змісту: перетин множини кішок із самими собою дорівнює тій самій множині. Якщо замість множини замінити звичайне число, сенс виразу буде втрачено.

Ще один вираз, який виглядає дивним з точки зору звичайної алгебри: $F + F = F$, асоціація множини кішок з самими собою дорівнює тій самій множині.

У булевій алгебрі запропоновано математичний спосіб розв’язання аристотелівського силогізму.

Згадаємо його перші два рядки: Усі люди смертні; греки — люди. Позначимо літерою H множину всіх людей, буквою D — множину смертних істот, а буквою G — множину греків. Що означає вислів “Усі люди смертні”? Це означає, що перетин усіх людей з усіма смертними є рівним для всіх людей: $H * D = H$. Але вираз $H * D = D$ буде невірним, оскільки до багатьох смертних, крім людей, належать кішки, собаки, та багато іншого. Вислів “греки — люди” можна передати так: перетин множини греків з множиною людей дорівнює множині греків $G * H = G$.

Щоб перевірити, чи правильно написано логічний вираз, давайте на деякий час забудемо про поняття об’єднання та перетину та замість цього використаємо оператори АБО та І. Крім того, замість одиниці зі знаком мінус можна використовувати оператор НЕ.

Так:

замість оператора об’єднання + пишемо АБО;

замість оператора перетину * пишемо І;

замість 1 - (все крім х) пишемо НЕ.

У сучасних текстах замість НЕ, І, АБО, вживаються знаки \neg , \wedge , \vee .

Отже, ми поступово перейшли до предикатної форми булевої алгебри. В ній букви вже не позначають множини, а предикати, які можуть бути істинними (1) або хибними (0). 1 означає істину: так, об'єкт відповідає цій умові. 0 означає хибний: ні, об'єкт не відповідає заданій умові.

Скажімо, нам потрібен білий кіт.

Ми визначаємо три предикати:

В: Це кіт.

Б: Це самець.

С: Він білий.

Формула $A \wedge B \wedge C$ вкаже потрібного нам кота. Якщо всі предикати для потрібного нам об'єкта правильні, тобто $1 \wedge 1 \wedge 1$, то об'єкт відповідає всім умовам.

При множенні чисел за правилами булевої алгебри можливі наступні результати:

$$0 * 0 = 0;$$

$$0 * 1 = 0;$$

$$1 * 0 = 0;$$

$$1 * 1 = 1;$$

Іншими словами, результат дорівнює 1, тільки якщо лівий І правий операнд дорівнюють 1. У логічному додаванні можливі наступні результати:

$$0 + 0 = 0;$$

$$0 + 1 = 1;$$

$$1 + 0 = 1;$$

$$1 + 1 = 1;$$

Результат дорівнює 1, якщо лівий АБО правий операнд дорівнюють 1. І ця операція мало чим відрізняється від звичайного додавання, за винятком того, що сума двох одиниць також дорівнює 1.

Що таке двійкова система числення?

Двійкова система числення — це позиційна система числення, подібна до десяткової системи (десяткова система використовує 10 цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), але вона використовує всього дві цифри: 0 і 1.

Основними характеристиками двійкової системи є:

Позиційність: Значенняожної цифри у двійковому числі залежить від її позиції в числі. Наприклад, в двійковому числі "1010", перше "1" представляє 1, друге "1" представляє 2, третє "1" представляє 4, і останнє "0" представляє 0. Отже, це число у двійковій системі означає $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 10$ в десятковій системі.

Використання ступенів двійки: Кожна позиція в двійковому числі представляє ступінь числа 2. Починаючи справа, перша позиція представляє 2^0 (1), друга - 2^1 (2), третя - 2^2 (4), четверта - 2^3 (8), і так далі.

Для перетворення числа з десяткової системи у двійкову, ви ділите число на 2 і запам'ятуєте залишок. Потім продовжуєте ділити результат на 2, доки не отримаєте 0. Потім записуєте всі залишки знизу вгору, і це буде двійкове представлення числа.

Наприклад, щоб перетворити 13 у двійкову систему, ви можете виконати такі дії:

$$\begin{aligned}13 / 2 &= 6 \text{ (залишок: 1)} \\6 / 2 &= 3 \text{ (залишок: 0)} \\3 / 2 &= 1 \text{ (залишок: 1)} \\1 / 2 &= 0 \text{ (залишок: 1)}\end{aligned}$$

Потім записуєте залишки від низу до верху, і отримуєте "1101" як двійкове представлення числа 13.

Двійкова система числення є дуже важливою для комп'ютерів, оскільки всі дані та операції в комп'ютерах виконуються з використанням електричних сигналів, які можна легко реалізувати за допомогою двійкових значень (0 і 1).

У двійковій системі числення всі натуральні числа представлені лише двома знаками 0 і 1.

000		0
001	•	1
010	••	2
011	•••	3
100	••••	4
101	•••••	5

$$101 = (1 * \text{pow}(2, 0)) + (0 * \text{pow}(2, 1)) + (1 * \text{pow}(2, 2)).$$

Двійковий код використовував Френсіс Бекон для шифрування повідомлень. Кожній літері алфавіту він приписував двійковий код, який складався лише з двох символів. Лейбніц, вивчаючи символи гуа, тобто триграми з “Книги змін”, підійшов до їх тлумачення математично і побачив, що вони являють собою двійковий код, таким чином Лейбніц став одним із перших математиків, які серйозно почали вивчати двійкову систему числення, що є основою сучасної обчислювальної техніки.

Подання цілих чисел у двійковому вигляді

$$7 = 0111,$$

$$6 = 0110,$$

$$5 = 0101,$$

$$4 = 0100,$$

$$3 = 0011,$$

$$2 = 0010,$$

$$1 = 0001,$$

$$0 = 0000,$$

$$-1 = 1111,$$

$$-2 = 1110,$$

$$-3 = 1101,$$

$$-4 = 1100,$$

$$-5 = 1011,$$

$-6 = 1010,$

$-7 = 1001,$

$-8 = 1000,$

$3 + 2 = 0011 + 0010,$

$0011 + 0010 = 0101,$

$0101 = 5,$

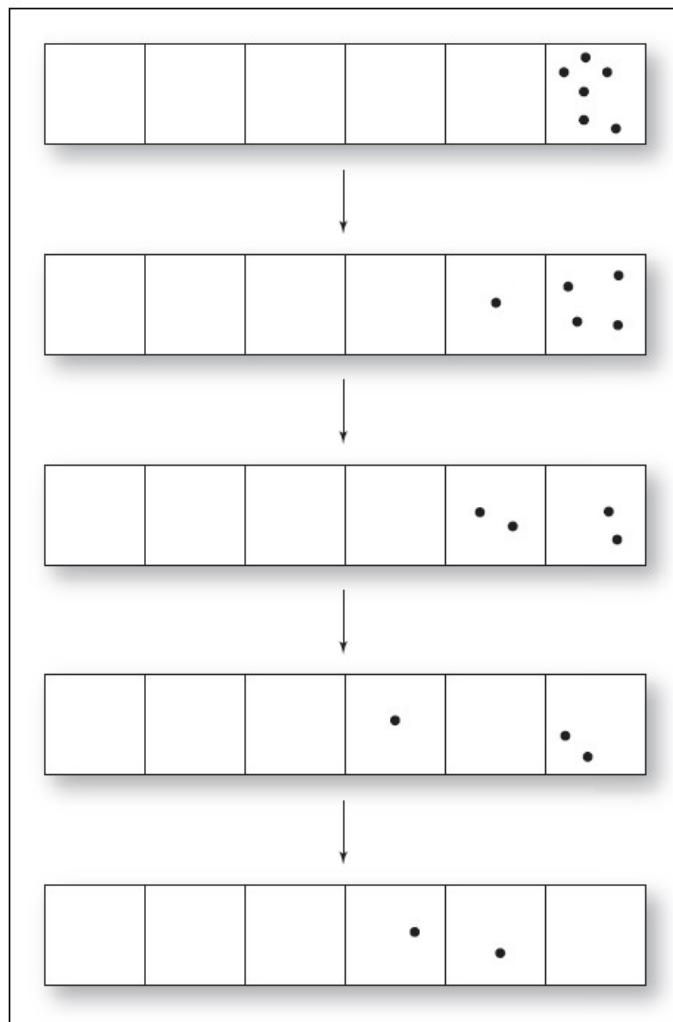
$-3 + -2 = 1101 + 1110,$

$1101 + 1110 = 1011,$

$1011 = -5.$

Числовий автомат “1←2”

Числовий автомат “1←2” (два до одного) складається з набору комірок (в один ряд). В першу праву комірку кладуть певну кількість монет. Автомат має просте правило: “Якщо в комірці є пара монет, забрати цю пару монет, а в сусідню ліву комірку поставити одну монету”. Тобто, якщо в першій правій комірці чотири монети, вони будуть усунуті, а в наступній комірці з'являться дві монети, які також будуть усунуті й в наступній комірці буде поставлена одна монета. Цей автомат переводить число у двійковий формат. Якщо в першу комірку поставити шість монет, ми отримаємо розподіл монет 110, тобто число 6 у двійковій формі.



Що таке стандарт IEEE 754?

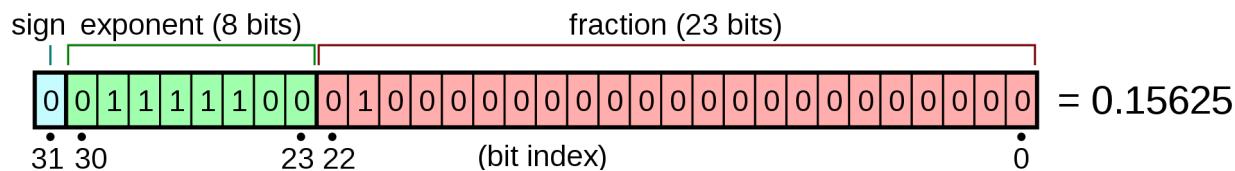
IEEE 754 — це стандарт для представлення чисел з рухомою (плаваючою) комою у бінарному форматі в обчислювальних системах. Цей стандарт був розроблений Інститутом інженерів електротехніки та електроніки (IEEE) і прийнятий як IEEE 754-1985. Він був оновлений пізніше у 2008 році і отримав назву IEEE 754-2008.

Стандарт IEEE 754 визначає формати збереження чисел з плаваючою комою (як дійсні числа) в пам'яті обчислювальних пристрій та правила для виконання операцій з числами з плаваючою комою, такими як додавання, віднімання, множення та ділення. Він також визначає, як перетворювати ці числа між бінарним та десятковим форматами.

Стандарт IEEE 754 надає можливість виконувати арифметичні операції на числах з плаваючою комою з високою точністю і надійністю, що є важливим в багатьох обчислювальних додатках, таких як наукові обчислення, фінансовий облік, графіка та інші.

Стандарт IEEE 754 — це технічний стандарт для арифметики з плаваючою комою, створений у 1985 році Інститутом інженерів з електрики та електроніки (IEEE).

У випадку 32-бітового представлення числа з плаваючою комою ми можемо описати правила цього стандарту так: Знак зберігається в біті 32. Експонента може бути обчислена з бітів 24-31 шляхом віднімання 127. Мантиса (також відома як фракція або сигніфікат) зберігається в бітах 1-23. Попереду розміщується невидимий провідний біт (тобто він фактично не зберігається) зі значенням 1,0, потім біт 23 має значення 1/2, біт 22 має значення 1/4 тощо. В результаті мантиса має значення між 1,0 і 2. Якщо експонента досягає -127 (двійковий 00000000), початкова 1 більше не використовується для включення поступового переповнення.



01111100 = 124; (експонента)

124 - 127 = -3; (Експонента може бути обчислена з бітів 24-31 шляхом віднімання 127)

$\frac{1}{4} = 0,25$. (мантиса)

$(1 + 0,25) * 2^{-3} = 0,15625$.

(Одиниця додається, щоб мантиса з усіма нульовими бітами була рівною 1)

NaN (Not-a-Number) — одне з особливих значень числа з рухомою комою. Відповідно до стандарту IEEE 754, такий стан задається через встановлення показника ступеня в зарезервоване значення. Використовується у багатьох математичних бібліотеках і математичних співпроцесорах. Цей стан може виникнути в різних випадках, наприклад, коли попередня математична операція завершилася з невизначенним результатом, або якщо в комірку пам'яті потрапило число, що не задовільняє умовам. NaN не дорівнює жодному іншому значенню (навіть самому собі); відповідно, найпростіший метод перевірки результату на NaN — це порівняння отриманої величини з самою собою.

IEEE 754 NaN кодуються з полем експоненти, заповненим одиницями (як нескінчені значення), і деяким відмінним від нуля числом у значущому полі (щоб відрізняти їх від нескінченних значень); це

дозволяє визначати кілька окремих значень NaN, залежно від того, які біти встановлено в полі сигніфікат, а також від значення біта провідного знака (але програми не зобов'язані надавати чітку семантику для цих різних значень NaN).

Що таке шістнадцяткова система числення?

Шістнадцяткова система числення, також відома як система числення за основою 16, є численням, яке використовує 16 символів для представлення чисел. Ця система включає в себе десять чисел від 0 до 9, а також шість літер латинського алфавіту від A до F, які використовуються для представлення чисел від 10 до 15. Ось перелік цих символів у шістнадцятковій системі та їх значень:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

У шістнадцятковій системі кожна цифра чи літера відповідає певному значенню, як у десятковій системі (за основою 10). Наприклад:

- 1 в шістнадцятковій системі відповідає числу 1 в десятковій системі.
- A в шістнадцятковій системі відповідає числу 10 в десятковій системі.
- F в шістнадцятковій системі відповідає числу 15 в десятковій системі.

Шістнадцяткова система числення часто використовується в інформатиці і програмуванні, оскільки вона зручна для представлення великої кількості даних, таких як кольори, адреси пам'яті та інші великі числа.

255 (десятикове) = FF (шістнадцяткове).

Що таке десяткова система числення?

Десяткова позиційна система

Цифри (знаки): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

$$3827 = 3000 + 800 + 20 + 7;$$

$$3827 = (3 * 10^3) + (8 * 10^2) + (2 * 10) + 7;$$

$$10 = 5 + 5;$$

$$20 = 10 + 10;$$

$$50 = 20 + 20 + 10;$$

$$100 = 50 + 50;$$

$$200 = 100 + 100;$$

$$500 = 200 + 200 + 100;$$

$$1000 = 500 + 500;$$

$$100 = 10^2;$$

$$1000 = 10^3;$$

Мухаммад аль-Хорезмі (780 – 850) — написав "Книгу про індійські числа". Арабський текст було втрачено, але його латинський переклад XII століття зберігся. Трактат розповідає про числа, які використовували індійці (індуси), і визначає дії над ними. Індійці розробили десяткову позиційну систему числення.

Леонардо Пізанський (1170 - 1250) — італійський математик, популяризатор десяткової системи числення в Європі. Фібоначчі написав кілька творів, найвідомішим з яких є "Книга абака", де описані цифри 1,2,3,4,5,6,7,8,9,0.

Майже всі цифри у нього записані в сучасній формі, за винятком чисел 4, 5, які дещо відрізнялися від сучасного запису.

Арабський вчений початку 15 століття Аль-Каші був одним з перших, хто використав десяткові дроби, зокрема, він їх описав у своїй книзі "Ключ арифметики". Він розділяв розряди (місця) пробілом, кожен розряд міг містити будь-яке натуральне число.

Наприклад, 33. 55 66 означатиме 33 одиниці 55 десятих 66 сотих. Для використання цього способу запису необхідно позначити опорний розряд (наприклад знаком ".") , відносно якого будемо визначати початок дробових розрядів. Бельгійський математик Симон Стевін запропонував спосіб запису десяткових дробів у своїй книзі "Десята", яка була видана в 1585 році (на нідерландській і французькій мові).

Запис десяткових дробів у Стевіна трішки відрізняється від сучасного запису.

Сучасний запис десяткових дробів знаходимо в книзі "Опис дивовижної таблиці логарифмів" (видана в 1614) шотландця Джона Непера, а саме: $9,33 = 9 + (33/100)$.

Раціональні числа в десятковій формі

$$0,x = x:10;$$

$$0,xy = x:100;$$

$$0,xyz = xyz:1000;$$

Приклад

$$0,5 = 5:10;$$

$$0,25 = 25:100;$$

$$0,xxxxxxxx... = x:9;$$

Знак "...” означає “до нескінченності”.

Приклад

$$1 / 3 = 0,33333... = 3:9;$$

$$0,xuyxuhuxuhuxy.... = xy:99;$$

Приклад

$$6 / 4 = 0,5454545454... = 54:99;$$

Дві валізи та кавун можуть скласти три предмети, які ви берете з собою в дорогу, але дві валізи та пів кавуна не становлять двох з половиною предметів, тому що використання дробових чисел передбачає однорідність.

Що таке експоненційний запис?

Експоненційний запис (наукова нотація) — це спосіб вираження чисел, які є занадто великими або занадто малими, щоб їх зручно було записати в десятковій формі. Він широко використовується вченими, математиками та інженерами.

У експоненційному записі всі числа записуються у вигляді:

$$m * \text{pow}(10, e) = m * 10^e,$$

де показник e є цілим числом, а коефіцієнт m будь-яке дійсне число.

Ціле число e називається порядком величини, а дійсне число m — мантисою, або сигніфікатом. Якщо число від'ємне, то перед m ставиться знак мінус (як у звичайній десятковій системі числення).

У нормалізованому записі експонента вибирається так, щоб абсолютне значення (модуль) значущої m було принаймні 1, але менше ніж 10.

Наприклад: $1,6e12 = 1,6 * \text{pow}(10, 12)$;

Наприклад: $1,6e-12 = 1,6 * \text{pow}(10, -12)$;

Експоненційний запис (наукова нотація) використовується в стандарті IEEE 754 та в багатьох мовах програмування.

Що таке ASCII та Unicode?

ASCII та Unicode — це стандарти кодування символів, які використовуються для представлення тексту в комп'ютерах. Ось короткий огляд кожного з цих стандартів:

1. ASCII:

- ASCII (American Standard Code for Information Interchange) — це стандарт кодування символів для електронних комунікацій. Коди ASCII представляють текст у комп'ютерах, телекомунікаційному обладнанні та інших пристроях. Через технічні обмеження комп'ютерних систем на той час, коли він був винайдений, ASCII має лише 128 кодових точок, з яких лише 95 є друкованими символами, що сильно обмежує його сферу застосування.

2. Unicode:

- Unicode - це стандарт, який розроблений для уніфікації представлення тексту у всьому світі. Він має за мету включити всі символи всіх писемностей у світі, що дає можливість представляти текст будь-якої мови.

- Unicode використовує більше одного байта для представлення символів. Найпоширеніше кодування - UTF-8 (використовує від 1 до 4 байтів для символу) та UTF-16 (використовує 2 байти для більшості символів).

- Однією з головних переваг Unicode є те, що він дозволяє використовувати символи будь-якої мови без необхідності перекодування тексту.

У більшості випадків, особливо в сучасних комп'ютерах і програмах, рекомендується використовувати Unicode, оскільки він забезпечує більшу універсальність і підтримку різних мов. Unicode використовується в багатьох операційних системах, програмах та на веб- сайтах для забезпечення правильного відображення тексту у всіх мовах світу.

На рівні процесора Unicode зазвичай опрацьовується у програмному забезпеченні (операційні системи та додатки) та не безпосередньо в самому апаратному процесорі. Процесори найчастіше операційні системи та програми взаємодіють із текстовими даними, які представлені у форматі Unicode. Тобто операційні системи та програми використовують Unicode для забезпечення підтримки тексту у різних мовах і писемностях.

Процесори (центральні процесори, CPU) виконують різні операції, але вони зазвичай не пряму взаємодіють з текстовими даними на рівні символів або кодових точок Unicode. Замість цього, операційні системи та програми обробляють дані в кодуванні Unicode, а процесори виконують обчислення та операції з цими даними, без усвідомлення внутрішньої структури Unicode.

Сам процесор відповідає за виконання інструкцій програм, операції з пам'яттю, арифметичні обчислення і т. д., але для операцій з текстовими даними, він спирається на програмне забезпечення, яке передає йому відповідні дані в потрібному кодуванні, такому як Unicode.

Що таке об'єм пам'яті?

Об'єм пам'яті стосується кількості інформації, яку може зберігати певний пристрій або система. Це може бути фізичний обсяг пам'яті у комп'ютерах, смартфонах, планшетах та інших електронних пристроях, або загальна місткість пам'яті, яку людина або організація може використовувати для зберігання даних.

“Інформація — це інформація, а не матерія чи енергія” (Норберт Вінер, “Кібернетика”, 5). Інформація це те що ми можемо прочитати, дізнатись аналізуючи матерію та феномени. На основі інформації ми приймаємо рішення.

Об'єм пам'яті може бути вимірюваний у байтах, кілобайтах (KB), мегабайтах (MB), гігабайтах (GB), терабайтах (TB) і так далі. Чим більший об'єм пам'яті, тим більше даних можна зберігати. Наприклад, комп'ютери зазвичай мають фізичну пам'ять у вигляді оперативної (RAM) та постійної (наприклад, жорсткий диск) пам'яті. Обсяг оперативної пам'яті впливає на продуктивність комп'ютера, оскільки в ній зберігаються дані, з якими працює поточно виконувана програма, а обсяг постійної пам'яті вказує на загальну кількість даних, які можна зберігати на пристрої.

Також об'єм пам'яті може відноситися до загальної ємності сховищ, таких як флеш-накопичувачі, SD-карти, хмарні сховища та інше, де користувачі зберігають свої файли і дані. Об'єм пам'яті є важливим параметром для визначення, скільки інформації можна зберігати на конкретному пристрої або в послугах зберігання даних.

Концепція біту стала основою для представлення та обробки інформації в бінарній системі числення, яка є основою для всіх сучасних обчислень та комп'ютерів. Бінарна система числення використовує два символи (0 і 1), і кожне число можна представити як комбінацію бітів. Біти використовуються для зберігання та обробки даних у цифровій електроніці, і вони є фундаментальними для роботи комп'ютерів та багатьох інших пристройів.

Біт може приймати одне з двох можливих значень: 0 або 1, і використовується для подання інформації в цифровій формі.

Американський математик Джон Тьюкі (1915-2000) відомий як автор двох комп'ютерних термінів — “софтвер” (програмне забезпечення) (1958) і “біт” (скорочення від binary digit) (1946).

“Якщо використовується основа 2, отримані одиниці можна назвати двійковими цифрами (binary digits), або, коротше, бітами, слово, запропоноване Дж. Тьюкі. Пристрій з двома стабільними положеннями, наприклад реле, може зберігати один біт інформації. N таких пристройів може зберігати N біт, оскільки загальна кількість можливих станів дорівнює 2^N і $\log_2(2^N) = N$ ” (Клод Шеннон, "Математична теорія зв'язку", 1948)

Інформаційна ентропія

Ентропія визначає невизначеність чи випадковість в інформації. Чим більше ентропія, тим більше невизначеності.

Формула Шенонна для ентропії (H) виглядає наступним чином:

$$H(X) = -\sum_{i=1}^n P(x_i) \cdot \log_2(P(x_i));$$

Дано дискретну випадкову величину X , яка приймає значення в алфавіті $\{x_i\}$.

У формулі логарифм дозволяє "зменшити вагу" високих ймовірностей, зробити їх внесок меншим, тим самим підсилюючи важливість менш ймовірних подій. Ймовірність визначена на відрізку $[0,1]$.

Нехай у нас є випадкова величина X , яка може приймати три можливі значення з ймовірностями:

$$P(X=1) = 0.4,$$

$$P(X=2) = 0.3,$$

$$P(X=3) = 0.3.$$

$$H(X) = -\sum_{i=1}^3 P(x_i) \cdot \log_2(P(x_i)) = -(0.4 * (-1.322) + 0.3 * (-1.737) + 0.3 * (-1.737)) \approx 1.57;$$

Якщо у вас є випадкова величина з трьома можливими значеннями, то максимальне значення ентропії буде:

$$H_{\max} = \log_2(3) = 1.585;$$

Чим менше ймовірність події, тим більший внесок в несподіваність вносить її логарифм.

Вибір основи для логарифма різний для різних програм. Основа 2 дає одиницю бітів (або "шанон"), тоді як основа е дає "природні одиниці" nat, а основа 10 дає одиниці "dits", або "hartleys".

Гартлі (hartley, dit) — логарифмічна одиниця вимірювання інформації або ентропії на основі логарифмів за основою 10 та степенів 10, радше ніж степенів 2 та логарифмів за основою 2, які визначають біт, або шенон. Один гартлі є кількістю інформації такої події, ймовірність трапляння якої становить 1/10.

Що цікаво, складність по Колмогорову та ентропія пов'язані так, що у випадку максимальної ентропії (найвищої невизначеності), складність по Колмогорову буде максимальною. Інакше кажучи, якщо інформація дуже випадкова і не має внутрішнього порядку, то її важко стиснути або описати коротко. Таким чином, ентропія та складність по Колмогорову можуть слугувати взаємозамінними показниками у різних контекстах теорії інформації.

Що таке логічні вентилі та напівсуматор?

Логічні вентилі та напівсуматори є електронними пристроями, які використовуються в цифрових логічних схемах та комп'ютерних системах для виконання операцій логічної обробки інформації.

Ось коротка інформація про обидва ці типи пристройів:

1. Логічні вентилі:

Логічні вентилі — це електронні компоненти, які виконують основні логічні операції, такі як кон'юнкція (I), диз'юнкція (АБО), заперечення (НЕ) тощо. Вони можуть мати один або більше входів і один вихід. Кожен вхід може бути в стані "0" (наприклад, низький електричний рівень) або "1" (наприклад, високий електричний рівень), і вентиль виконує операцію відповідно до вхідних сигналів згідно зі своєю логічною функцією. Популярні типи логічних вентилів включають логічні I, АБО, НЕ, XOR, NAND та інші.

2. Напівсуматор:

Напівсуматор — це комбінаційна логічна схема, яка виконує дві операції: додавання двох однорозрядних чисел і визначення переносу. Він має два входи (A і B) для введення двох бінарних чисел та два виходи: один для суми (S) і один для переносу (C). Перенос вказує, чи виник перенос під час додавання двох чисел. Напівсуматор використовується як базовий блок для побудови суматорів та інших складніших арифметичних логічних пристройів у цифрових системах.

3. RS-триггер (ресет-сет триггер) — це один з основних типів логічних тригерів, який використовується в електроніці. Він отримав свою назву від англійських слів "Reset" (ресет) і "Set" (сет), що вказує на його основні функції — встановлення (сет) та скидання (ресет) стану.

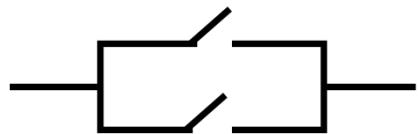
RS-триггер має два входи — вхід Set (S) і вхід Reset (R). Ці входи визначають його стан або значення. Вихід триггера позначається як Q. Залежно від значень на входах S та R, RS-триггер може перебувати в одному з чотирьох можливих станів: Set (S=1, R=0), Reset (S=0, R=1), Hold (S=0, R=0), або Indeterminate (S=1, R=1).

Ці типи пристройів є важливими в логіці та цифровому дизайні та використовуються для створення різноманітних логічних функцій та арифметичних операцій у цифрових пристроях та комп'ютерах.

Логічні вентилі, як дріт з електричним струмом:

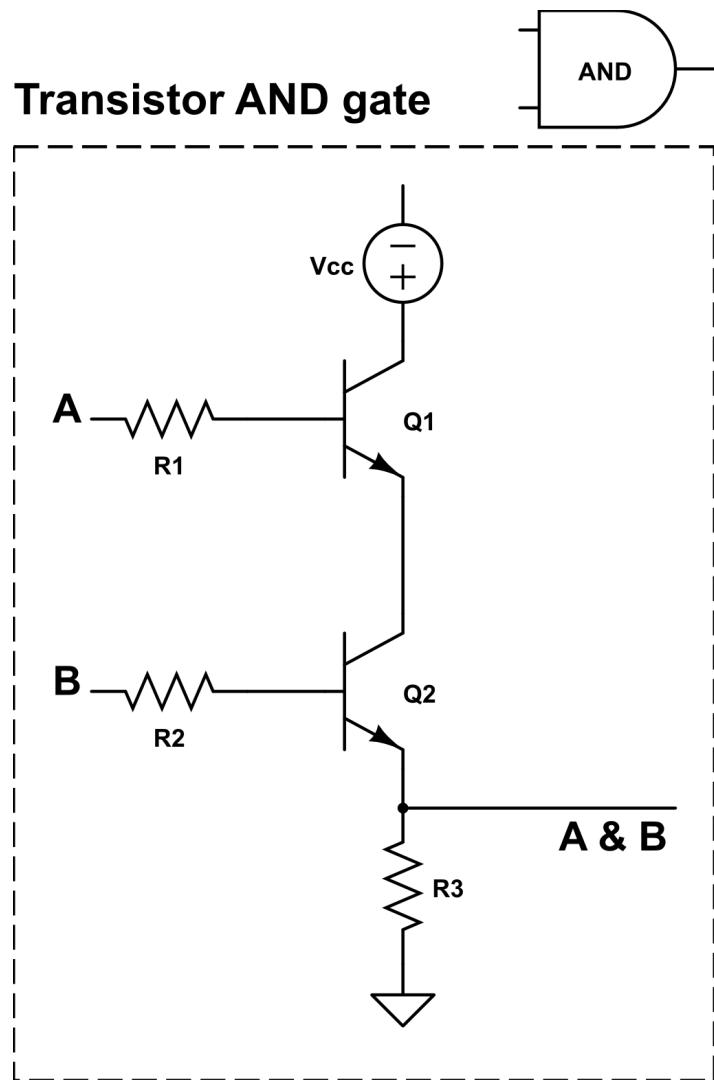


Вентиль — “І”. Якщо обидва контакти замкнуті, тоді по провіднику потече електричний струм, що буде означати в математичному сенсі 1, інакше 0.

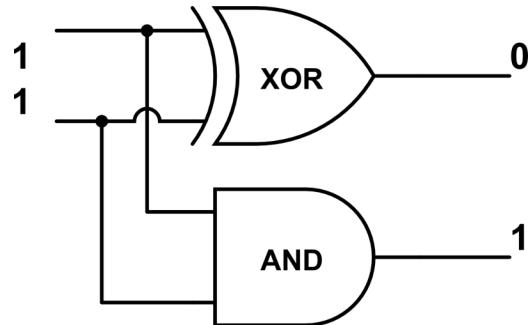


Вентиль — “Або”. Струм потече по провіднику якщо хоча б один з контактів замкнутий.

Принципова електрична схема вентиля AND



У листопаді 1937 року Джордж Стібіц, який тоді працював у Лабораторії Белла, завершив оснований на телефонних реле “Модель K” двійкового суматора (K означало “Кухонний стіл”, на якому Стібіц його зібраав).



Напівсуматор, реалізований на елементах “ВИКЛЮЧНЕ АБО” та “І”. Якщо напруга подається до кожного з двох входних контактів логічного вентиля “ВИКЛЮЧНЕ АБО”, то на виході цього вентиля напруги не буде. Такий же результат буде отримано, якщо на двох входах немає напруги. Вихід “ВИКЛЮЧНЕ АБО” буде передавати напругу, тільки якщо один і тільки один з входів знаходиться під напругою (а інший ні). Логічний вентиль “І” видасть напругу на вихід, тільки якщо обидва його входні контакти знаходяться під напругою. Об’єднавши ці два елементи, можна отримати напівсуматор, завданням якого є перенесення числа в поточний або наступний регістр. Напівсуматор передає число в наступний регістр, тільки якщо два входи під напругою, інакше він записує його в поточний регістр.

XOR (виключне АБО) — це одна з основних бінарних логічних операцій. Вона приймає два бінарних входи (зазвичай позначені як A і B), і її результат вираховується наступним чином:

Якщо тільки один з входів A або B рівний 1, а інший вход рівний 0, то результат XOR дорівнює 1.

Якщо обидва входи A і B однакові (або обидва 0 або обидва 1), то результат XOR дорівнює 0.

Логічний символ для операції XOR зазвичай виглядає як " \oplus ", і ця операція часто використовується в цифрових логічних схемах та комп'ютерних програмах для вирішення різних завдань. XOR має кілька важливих властивостей і застосувань, включаючи:

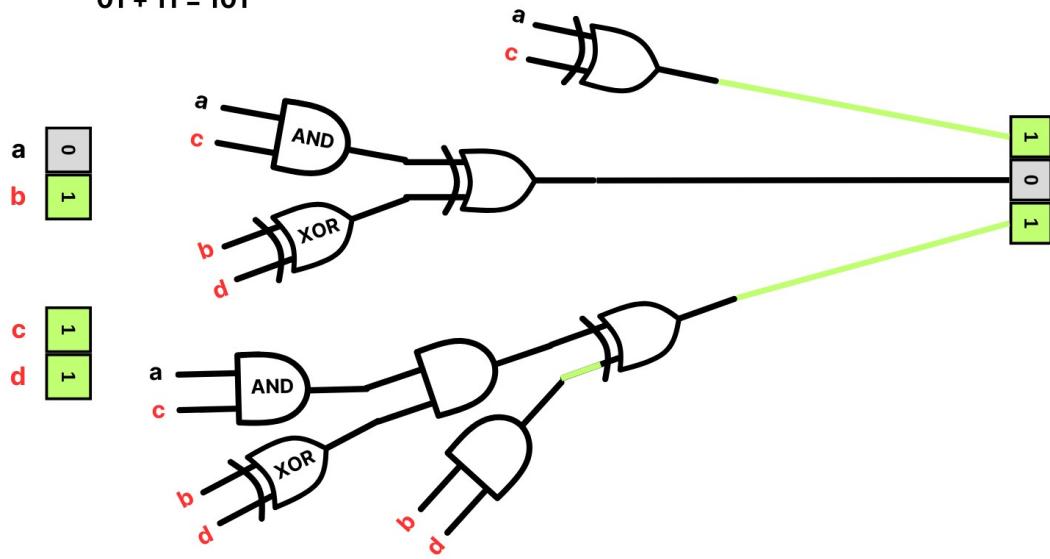
1. Переключення (zmіна) бітів: XOR може використовуватися для зміни значення бітів.
Наприклад, якщо ви використовуєте XOR для операції між бітом і ключем, то це допоможе вам зашифрувати або розшифрувати дані (як в шифрі Вернама).
2. Парність: XOR може бути використаний для визначення парності бітового рядка. Якщо кількість одиниць (1) у рядку непарна, то результат XOR буде 1, в іншому випадку - 0.

3. Обробка даних: XOR використовується для контролю цілісності даних і виявлення помилок.
4. Логічні операції: XOR використовується для створення складніших логічних операцій, таких як XNOR (негативне XOR).

XOR - це важлива складова логічних схем і операцій, і вона знайде своє застосування в різних аспектах обчислення та обробки інформації.

Схема додавання двох чисел

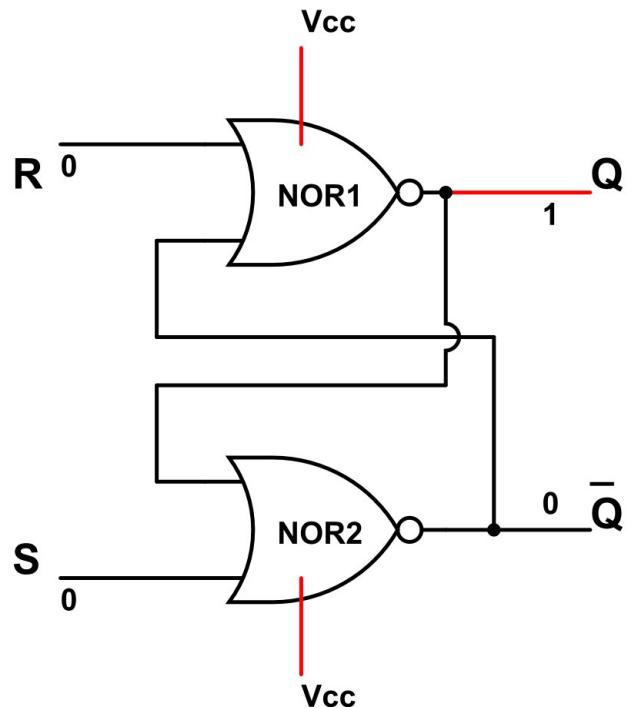
$2 + 3 = 5$
в двійковому форматі
 $01 + 11 = 101$



Тригер

RS-тригер на елементах NOR (Стрілка Пірса)

Вентиль NOR є істинним (1), лише коли два входи є хибними (0)



Що таке операційна система?

Сучасний комп'ютер — це машина, на яку можна встановити операційну систему.

Сучасні комп'ютери аналізують інформацію, здійснюють пошук інформації, генерують інформацію, виконують обчислення.

Сучасні комп'ютери мають обчислювальні здібності еквівалентні обчислювальним здібностям машини Тюрінга.

Операційна система (ОС) — це програма, яка буде керувати майже всіма процесами комп'ютера, зокрема, запуском і виконанням інших програм. Операційна система полегшить керування пристроями вводу та виводу, та операціями з пам'яттю комп'ютера, шляхом надання набору системних викликів, які можуть бути використані в програмах.

Процес в ОС — це програма, яка виконується через ОС.

Процесом керує операційна система. Операційна система відповідає за розподіл ресурсів, таких як час ЦП (центральний процесор) і пам'ять, для процесів.

Програма — це набір інструкцій, які може виконати комп'ютер.

Уявимо машину, яка може виконувати наступні команди:

$r(i)$ - зміст регістру i .

1. Load a, завантажте a до регістру 0;
2. Store a, збережіть $r(0)$, до регістру a;
3. Add a, виконайте $r(0) + a$ і збережіть до регістру 0;
4. Sub a, виконайте $r(0) - a$ і збережіть до регістру 0;
5. Mult a, виконайте $r(0) * a$ і збережіть до регістру 0;
6. Div a, виконайте $r(0) / a$ і збережіть до регістру 0;
7. Jump a, лічильник команд до a.
8. JZERO a, якщо $r(0) = 0$, лічильник команд встановлюється до a, якщо ні до наступної команди.
9. Halt, припинити роботу.

Усі ці команди мають числові представлення, які зберігаються в пам'яті та можуть бути динамічно (в роботі) змінені.

Цей набір команд представляє RASP.

RASP (Random-access stored-program machine — машина зі збереженою програмою та довільним доступом).

1. RASP має пам'ять, яка поділена на клітинки (реєстри), які мають власні номери.
2. RASP має "блок управління пам'яттю", який може зберігати, перезберігати, читувати дані з регістра за номером його позначки.
3. RASP має "блок керування програмою", який може читати нумерований список рядків, які є командами, і виконувати ці команди.
4. RASP має лічильник команд, які зберігають номер поточної команди, за замовчуванням він починається з 1.

В нашому випадку, програма — це список команд, кожна з яких має числове представлення. Програма зберігається в частині пам'яті та може бути змінена “блоком управління пам'яттю”, тоді “блок керування програмою” може прочитати оновлену команду.

Початкові дані для машини записують безпосередньо в реєстри, а результати роботи читаються безпосередньо з реєстрів (ми не визначаємо спеціальний інтерфейс для введення та виведення даних).

5. RASP має кнопку запуску, яка запускає “блок керування програмою” з першого рядка програми і після виконання лічильник команд збільшується на одиницю і виконує наступну команду в списку.

Команди RASP (Дуже проста мова асемблера, яка в пам'яті є як машинний код).

А тепер уявимо, що ми з допомогою цих команд написали програму, яка може запускати інші програми (процеси).

Це й буде операційна система.

Насправді операційну систему можна написати мовою асемблера, або мовою високого рівня як С (стандарт ISO/IEC 9899:2018).

Компілятор перетворить програму мовою високого рівня в програму на мові, яку розуміє комп'ютер (процесор), зазвичай це бінарний (двійковий) машинний код.

Компілятор — це програма, яка перетворює програму одною мовою в програму іншою мовою.

Компілятор оптимізує програму, перевіряє помилки та переводить її на іншу мову. Зазвичай, на мову нижчого рівня.

Операційну систему можна реалізувати в рамках іншої операційної системи, наприклад, на мові JavaScript (стандарт ECMA-262), яка не компілюється, а інтерпретується.

Інтерпретатор — це програма, яка перетворює комп'ютер в певну віртуальну машину, яка напряму розуміє мовою високого рівня й може виконувати її команди.

Сучасний комп'ютер — це машина, яка може мати багато станів, які змінюються з допомогою інтерфейсу, і яка має зворотний зв'язок, тобто зміни стану реєстрів комп'ютера в результаті маніпуляцій з інтерфейсом комп'ютера впливають на подальшу поведінку інтерфейсу.

Тепер подивимося, як працює стандартний сучасний комп'ютер.

Коли ви вмикаєте електричне живлення комп'ютера, він запускає операційну систему, яка на ньому встановлена.

Операційна система — це відносно велика програма, встановлення якої на комп'ютер було враховано заздалегідь при розробці комп'ютерних схем. Операційна система запускається BIOS (Біос) і центральним процесором. Центральний процесор є основним обчислювальним центром комп'ютера, він побудований з використанням напівпровідників і логічних вентилів (булевої алгебри).

BIOS (базова система вводу/виводу) — це мікросхема, в якій записана фірмова програма. Програма в BIOS (Біос) у поєднанні з апаратними методами відповідає за запуск операційної системи та за системи введення та виведення інформації в реєстри комп'ютера. В операційній системі реалізований зручний інтерфейс для роботи з комп'ютером, в ній вже передбачені всі стандартні функції. Таким чином, вам не потрібно створювати програму для запису файлів в пам'ять комп'ютера, потрібно лише виконати команду збереження, вказати файл і місце збереження. Якщо ви встановите віртуальну машину (з допомогою ОС) для інтерпретації мови, яка вам потрібна, то ваш комп'ютер зможе виконувати коди вашої мови програмування. Коли ви починаєте виконувати файл з кодами на потрібній вам мові програмування, відповідна віртуальна машина починає виконувати цей код, при цьому віртуальна машина активно використовує можливості операційної системи. Виведення візуальної інформації на екран комп'ютера зазвичай здійснюється відеокартою, яка бере на себе навантаження по обробці візуальної інформації, за аналогією і зі звуковою інформацією, є звукова карта. Мережева карта відповідає за мережевий зв'язок між комп'ютерами. Для зображення текстових символів, тобто букв, прийнятий певний набір кодів. Коли ви зберігаєте текстовий файл, текстовий редактор автоматично призначає вашому файлу кодування. Сам файл, по суті, є набором двійкових кодів, записаних у певному місці в довготривалій пам'яті комп'ютера, але коли ви відкриваєте файл для перегляду, комп'ютер, орієнтуючись на кодування, автоматично показує деякі коди у вигляді літер. Відомі таблиці символів ASCII та Unicode, вони визначають таблицю міжсимвольних зв'язків, тобто стандартизують зображення символів.

Драйвери — це посередники між операційною системою та апаратними пристроями комп'ютера, що використовуються для обміну даних між ними. Драйвер — комп'ютерна програма, за допомогою якої операційна система отримує доступ до певного приладу чи частини апаратного забезпечення. Однією з перших операційних систем була система OS/360.

Операційна система 360 (або OS/360) — сімейство операційних систем, розроблених компанією IBM для мейнфреймів System/360 в період починаючи з 1964 року. Історія створення OS/360 згадується в відомій книзі Фредеріка Брукса "Міфічний людино-місяць" (1975).

Сучасні операційні системи в основному під впливом системи UNIX та інтерфейсу, який презентував американець Дуглас Енгельбарт в 1968 році, і який вплинув на інтерфейс комп'ютера Зірокс Альто (Xerox Alto, 1970) та на інтерфейси операційних систем Mac та Windows. Дуглас Енгельбарт (1925 — 2013) в 1968 році в Сан-Франциско продемонстрував комп'ютер, інтерфейс якого дозволяв відкривати окремі віконця на робочому столі та рухати їх мишкою, яка контролювала курсор на екрані.

UNIX (Юнікс) — сімейство переносних, багатозадачних і розрахованих на багато користувачів операційних систем, які засновані на ідеях оригінального проекту AT&T UNIX, розробленого в 1970-х роках в дослідному центрі Bell Labs Кеном Томпсоном, Деннісом Рітчі (1941 — 2011) та іншими. В 1982 році Денніс Рітчі описував UNIX так: “Операційна система UNIX в основному складається з трьох частин: 1. Ядро або власне операційна система є частиною, яка відповідає за керування машини та контролює планування різноманітних програм користувача. 2. Shell або інтерпретатор команд піклується про спілкування між користувачем і самою системою. 3. Допоміжні програми (насправді найбільша частина), які виконують певні завдання, як-от редактування файлу, іншими словами, усі інші програми, які безпосередньо не є частиною ядра операційної системи”.

Багатотермінальні системи, що працюють в режимі поділу часу, стали першим кроком на шляху створення локальних обчислювальних мереж.

Ендрю Таненбаум створив ОС MINIX, щоб продемонструвати принципи, викладені в його підручнику “Операційні системи: проектування та реалізація” (1987). Ранні версії MINIX були створені Ендрю Таненбаумом для освітніх цілей. Починаючи з MINIX 3, основна мета розробки змістилася з освіти на створення високонадійної та самовідновлюваної мікроядерної ОС.

MINIX 3 — це безоплатна операційна система з відкритим вихідним кодом, розроблена як високонадійна, гнучка та безпечна. Він заснований на крихітному мікроядрі, що працює в режимі ядра, а решта операційної системи працює як кілька ізольованих, захищених процесів у режимі користувача. Він працює на процесорах x86 і ARM.

Мікроядро (Microkernel) — це концепція архітектури операційної системи, де базовий функціонал ядра обмежується до найнеобхідніших операцій, а інші сервіси та функції виносяться в простір користувача.

Основна ідея полягає в тому, щоб максимально розділити різноманітні функції операційної системи, забезпечуючи високу модульність та гнучкість системи.

Деякі приклади операційних систем з мікроядерною архітектурою включають QNX, MINIX, та L4.

Звичайним мікроядерним підходом є реалізація більшості функцій операційної системи в процесах користувача. Щоб отримати послугу, наприклад читання блоку файлу, процес користувача (процес клієнта) надсилає запит серверному процесу, який потім виконує роботу та надсилає відповідь.

Помилки в одному модулі мікроядра не повинні впливати на інші модулі, що може поліпшити стійкість системи.

Класичні мікроядра реалізують лише дуже обмежений набір низькорівневих примітивів, або системних викликів, що являють собою базові сервіси операційної системи.

До них належать:

- керування адресним простором оперативної пам'яті
- керування адресним простором віртуальної пам'яті

- керування процесами і потоками
- засоби міжпроцесної взаємодії.

Всі інші сервіси ОС, які в класичних монолітних ядрах ОС реалізуються безпосередньо ядром, в мікроядерній архітектурі реалізуються в користувацькому адресному просторі.

Переривання (interrupt) в операційних системах (ОС) — це механізм, що дозволяє пристроям або програмам "переривати" нормальній хід виконання програми для того, щоб передати управління операційній системі або іншій частині комп'ютерної системи. Це потрібно для обробки різних подій, таких як введення користувача, звернення до пристройів вводу-виводу, помилки або інші події, що виникають у системі.

Переривання можуть бути програмними або апаратними. Програмні переривання генеруються програмним кодом, наприклад, при виклику операцій системних служб або обробці винятків. Апаратні переривання спричинені зовнішніми подіями, такими як натискання клавіші на клавіатурі, завершення передачі даних через мережу або таймером, що сигналізує про минулий час.

Коли переривання виникають, виконання поточного коду призупиняється, а управління передається до обробника переривань, який вирішує, як реагувати на переривання. Обробка переривань дозволяє операційній системі реагувати на різноманітні події в системі та ефективно керувати ресурсами.

При виникненні переривання процесор зберігає в стеку вмісту лічильника команд і завантажує в нього адресу відповідного вектора переривання, в якому, як правило, міститься команда безумовного переходу до підпрограми обробки переривання. Останньою командою підпрограми-обробника переривань має бути команда повернення з переривання, яка забезпечує повернення в основну програму шляхом відновлення значення заздалегідь збереженої лічильника команд.

Що таке процес в операційній системі?

Процес в операційній системі — це основна одиниця управління завданнями і виконанням програм в операційній системі. Він являє собою програму в стані виконання, разом із всіма ресурсами й даними, необхідними для її виконання. Процеси дозволяють операційній системі керувати багатьма різними завданнями одночасно, а також забезпечувати ізоляцію між ними, що забезпечує стабільність та безпеку системи.

Основні характеристики процесу в операційній системі включають:

1. Програмний код: Процес містить в собі виконуваний програмний код, який визначає, які операції повинні бути виконані.
2. Власні області пам'яті: Кожен процес має свої власні області пам'яті для зберігання виконуваного коду, даних та стеку викликів.
3. Регістри та змінні: Процеси можуть мати власні регістри та змінні, що використовуються для зберігання проміжних результатів обчислень.
4. Системні ресурси: Операційна система надає процесам доступ до ресурсів, таких як процесорний час, пам'ять, введення/виведення, мережеві ресурси та інші.
5. Стан виконання: Кожен процес може перебувати у різних станах виконання, таких як активний, призупинений, готовий до виконання або завершений.
6. Ідентифікатор процесу: Кожен процес має унікальний ідентифікатор, який використовується для ідентифікації та управління процесами в операційній системі.

Операційна система відповідає за планування виконання процесів, розподіл ресурсів, управління станами процесів та забезпечення безпеки та ізоляції між ними. Керування процесами є важливою функцією операційної системи, яка дозволяє ефективно використовувати обчислювальні ресурси та забезпечувати стабільну та надійну роботу комп'ютерної системи.

Потік (thread) або повніше потік виконання — в інформатиці так називається спосіб програми розділити себе на дві чи більше паралельні задачі. Реалізація потоків та процесів відрізняються в різних операційних системах, але загалом потік міститься всередині процесу і різні потоки одного процесу спільно розподіляють деякі ресурси, у той час, як різні процеси ресурси не розподіляють. На багатопроцесорних чи на багатоядерних системах робота потоків здійснюється справді одночасно, оскільки різні потоки і процеси виконуються буквально одночасно різними процесорами або ядрами процесора.

Що таке проблема обідаючих філософів?

Проблема обідаючих філософів є класичним прикладом синхронізаційної проблеми, яка виникає в багатозадачних програмах і має за мету продемонструвати проблеми, пов'язані з конкуренцією за ресурси. Ця проблема часто використовується для ілюстрації принципів мультипроцесорного програмування та синхронізації в середовищах, де кілька потоків або процесів спільно використовують загальні ресурси.

Сценарій проблеми обідаючих філософів виглядає так:

1. Є п'ять філософів, які сидять за круглим столом.
2. Кожен філософ має перед собою тарілку з їжею.
3. Між кожною парою філософів лежить паличка для їжі.
4. Філософ може виконувати дві дії: думати і їсти.
5. Щоб поїсти, філософ повинен узяти дві палички (ліву та праву), а потім взяти їжу.
6. Після призначеного обіду філософ кладе палички назад на стіл.

Проблема полягає в тому, як синхронізувати дії філософів так, щоб уникнути взаємоблокування (deadlock) та голодування (starvation), коли деякі філософи можуть не отримати можливість поїсти через невірну синхронізацію.

Різні алгоритми та підходи в програмуванні можуть бути використані для вирішення проблеми обідаючих філософів, включаючи семафори, блокування та інші механізми синхронізації. Ця проблема допомагає розуміти важливі концепції синхронізації та уникнути проблем, пов'язаних з одночасним доступом до загальних ресурсів в багатозадачних системах.

Проблема обідаючих філософів може бути вирішена з використанням семафорів або мютексів (mutexes), які є інструментами для синхронізації доступу до ресурсів в багатозадачних програмах. Ось спосіб вирішення цієї проблеми з використанням семафорів:

Створіть масив семафорів, де кожен елемент масиву представляє одну паличку для їжі.

Ініціалізуйте всі семафори в стані "1", що означає, що всі палички вільні.

Кожен філософ повинен використовувати два семафори: один для лівої палички та інший для правої палички.

Філософ повинен застосовувати операції з семафорами "wait" (чекати) та "signal" (сигнал), щоб отримати доступ до паличок для їжі.

Псевдокод для філософа може виглядати так:

```
while True:  
    think() # Філософ думає  
    take_left_chopstick() # Бере ліву паличку  
    take_right_chopstick() # Бере праву паличку  
    eat() # Їсть  
    put_right_chopstick() # Покладає праву паличку  
    put_left_chopstick() # Покладає ліву паличку
```

Операції "take_left_chopstick", "take_right_chopstick", "put_left_chopstick" і "put_right_chopstick" реалізовані з використанням семафорів і виконуються за допомогою "wait" і "signal". Наприклад:

```
def take_left_chopstick():
    left_chopstick_semaphore.acquire()

def take_right_chopstick():
    right_chopstick_semaphore.acquire()

def put_left_chopstick():
    left_chopstick_semaphore.release()

def put_right_chopstick():
    right_chopstick_semaphore.release()
```

Цей підхід до вирішення проблеми обідаючих філософів з використанням семафорів забезпечує синхронізацію доступу до паличок для їжі та гарантує, що філософи не будуть одночасно брати одну паличку, що може привести до взаємоблокування.

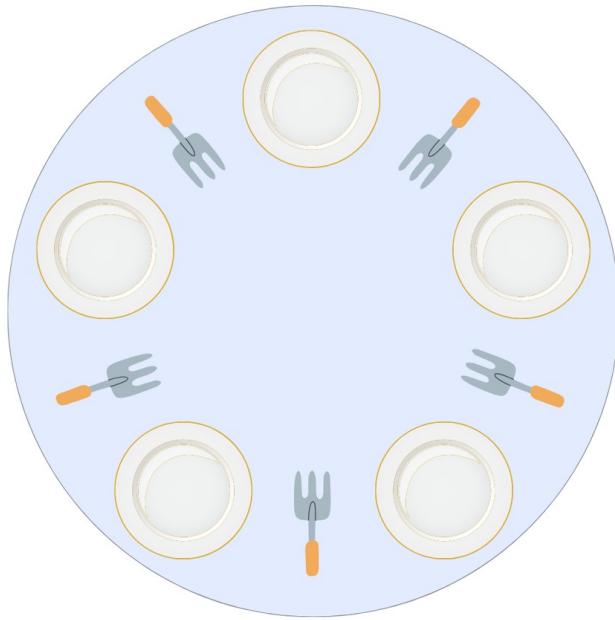
Семафор — це концепція, яка використовується для керування доступом до ресурсів, особливо в багатозадачних або багатопроцесорних середовищах. Семафор — це змінна або структура даних, яка дозволяє потокам (або процесам) синхронізувати свою роботу та взаємодіяти між собою, щоб уникнути конфліктів і гарантувати правильний порядок виконання.

Існують два основних типи семафорів:

1. Бінарний семафор (мютекс): Це семафор з двома можливими станами - вільний і заблокований. Він використовується, коли потрібно дозволити або заборонити доступ до ресурсу лише одному потоці (процесу) в певний момент часу.
2. Семафор лічильник (mutex): Цей тип семафора дозволяє обмежити доступ до ресурсу для певної кількості потоків або процесів. Він зберігає лічильник, який може бути збільшений або зменшений потоками для контролю доступу до ресурсу.

Семафори часто використовуються в операційних системах і при розробці багатозадачних програм для управління конкурентним доступом до ресурсів, таких як файли, принтери, мережеві з'єднання тощо. Вони є важливими інструментами для забезпечення коректності та надійності взаємодії потоків в програмах і операційних системах.

Проблема філософів, що обідають.



У 1965 році Дейкстра поставив і розв'язав проблему синхронізації, яку він назвав “проблемою філософів, які обідають”.

Едсгер Вібе Дейкстра (1930–2002) був нідерландським пionером у галузі обчислювальної техніки.

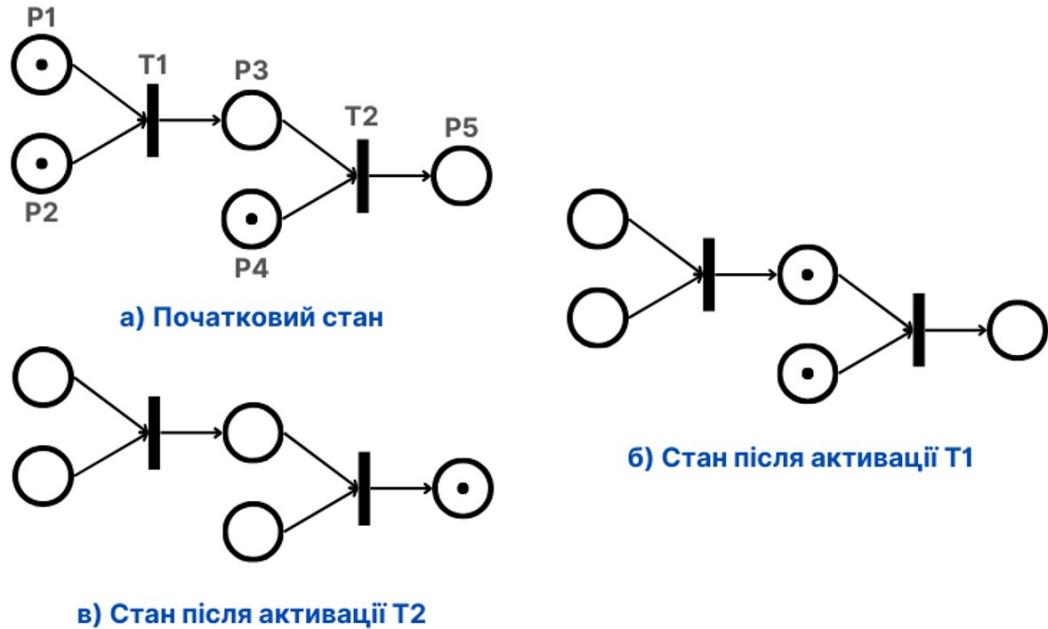
Проблему обідаючих філософів можна ще сформулювати так: П'ятеро філософів сидять за круглим столом. У кожного філософа є тарілка спагеті. Спагеті настільки слизькі, що філософу потрібні дві вилочки, щоб їх з'їсти. Між кожною парою тарілок одна вилка. Ми спростили й припустимо, що життя філософа складається з чергування періодів їжі та мислення. Коли філософ зголодніє, він намагається дістати свою ліву і праву вилку, одну за одною, в будь-якому порядку. Якщо йому вдається взяти дві вилки, він деякий час їсть, потім відкладає вилки та продовжує думати. Питання таке: чи можете ви написати програму для кожного філософа, яка виконує те, що має робити, і ніколи не застряє?

Проблема обідаючих філософів корисна для моделювання процесів, які змагаються за ексклюзивний доступ до обмеженої кількості ресурсів, таких як пристрой введення-виводу, читання-запису.

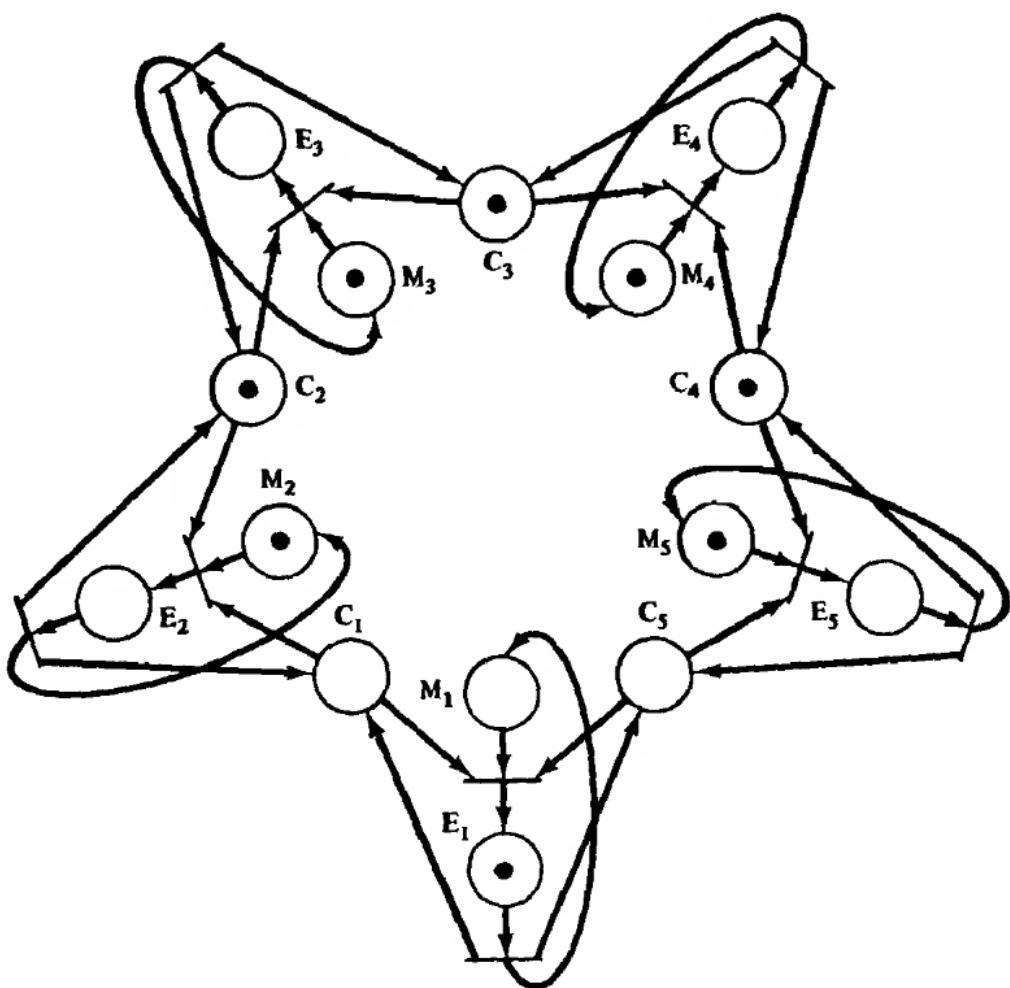
Мережі Петрі є популярною та корисною моделлю для представлення дискретних динамічних систем. Вони названі на честь Карла Адама Петрі, який розробив їх на початку 1960-х років в Боннському університеті в Німеччині. Мережа Петрі – це орієнтований граф з двома типами вузлів – місцями та переходами. Місця малоються у вигляді кіл, а переходи — у вигляді смуг. Направлені дуги (стрілки) з'єднують “місця” з “переходами” та “переходи” з “місцями”. Для кожного переходу спрямовані дуги визначають його вхідні місця (від місця до переходу) і вихідні місця (від переходу до місця). Мережа Петрі виконується шляхом визначення розмітки, а потім запуску переходів. Розмітка — це розподіл жетонів на місця мережі Петрі. Жетон представлений на графіку мережі Петрі маленькою суцільною крапкою на місці. Переход увімкнено, коли всі його місця введення мають один або кілька жетонів (маркерів). Переход запускається шляхом видалення одного жетону з кожного місця входу та додавання одного жетону до кожного місця виходу. З допомогою мереж Петрі можна змоделювати систему видачі та отримання наказів, або чергу виконання команд процесором комп’ютера. “Перехід” можна підключати тільки до “місця”, а “місце” тільки до “переходу”. “Перехід” в мережах Петрі спрацьовує коли всі під’єднані до нього місця мають принаймні по одному жетону. Коли “перехід” спрацьовує він генерує по одному жетону на кожен свій вихід. “Місця” в мережах Петрі може містити декілька жетонів. До “місця” можуть бути під’єднані декілька виходів, які спрацьовують тільки, якщо для всіх виходів достатньо жетонів, тобто один вихід — один жетон. Якщо від “місця” до “переходу” ведуть два шляхи (стрілки), тоді “місце” повинно містити два жетони, щоб спрацював переход. Коли “перехід” спрацьовує жетони переходять в відповідне місце.

Складність кібернетичної системи визначається кількістю станів, які система може приймати та кількістю параметрів (зв'язків), які впливають на зміну стану.

Приклад простої мережі Петрі.

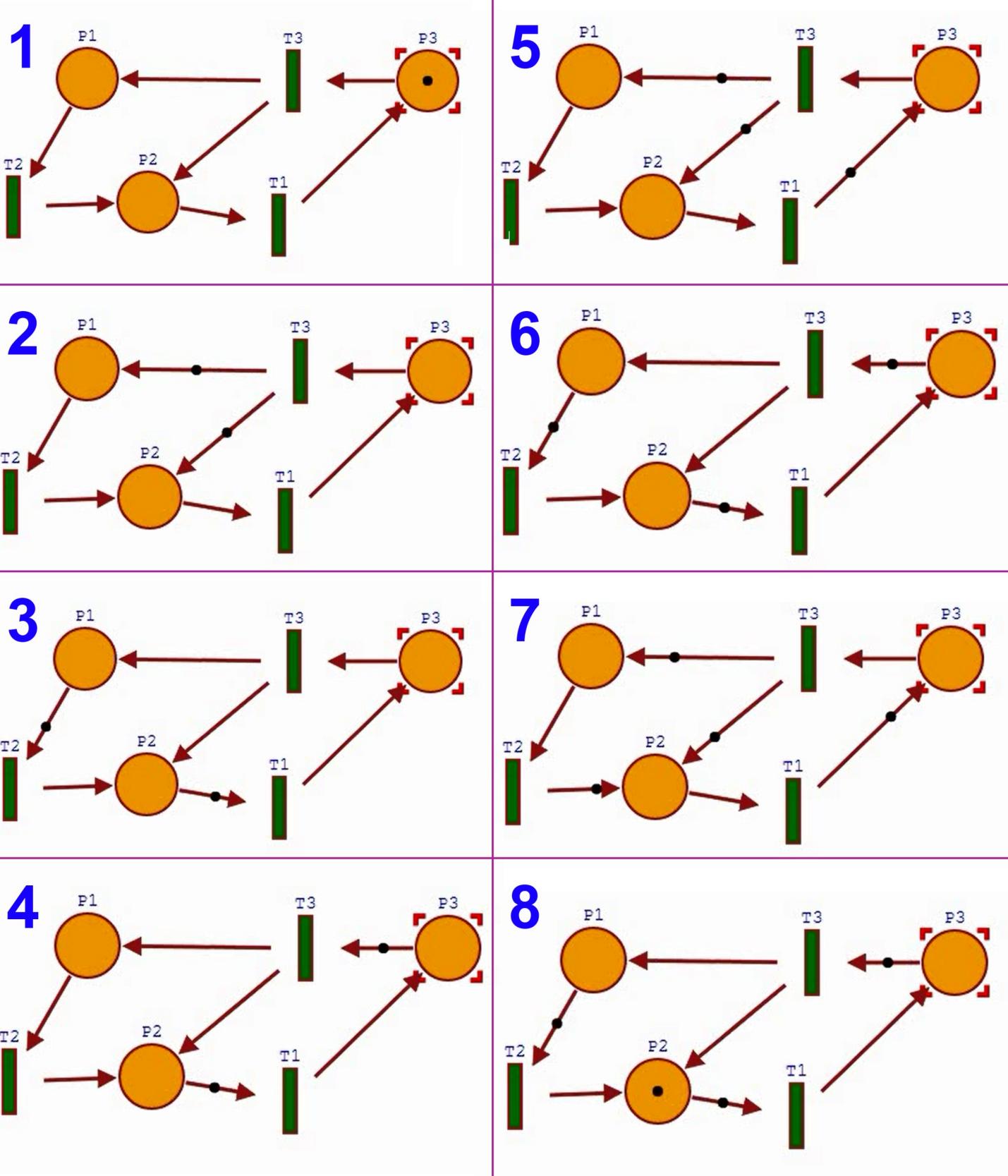


Проблему обідаючих філософів запропонував Едсгер Дейкстра. Вона асоціюється з п'ятьма філософами, які по черзі думали і їли. Філософи сидять за великим круглим столом із безліччю китайських страв. Між сусідами одна паличка. Однак для китайської їжі потрібні дві палички, тому кожен філософ повинен взяти паличку для їжі зліва і паличку для їжі праворуч. Проблема, звичайно, полягає в тому, що коли всі філософи візьмуть палички ліворуч, а потім чекатимутъ, поки палички з правого боку звільнятися, вони чекатимуть вічно й помрутъ з голоду. На малюнку показано розв'язання цієї задачі з допомогою мережі Петрі. Позиції C_1, \dots, C_5 представляють палички для їжі, і оскільки кожна з них спочатку вільна, то в початковій позначці в кожній із цих позицій є жетон (чорне коло). Кожен філософ представлений двома позиціями M_i та E_i для станів мислення та їжі відповідно. Щоб філософ перейшов зі стану мислення в стан їжі, обидві палички (ліва і права) повинні бути вільними.



Що таке мережі Петрі?

Мережа Петрі є однією з кількох мов математичного моделювання для опису розподілених систем. Це клас дискретної динамічної системи подій. Мережа Петрі — це орієнтований дводольний граф, який має два типи елементів, місця та переходи, зображені у вигляді білих кіл і прямокутників відповідно. Місце може містити будь-яку кількість жетонів, зображених у вигляді чорних кіл. Переход увімкнено, якщо всі місця, підключені до нього як вхідні дані, містять принаймні один жетон (маркер). Мережа Петрі складається з місць, переходів і дуг між ними. Дуги проходять від місця до переходу або навпаки, ніколи між місцями або між переходами. Місця, з яких дуга йде до переходу, називаються вхідними місцями переходу. Місця, до яких відходять дуги від переходу, називаються вихідними місцями переходу. Графічно місця в мережі Петрі можуть містити дискретну кількість позначок, які називаються жетонами (маркерами). Будь-який розподіл жетонів по місцях представляє конфігурацію мережі, яка називається маркуванням. Переход мережі Петрі може спрацювати, якщо він включений, тобто є достатня кількість маркерів на всіх його входах. Коли спрацьовує переход, він споживає необхідні вхідні маркери та створює маркери на своїх вихідних місцях.



Що таке віртуальна машина?

Віртуальна машина — це програмний емулятор фізичної комп'ютерної системи, яка працює в середовищі ізольованому від реальної апаратної платформи. Вона дозволяє виконувати операційні системи та програми в ізольованому середовищі, яке називається "віртуальною машиною", над реальною операційною системою (господарем). Ця технологія дозволяє одному фізичному серверу або комп'ютеру виконувати декілька віртуальних машин з різними операційними системами та додатками.

Віртуальні машини надають кілька переваг:

1. **Ізоляція:** Вони дозволяють ізолювати одну віртуальну машину від інших та від господарської операційної системи. Це забезпечує безпеку та стабільність роботи віртуальних середовищ.
2. **Гнучкість:** Ви можете легко створювати, запускати, зупиняти і переміщати віртуальні машини. Це полегшує управління ресурсами та дозволяє швидко реагувати на змінні потреби.
3. **Відновлення та резервне копіювання:** Віртуальні машини дозволяють створювати резервні копії та забезпечувати відновлення системи, що полегшує роботу і забезпечує високу доступність.
4. **Економія ресурсів:** Ви можете спільно використовувати ресурси фізичного сервера між декількома віртуальними машинами, що призводить до більш ефективного використання обладнання.

Віртуальні машини поділяються на 2 головні категорії, в залежності від їх використання та відповідності до реальної апаратури:

-системні (апаратні) віртуальні машини, що забезпечують повноцінну емуляцію всієї апаратної платформи та відповідно підтримують виконання операційної системи.

-прикладні віртуальні машини, які розроблені для виконання лише застосунків (прикладних програм), наприклад, Віртуальна машина Java.

Наприклад, браузери використовують будований JavaScript-двигун, який інтерпретує і виконує JavaScript-код, використовуючи віртуальну машину для JavaScript (наприклад, V8 для Google Chrome). Такі віртуальні машини використовуються для виконання JavaScript-коду в ізольованому середовищі браузера, щоб забезпечити безпеку та ефективність виконання вебсторінок.

Віртуальна машина Java (Java Virtual Machine або JVM) — це віртуальна машина, яка використовується для виконання програм, написаних мовою програмування Java. JVM перетворює байт-код Java (скомпільований код з програми Java) на низькорівневий машинний код, який виконується на конкретній апаратній платформі.

Основні функції JVM включають:

1. **Виконання байт-коду:** JVM перетворює байт-код Java на машинний код, який може виконуватися на реальних комп'ютерах.
2. **Управління пам'яттю:** JVM відстежує виділення і звільнення пам'яті для об'єктів Java, автоматично виконуючи збірку сміття.
3. **Управління виконанням:** JVM керує виконанням Java-програм, забезпечуючи безпеку та надійність виконання.

4. Віртуалізація: JVM надає віртуальне середовище, яке дозволяє виконувати Java-програми на різних платформах без необхідності зміни початкового коду.

Однією з головних переваг Java є її можливість використовувати JVM. Це дозволяє розробникам писати один раз і виконувати програми на різних платформах, що робить Java мовою програмування "напишіть один раз, виконайте будь-де". Багато різноманітних додатків і вебслужб на Java базуються на використанні JVM для виконання коду.

Байт-код — це код, який являє собою список команд (опкодів) та їх операндів. Команди представлені байтом, і може бути до 255 команд. Стандартний байт-код описаний в специфікації інтерпретатора мови Java. Байт-код зазвичай не виконується напряму процесором, але його можна перевести у машинний код на порядок швидше, ніж початковий код мови Java. Деякі команди можуть вимагати один байт для коду команди та один або кілька додаткових байтів для параметрів. По структурі він відповідає мові асемблера.

Команда,

Операнд,

Операнд,

Команда,

Що таке мова програмування?

Мова програмування — це мова, якою програмуєш комп'ютер, в сенсі, вчиш комп'ютер нових алгоритмів.

“Мови програмування — це нотації для опису обчислень для людей і машин” (книга “Компілятори” Альфреда Ахо)

Якщо мова не дозволяє навчити комп'ютер нових алгоритмів, це не мова програмування, це мова розмітки (HTML, XML), запитів (SQL, GraphQL), конфігурацій (YAML) тощо.

Це означає, що чиста декларативна мова, не є мовою програмування, а є мовою для спілкування.

Наприклад, ткацький верстат Жаккарда (19 ст), який читував перфокарти й на основі них створював візерунок на тканині не можна вважати машиною, яка програмується, бо станок виконував одну задачу: зчитуючи отвори на картах, підіймав ту чи іншу нитку, щоб під нею пробіг ткацький човник. Програмування в строгому сенсі не просто задає дані для обробки машині, а й вчить машину як ці дані обробляти.

До 20 століття алгоритми записувалися неформально, бо їх повинні були розуміти люди. Але вчені захотіли створити мову, яка б дозволяла записувати алгоритми так, щоб елементарна машина могла їх зрозуміти. Так була створена мова програмування, яка включала один вічний цикл, умовний оператор, операцію зупинки циклу, арифметичні операції. Відомий алгоритм Евкліда, ще до нашої ери створений, потрібно було якось комп'ютеру пояснити, от й створили машину Тюрінга з її елементарною мовою, потім лямбда-числення, мову асемблера, Cobol, Fortran, Algol, Lisp, Simula, Pascal, C, C++.

Мова програмування — це формальний засіб комунікації, який використовується для написання програм для комп'ютерів. Це система символів, словесних правил і синтаксичних структур, які дозволяють програмістам створювати інструкції для комп'ютера для виконання певних завдань.

Мови програмування використовуються для розробки різноманітних програм і додатків, від вебсайтів і мобільних додатків до наукових обчислень і системного програмування. Вони можуть мати різні рівні складності та призначення, і кожна мова має свої особливості та властивості.

Популярні приклади мов програмування включають Python, Java, C++, JavaScript, Ruby, PHP і багато інших. Кожна мова має свою власну сферу застосування і може бути краще підходити для певних видів завдань. При виборі мови програмування важливо враховувати потреби проекту і власні навички програмування.

Структурне програмування — методологія програмування, запропонована в 1970-х роках голландським науковцем Дейкстрою, була розроблена та доповнена Ніклаусом Віртом.

Відповідно до цієї методології, будь-яку програму можна створити, використовуючи три конструкції:

1. послідовне виконання — одноразове виконання операції в порядку запису їх в тексті програми;
2. розгалуження — виконання певної операції або декількох операцій залежно від стану певної, наперед заданої умови;
3. цикл — багаторазове виконання операції або групи операцій за умови виконання деякої наперед заданої умови.
4. Зміні та арифметичні операції.

“Мова програмування, хочемо ми цього чи ні, впливає на наше мислення”

(Едсгер Вібе Дейкстра, “Дисципліна програмування”, 1976)

Ось так ми б писали код, якби не було умовних операторів.

```
const a = 5;  
const b = 2;  
  
const state = 1;  
  
const result = (a * (1-state) + b * state);  
  
console.log(result); //2, or 5 if state 0
```

Мова програмування — це мова, якою можна писати комп’ютерні програми, тобто написати набір інструкцій, які виконуватиме комп’ютер.

Специфікація мова програмування — це набір символів, правил їх комбінування та правил, що визначають їхні ефекти під час виконання комп’ютером. Специфікація мови програмування задає граматику мови, її алфавіт, синтаксис та семантику. Нотація Бекуса-Наура використовується як метамова для задання контекстно-вільної граматики мови програмування.

Визначення мови програмування, зокрема, дає американська програмістка Джін Саммет в книзі "Мови програмування: історія й основи" (1969).

Для того, щоб комп’ютер зрозумів мову програмування, потрібна спеціальна програма, яка виконуватиме вирази мови високого рівня або спочатку переведитиме їх у машинний код, а потім виконуватиме.

Машинний код (машинну мову) можна розглядати як мову програмування, команди якої безпосередньо здатен виконувати комп’ютер, тобто без додаткових програм. Як правило, машинний код являє собою послідовний набір інструкцій у двійковому коді. Такий машинний код можна вважати мовою найнижчого рівня. Деякі програмісти не вважають машинний код мовою програмування, а починають з мови асемблера, мови низького рівня.

Мови низького рівня дозволяють писати максимально оптимальний код.

Для того, щоб комп’ютер зрозумів мову високого рівня (C++, C#, Java, Python), потрібна спеціальна програма-транслятор або віртуальна машина. Деякі мови перекладаються на машинний код, а потім безпосередньо виконуються комп’ютером, а деякі виконуються інтерпретатором, іншими словами, віртуальною машиною.

Віртуальна машина — це програма, яка встановлюється на комп’ютер і може виконувати певну мову програмування. Таким чином, віртуальна машина ніби перетворює комп’ютер на машину, здатну розуміти мову програмування високого рівня.

Перевага інтерпретованих мов полягає в тому, що їх не потрібно перекладати перед запуском, крім того, кожна програма, яка перекладається (трансліюється) на машинний код, може виконуватися лише на певному комп’ютері, на певному процесорі, на відміну від інтерпретованих програм. Недоліком інтерпретованих програм є те, що їх перевірка та оптимізація може виконуватися інтерпретатором під час процесу виконання, що займає час. Щоб усунути цей недолік, деякі мови програмування спочатку трансліюються в проміжний код (байт-код), який оптимізується і перевіряється на наявність помилок, а проміжний код виконується віртуальною машиною.

Мови програмування високого рівня відрізняються від мов низького рівня кількома ознаками, а саме:

1. Мови високого рівня мають вищий рівень абстракції, ніж мови низького рівня. Наприклад, поняття функції, класу, об’єкта, кортежу, зазвичай не зустрічається в мовах низького рівня.
2. Мови високого рівня зазвичай мають набагато більше готових рішень, тобто мають великий набір доступних операцій і методів, що дає можливість не так детально описувати набір дій, а вказувати

лише те, що має бути зроблено (декларативний стиль).

3. Низькорівневі мови програмування зазвичай мають максимальний ступінь конкретизації, тобто програміст буквально вручну прописує кожну дію, яку повинен виконати комп'ютер. У мовах високого рівня, навпаки, більшість з них виконується компілятором, що з одного боку зручно, але, з іншого боку, компілятор не завжди може робити оптимальні рішення.

Професійний програміст повинен знати, що в мові основне, а що другорядне, похідне. Це робиться з допомогою техніки під назвою деструктуризація (розділення на складові). Сутність, яка не піддається деструктуризації є базовою.

Синтаксис мови програмування вказує на те, як писати код, а семантика вказує на значення коду.

Існують рядки з однаковим синтаксисом, але різною семантикою, і навпаки, існують рядки з однаковою семантикою, але різним синтаксисом.

Синтаксис мови програмування вказує на те як писати код, а семантика вказує на значення коду.

Існують рядки з однаковим синтаксисом та різною семантикою, і навпаки. існують рядки з однаковою семантикою та різним синтаксисом.

Синтаксис це набір правил, які описують, як можна комбінувати символи та слова для формування коректних програмних структур. Наприклад, у більшості мов програмування крапка з комою (;) використовується для вказівки кінця інструкції. Правила синтаксису строго визначають, як повинен виглядати код, але не тлумачать його значення.

Семантика мови програмування визначає значення програмних структур та інструкцій. Семантика вказує на те, що саме відбувається в комп'ютері, коли виконуються інструкції програми. Наприклад, оператор додавання (+) в більшості мов використовується для обчислення суми чисел, але в деяких мовах він може також використовуватися для об'єднання рядків.

Слова синоніми мають однакову семантику.

Що таке компілятор?

Компілятор — це програмне забезпечення, яке використовується для перетворення початкового коду, написаного однією мовою програмування (зазвичай високорівневі мови, такі як C, C++, Java, або Python), у виконуваний код або машинний код, який може бути виконаний на комп'ютері.

Процес компіляції включає такі основні етапи:

1. Парсинг: Компілятор аналізує вихідний код і перетворює його на внутрішню структуру даних, яка представляє програму в більш абстрактній формі.
2. Аналіз: Компілятор проводить аналіз програми для виявлення помилок і недоліків, таких як синтаксичні помилки або помилки типізації.
3. Оптимізація: На цьому етапі компілятор може виконати оптимізацію коду, щоб поліпшити продуктивність програми, зменшивши використання ресурсів і зробити код більш ефективним.
4. Генерація коду: Компілятор генерує виконуваний код або машинний код з аналізованої та оптимізованої програми.
5. Виведення результату: Вихідний файл компілятора містить виконуваний код, який може бути запущений на комп'ютері або іншому обчислювальному пристрої.

Компілятори використовуються для трансляції початкового (вихідного) коду програми у формат, зрозумілий комп'ютеру, що дозволяє виконати програму. Цей процес відрізняється від інтерпретації, де вихідний код програми виконується безпосередньо інтерпретатором без створення окремого виконуваного файлу.

Just-In-Time Compilation (JIT Compilation): Це техніка компіляції, коли програмний код перетворюється в машинний код в процесі виконання програми, а не перед самим запуском. Це дозволяє оптимізувати виконання програми під час її роботи. Компіляція відбувається "на льоту" під час роботи програми, і результатом є машинний код, який може бути безпосередньо виконаний процесором. Переваги JIT-компіляції включають можливість оптимізації виконання програми в залежності від конкретного середовища виконання та характеристик системи. Замість того, щоб використовувати інтерпретатор для виконання вихідного коду нативною мовою (як це робить велика частина мов програмування), JIT-компілятор перетворює вихідний код програми в машинний код або код вищого рівня прямо перед виконанням. Це може значно підвищити швидкодію програми, оскільки нативний машинний код зазвичай виконується швидше, ніж інтерпретований код.

Що таке інтерпретатор?

Інтерпретатор — це програмне забезпечення або компонент, який виконує інтерпретацію (виконання) початкового коду в мові програмування. Він аналізує інструкції вихідного коду і виконує їх безпосередньо, рядок за рядком, без попереднього компілювання у машинний код. Це відрізняє інтерпретатори від компіляторів, які перетворюють вихідний код у машинний код під час компіляції.

Інтерпретатори зазвичай використовуються для мов програмування, таких як Python, Ruby, JavaScript, PHP, і багатьох інших. Вони дозволяють розробникам відразу виконувати свій код, що спрощує тестування і розробку. Проте, через те, що інтерпретатор виконує код одноразово під час виконання програми, він може бути менш ефективним у порівнянні з програмами, що компілюються у машинний код перед виконанням.

Загалом, інтерпретатори корисні для розробки і виконання скриптів та інших коротких програм, а також для інтерактивного виконання коду під час розробки та тестування.

ЛТ-компіляція, з іншого боку, використовує обидві концепції компіляції та інтерпретації. Код програми спочатку інтерпретується, але замість того, щоб виконувати його напряму, ЛТ-компілятор аналізує інструкції програми та генерує оптимізований машинний код для конкретної платформи.

Що таке функціональна мова програмування?

Функціональне програмування (functional programming, FP) — це парадигма програмування, яка базується на використанні функцій як основного будівельного блоку для створення програм.

Основними принципами функціонального програмування є:

1. Незмінність (Immutability): Об'єкти в FP вважаються незмінними, тобто їхні стани не можуть бути змінені після створення. Замість цього, коли потрібно змінити стан об'єкта, створюється новий об'єкт з оновленими значеннями.
2. Чисті функції (Pure Functions): Чисті функції не мають побічних ефектів і повертають результат, що залежить лише від їхніх вхідних параметрів. Вони завжди повертають одинаковий результат для однакових вхідних даних, що робить їх передбачуваними та легкими для тестування.
3. Рекурсія (Recursion): У функціональному програмуванні рекурсія використовується для заміни ітерації. Функції викликають сами себе з новими параметрами для обробки даних.
4. Каррінг (Currying): Каррінг — це процес розбиття функції, яка приймає кілька аргументів, на послідовні функції, які приймають по одному аргументу. Це дозволяє частково застосовувати функції та створювати нові функції шляхом комбінування їх.
5. Функції першого класу: Функції у функціональних мовах є "першого класу", що означає, що вони можуть бути передані як аргументи в інші функції, повернуті як результати функцій і збережені в змінних.

Функціональне програмування підвищує стійкість, спрощує розробку та обслуговування програм, дозволяє паралельне програмування та сприяє оптимізації коду. Багато сучасних мов програмування, такі як Haskell, Lisp, Scala, Clojure, F# та JavaScript, підтримують функціональні концепції та дозволяють розробникам використовувати їх для створення програм.

Каррінг (Каррування, currying) — це техніка функціонального програмування, яка дозволяє перетворити функцію з декількома аргументами в послідовність вкладених функцій, кожна з яких має один аргумент.

Ось приклад реалізації каррінгу на мові JavaScript:

```
// Функція, яку ми хочемо покаррінгувати

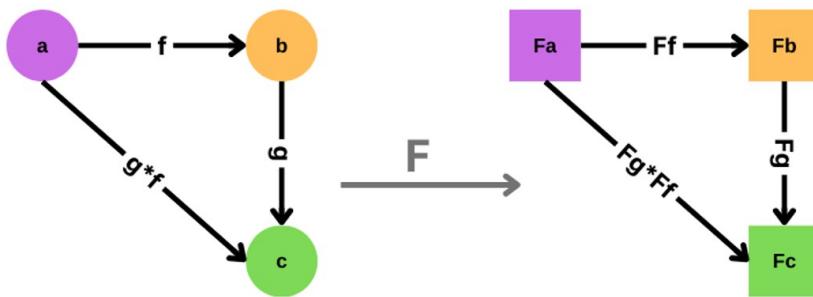
function add(a) {
  return function(b) {
    return a + b;
  };
}

// Використання каррінгу для створення нових функцій

const addFive = add(5);
const addTen = add(10);

console.log(addFive(2)); // Виведе 7 (5 + 2)
console.log(addTen(2)); // Виведе 12 (10 + 2)
```

Функтор



У теорії категорій "категорія" — це математична структура, що складається з об'єктів і морфізмів (візуально стрілок) між ними. Ці морфізми повинні задовольняти двом основним властивостям: композиції, коли ви можете поєднати два морфізми, щоб отримати інший, та ідентичності, де кожен об'єкт має морфізм ідентичності, який діє як нейтральний елемент у композиції. Категорії забезпечують основу для вивчення математичних структур і зв'язків між ними в абстрактний і узагальнений спосіб. Варто зауважити, що слово категорія має інше значення в теорії Арістотеля, або в логіці предикатів, зокрема, в широкому сенсі категорія це множина задана певним предикатом.

Неформально, функтори — це відображення (з однієї категорії в іншу), що зберігають структуру між категоріями. Самуель Ейленберг і Саундерс Мак-Лейн (1945) перші дали абстрактне визначення категорії в контексті теорії категорій. Ви можете створювати категорії, просто з'єднуючи об'єкти стрілками. Ви можете уявити, що починаєте з будь-якого орієнтованого графа та перетворюєте його на категорію, просто додаючи більше стрілок. Спочатку додайте ідентифікаційну стрілку до кожного вузла. Потім для будь-яких двох стрілок, кінець однієї з яких збігається з початком іншої (іншими словами, будь-які дві складені стрілки), додайте нову стрілку, яка буде служити їх композицією. Кожного разу, коли ви додаете нову стрілку, ви також повинні враховувати її композицію з будь-якою іншою стрілкою (окрім ідентифікаційних стрілок) і самою собою.

Функтор — це відображення між категоріями, яке зберігає зв'язки (але не обов'язково ізоморфне). Маючи дві категорії, C і D, функтор F відображає об'єкти в C на об'єкти в D. Якщо a є об'єктом у C, ми запишемо його зображення в D як F a. Але категорія — це не просто об'єкти — це об'єкти та морфізми, які їх поєднують. Функтор також відображає морфізми. Але він не відображає морфізми довільно — він зберігає зв'язки.

Ізоморфізм — це біективне відображення, тобто взаємно однозначне відображення між об'єктами двох структур, яке зберігає всі структурні властивості. Формально, ізоморфізм є біективним морфізмом. Ізоморфізм — морфізм, що може бути обернений; або пара морфізмів, один з яких є зворотним до іншого.

Функтор відображає об'єкти та морфізми між категоріями, зберігаючи їхню структуру, але не обов'язково встановлює взаємно однозначну відповідність між об'єктами. У функторі можуть

виникати ситуації, коли різні об'єкти зображаються на один і той самий об'єкт в іншій категорії, або навпаки.

Функтор дуже схожий на метод (або функцію) map в багатьох мовах програмування. Славнозвісна монада у функціональному програмуванні поєднує функтори та операцію розгортання flat, тому схожа на функцію fmap, або flatMap. Монада це інтерфейс (зокрема в мові Haskell), який дозволяє виконувати декілька операцій map (функтор) та flat підряд, дозволяючи гарантувати безпечну композицію часткових функцій. Метод map() та flat() є загальним (generic).

Часткова функція (на відміну від повної) — це функція, визначена не для всіх можливих вхідних значень.

Ви можете реалізувати монади в JavaScript та багатьох інших мовах програмування (Python, C++, Java). Монади є абстракцією програмування, і конкретний спосіб їх реалізації може змінюватися в залежності від мови програмування.

Реалізація монади “Maybe” мовою JavaScript.

У даному прикладі монада Maybe обробляє можливі значення null або undefined, забезпечуючи коректність операцій.

```
// Монада Maybe
class Maybe {
  constructor(value) {
    this.value = value;
  }

  // Метод застосування функції до значення всередині монади
  map(fn) {
    return this.value !== null && this.value !== undefined
      ? new Maybe(fn(this.value))
      : new Maybe(null);
  }

  // Метод для ланцюжка операцій всередині монади
  chain(fn) {
    return this.value !== null && this.value !== undefined
      ? fn(this.value)
      : new Maybe(null);
  }

  // Метод для отримання значення з монади
  get() {
    return this.value;
  }
}

// Приклад використання Maybe монади
function divideBy2(x) {
  return x / 2;
}

const result = new Maybe(10)
  .map(divideBy2)
  .map(divideBy2)
```

```
.chain((value) => new Maybe(value + 1))  
.get();  
  
console.log(result); // 3.25
```

Що таке об'єктно-орієнтована мова програмування?

Об'єктно-орієнтоване програмування (ООП) — це парадигма програмування, яка базується на концепції об'єктів. В об'єктно-орієнтованому програмуванні програма розбивається на набір об'єктів, які взаємодіють між собою. Кожен об'єкт має свої властивості (атрибути) і методи (функції), які можуть бути викликані для обробки цих об'єктів. ООП спрямоване на моделювання реальних об'єктів і процесів, що відбуваються в програмному коді.

Основні концепції об'єктно-орієнтованого програмування:

1. Polymorphism (поліморфізм): Це здатність об'єктів одного класу виконувати різні дії залежно від контексту. Поліморфізм може бути реалізований за допомогою перевантаження функцій або віртуальних методів.
2. Inheritance (наслідування, успадкування): Це механізм, який дозволяє створювати новий клас на основі існуючого (батьківського) класу. Клас-нащадок успадковує властивості та методи класу-батька і може розширювати їх або перевизначати.
3. Encapsulation (інкапсуляція):
1. Повне обмеження прямого доступу до певних даних й надання можливості працювати з ними тільки через описаний інтерфейс.
2. Приховування даних та реалізації, але опис інтерфейсу для опосередкованого доступу до них.
Інкапсуляція може здійснюватися з допомогою closure, класів та об'єктів, модулів, структур (struct) та вказівників (pointers).
4. Abstraction (абстракція): Абстракція полягає в ізоляції основних характеристик об'єкта, відокремленні від звичних деталей. Це дозволяє створювати класи, які представляють сутність об'єкта, і використовувати ці класи без необхідності знати всі деталі реалізації.
Уявіть, що ви користуєтесь автомобілем. Вам не потрібно знати всі технічні деталі щодо роботи двигуна чи системи гальмування, щоб керувати автомобілем. Ви можете просто використовувати руль, педалі газу та гальма для керування автомобілем. У цьому випадку інтерфейс керування автомобілем надає абстракцію від його реалізації.

Розглянемо приклади параметричного поліморфізму та ad hoc поліморфізму в мові C#.

Параметричний поліморфізм (Generics). У цьому прикладі клас Box має параметризований тип T, що дозволяє йому працювати з будь-яким типом даних.

“Generic programming” дозволяє записувати алгоритми таким чином, щоб пізніше можна було вказати типи. Щоб зробити функцію загальною, ви додаєте параметр типу, укладений у кутові дужки (<>) одразу після імені функції.

```
using System;
using System.Collections.Generic;

public class Box<T>
{
    private T content;

    public Box(T content)
    {
```

```

        this.content = content;
    }

    public T GetContent()
    {
        return content;
    }
}

public class Program
{
    public static void Main()
    {
        Box<int> intBox = new Box<int>(42);
        Box<string> stringBox = new Box<string>("Hello, Generics!");

        Console.WriteLine(intBox.GetContent()); // Виведе 42
        Console.WriteLine(stringBox.GetContent()); // Виведе "Hello, Generics!"
    }
}

```

Приклад ад хос поліморфізму в C# через перевантаження оператора:

```

using System;

public class Calculator
{
    public static int Add(int a, int b)
    {
        return a + b;
    }

    public static double Add(double a, double b)
    {
        return a + b;
    }
}

public class Program
{
    public static void Main()
    {
        int sumInt = Calculator.Add(2, 3);
        double sumDouble = Calculator.Add(2.5, 3.7);

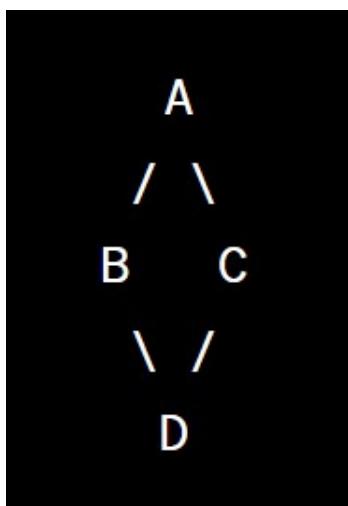
        Console.WriteLine($"Sum of integers: {sumInt}"); // Виведе "Sum of integers: 5"
        Console.WriteLine($"Sum of doubles: {sumDouble}"); // Виведе "Sum of doubles: 6.2"
    }
}

```

Multiple Inheritance (Множинне успадкування): Це можливість класу успадковувати властивості та методи від декількох батьківських класів. Деякі мови дозволяють множинне успадкування, але це може призводити до складнощів, таких як "Diamond problem".

Diamond Problem (проблема ромба): Це конфлікт, який виникає при спробі успадкування властивостей і методів через більше ніж один шлях у ієрархії спадкування. Це може призводити до непослідовності у викликах методів.

Проблема ромба (іноді відома як алмазна проблема або "Diamond Problem") — це конфлікт, який виникає в об'єктно-орієнтованому програмуванні, коли один клас успадковує властивості або методи від двох класів, які успадковують від одного і того ж батьківського класу. Це призводить до непередбачуваної поведінки та конфліктів у програмі. По суті, це виникає тоді, коли ієрархія класів має наступну структуру:



У цій ієрархії клас D успадковує від як мінімум двох батьківських класів B і C, і обидва ці класи успадковують від класу A. Така ситуація може виникнути в деяких мовах програмування, які підтримують множинне успадкування.

Основні проблеми, пов'язані з проблемою ромба, включають:

Конфлікти імен: Якщо класи B і C мають методи або властивості з одинаковими іменами, то клас D не може однозначно визначити, які саме методи або властивості використовувати.

Непередбачувана поведінка: Визначення, які методи викликаються або які властивості використовуються в класі D, може залежати від порядку успадкування і реалізації класів B і C. Це робить програму менш передбачуваною.

Складна підтримка та налагодження коду: Розуміння та налагодження коду, який стикається з проблемою ромба, може бути складним завданням, оскільки потрібно враховувати багато можливих варіантів спадкування та взаємодії класів.

Деякі мови програмування (зокрема C++) розв'язують проблему ромба, надаючи спеціальні правила для вирішення конфліктів, наприклад, за допомогою абстрактних класів та вказівників. Інші мови, як от Java, дозволяють спадкування лише від одного класу (немає множинного успадкування), щоб уникнути цієї проблеми.

SOLID — це абревіатура, що представляє собою п'ять основних принципів об'єктно-орієнтованого дизайну. Кожна літера представляє один із цих принципів:

S - Принцип єдиної відповідальності (Single Responsibility Principle).

O - Принцип відкритості/закритості (Open/Closed Principle).

L - Принцип підстановки Барбари Лісков (Liskov Substitution Principle).

I - Принцип розділення інтерфейсів (Interface Segregation Principle).

D - Принцип інверсії залежностей (Dependency Inversion Principle).

Ці принципи спрямовані на створення коду, який є масштабованим, легким у зміні та підтримці, і відповідає принципам чистого об'єктно-орієнтованого дизайну.

У своїй статті Барбара Лісков сформулювала свій принцип так:

“Нехай $q(x)$ є властивістю правильною для об'єктів x деякого типу T . Тоді $q(y)$ також має бути правильною для об'єктів у типу S , де S — підтип типу T ”.

Наприклад функція, яка приймає об'єкт Прямокутник, повинна також приймати об'єкт Квадрат, бо він також прямокутник.

Дотримання принципу підстановки Лісков гарантує, що дочірній клас залишається підмножиною батьківського класу, як зображенено на картинці.



Принципи SOLID описує Роберт Мартін у своїй книзі “Чиста архітектура”.

“Open/Closed Principle” описаний в книзі Бертрана Месра "Object-Oriented Software Construction" (1988). Принцип відкритості/закритості означає, що “програмні сутності, такі як класи, модулі, функції, методи та ін. мають бути відкритими для розширення та закритими для змін”.

GRASP і GoF — це два набори патернів (шаблонів) до проектування програмного забезпечення, які надають рекомендації та патерни для розподілу відповідальностей та структуризації коду.

GoF (Gang of Four):

GoF вказує на четвірку авторів книги "Design Patterns: Elements of Reusable Object-Oriented Software" (Патерни проектування: Елементи повторного використання об'єктно-орієнтованого програмного забезпечення). Ці чотири автори — Еріх Гамма, Річард Хелм, Ральф Джонсон і Джон Вліссідес — визначили 23 різні патерни проектування, які можна використовувати для розв'язання типових проблем у програмуванні.

Деякі GoF шаблони: Prototype pattern, Singleton pattern, Factory method pattern, Abstract factory pattern, Decorator pattern, Proxy pattern, Iterator pattern, Observer pattern, Chain-of-responsibility pattern.

GRASP (General Responsibility Assignment Software Patterns):

GRASP — це набір принципів та патернів, які допомагають визначити, як об'єкти повинні взаємодіяти між собою та розподіляти відповідальності в системі. Ці принципи допомагають проектувальникам приймати рішення щодо призначення обов'язків об'єктам в системі, щоб забезпечити більш гнучку та обслуговувану архітектуру. За своєю суттю, цей набір патернів більш абстрактний, ніж загально відомий каталог шаблонів від "Банди чотирьох" (GoF-шаблони).

Ось простий приклад об'єктно-орієнтованого програмування на TypeScript. У цьому прикладі ми створимо просту ієархію класів для тварин, використовуючи наслідування, та інтерфейс для реалізації специфічної функціональності.

```
// Інтерфейс для опису функціональності тварини
interface Animal {
    name: string;
    makeSound(): void;
}

// Базовий клас тварини
class BaseAnimal implements Animal {
    constructor(public name: string) { }

    makeSound(): void {
        console.log(`#${this.name} видає деякий звук.`);
    }
}

// Похідний клас для собаки
class Dog extends BaseAnimal {
    constructor(name: string) {
        super(name);
    }

    // Перевизначаємо метод makeSound для собаки
    makeSound(): void {
        console.log(`#${this.name} гавкає.`);
    }
}
```

```

// Новий метод для собаки
fetch(): void {
    console.log(`\$ {this.name} принесла паличку.`);
}

// Похідний клас для кота
class Cat extends BaseAnimal {
    constructor(name: string) {
        super(name);
    }

    // Перевизначаємо метод makeSound для кота
    makeSound(): void {
        console.log(`\$ {this.name} муркоче.`);
    }

    // Новий метод для кота
    climbTree(): void {
        console.log(`\$ {this.name} лазить по дереву.`);
    }
}

// Створюємо екземпляри тварин
const myDog = new Dog("Барни");
const myCat = new Cat("Вася");

// Викликаємо методи та властивості тварин
myDog.makeSound();
myDog.fetch();

myCat.makeSound();
myCat.climbTree();

```

Access modifiers (модифікатори доступу) в програмуванні визначають область видимості (accessibility) для методів, змінних чи інших елементів програми. Вони контролюють, як і звідки можна отримати доступ до певних елементів коду. Це допомагає управляти рівнями доступу та забезпечити правильну і безпечно взаємодію між різними частинами програми.

Існують звичайні модифікатори доступу, такі як:

Public (Публічний): Елементи, які мають цей модифікатор, доступні з будь-якої частини програми. Наприклад, метод чи змінна може бути позначена як `public`.

Private (Приватний): Елементи з цим модифікатором доступу можуть бути використані лише в межах самого класу, в якому вони оголошенні. Інші частини програми не можуть звертатися до приватних елементів. Це допомагає забезпечити ізоляцію та безпеку класу.

Protected (Захищений): Елементи з цим модифікатором доступу можуть бути використані в межах класу, а також в класах-нащадках (наслідування). Це дозволяє спадкоємцям мати доступ до певних частин базового класу.

Модифікатори доступу в мові TypeScript

```
// Приклад класу з різними модифікаторами доступу

// Приклад 1: Public (Публічний)
class Car {
    public brand: string;

    constructor(brand: string) {
        this.brand = brand;
    }

    public startEngine(): void {
        console.log(`Starting the engine of ${this.brand}`);
    }
}

// Приклад 2: Private (Приватний)
class BankAccount {
    private balance: number;

    constructor(initialBalance: number) {
        this.balance = initialBalance;
    }

    private displayBalance(): void {
        console.log(`Current balance: ${this.balance}`);
    }

    public deposit(amount: number): void {
        this.balance += amount;
        this.displayBalance();
    }

    public withdraw(amount: number): void {
        if (amount <= this.balance) {
            this.balance -= amount;
            this.displayBalance();
        } else {
            console.log("Insufficient funds");
        }
    }
}

// Приклад 3: Protected (Захищений)
class Animal {
    protected name: string;

    constructor(name: string) {
        this.name = name;
    }

    protected makeSound(): void {
        console.log(`${this.name} makes a sound`);
    }
}
```

```

}

class Dog extends Animal {
    private breed: string;

    constructor(name: string, breed: string) {
        super(name);
        this.breed = breed;
    }

    public displayDetails(): void {
        console.log(`Name: ${this.name}, Breed: ${this.breed}`);
        this.makeSound(); // Доступ до захищеного методу з батьківського класу
    }
}

// Приклад використання класів

const myCar = new Car("Toyota");
myCar.startEngine();
console.log(`Car brand: ${myCar.brand}`);

const myAccount = new BankAccount(1000);
myAccount.deposit(500);
myAccount.withdraw(200);

const myDog = new Dog("Buddy", "Labrador");
myDog.displayDetails();

```

Ось реалізація шаблону Singleton з допомогою модифікаторів доступу.

Шаблон Singleton (Одинак) — це один з патернів проектування, який використовується для забезпечення того, щоб клас мав лише один екземпляр і надає глобальну точку доступу до цього екземпляра.

У TypeScript патерн Singleton можна реалізувати, використовуючи клас і статичний метод для отримання єдиного екземпляра класу. Ось приклад реалізації патерну Singleton на TypeScript:

```

class Singleton {
    private static instance: Singleton | null = null;

    private constructor() {
        // Приватний конструктор
    }

    public static getInstance(): Singleton {
        if (!Singleton.instance) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }
}

```

```
}

public someMethod(): void {
    console.log("Виклик методу в єдиному екземплярі Singleton");
}

// Використання Singleton
const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

console.log(instance1 === instance2); // true, оскільки це є один і той самий екземпляр

instance1.someMethod(); // Виклик методу в єдиному екземплярі Singleton
instance2.someMethod(); // Виклик методу в єдиному екземплярі Singleton
```

Що таке шаблон Спостерігач (Observer)?

"Спостерігач" (Observer) — це патерн проєктування, що належить до категорії поведінкових патернів. Його основна ідея полягає в тому, щоб визначити залежність один до одного об'єктів таким чином, щоб при зміні стану одного об'єкта всі йому залежні об'єкти автоматично отримували сповіщення та оновлювали свій стан.

Ось приклад реалізації шаблону Observer мовою TypeScript:

```
// Інтерфейс, який визначає метод оновлення для спостерігача
interface Observer {
    update(data: any): void;
}

// Клас, який представляє спостерігача
class ConcreteObserver implements Observer {
    update(data: any): void {
        console.log(`Отримано оновлення: ${data}`);
    }
}

// Клас, який представляє спостережуваний об'єкт
class Subject {
    private observers: Observer[] = [];

    // Додає спостерігача до списку
    addObserver(observer: Observer): void {
        this.observers.push(observer);
    }

    // Видаляє спостерігача зі списку
    removeObserver(observer: Observer): void {
        const index = this.observers.indexOf(observer);
        if (index !== -1) {
            this.observers.splice(index, 1);
        }
    }

    // Повідомляє всіх спостерігачів про зміни
    notify(data: any): void {
        for (const observer of this.observers) {
            observer.update(data);
        }
    }
}

// Приклад використання
const subject = new Subject();

const observer1 = new ConcreteObserver();
const observer2 = new ConcreteObserver();
```

```
subject.addObserver(observer1);
subject.addObserver(observer2);

subject.notify("Нові дані 1"); // Виводиться: "Отримано оновлення: Нові дані 1"

subject.removeObserver(observer1);

subject.notify("Нові дані 2"); // Виводиться: "Отримано оновлення: Нові дані 2"
```

Що таке шаблон Декоратор?

Шаблон Декоратор (Decorator) — це структурний шаблон проєктування, який дозволяє додавати нову функціональність до існуючих об'єктів динамічно, без зміни їхньої структури. Цей шаблон базується на принципі композиції, де класи обгортають один одного.

Опис шаблону мовою TypeScript:

```
// Інтерфейс, який визначає основну функціональність
interface Component {
    operation(): void;
}

// Конкретна реалізація основного компонента
class ConcreteComponent implements Component {
    operation(): void {
        console.log('Виконання основного компонента.');
    }
}

// Абстрактний клас декоратора, який реалізує інтерфейс Component і містить посилання на компонент
abstract class Decorator implements Component {
    protected component: Component;

    constructor(component: Component) {
        this.component = component;
    }

    operation(): void {
        this.component.operation();
    }
}
```

```
// Конкретний декоратор, який додає додаткову функціональність до компонента
class ConcreteDecorator extends Decorator {
    constructor(component: Component) {
        super(component);
    }

    operation(): void {
        super.operation();
        this.additionalOperation();
    }

    private additionalOperation(): void {
        console.log('Виконання додаткової операції.');
    }
}

// Приклад використання
const component: Component = new ConcreteComponent();
const decoratedComponent: Component = new ConcreteDecorator(component);

// Виклик операції через декорований об'єкт
decoratedComponent.operation();
```

Що таке система типів?

Система типів (або система типізації) - це концепція в програмуванні, яка визначає, які види даних можуть бути використані в конкретній мові програмування і як ці дані можуть взаємодіяти один з одним. Система типів допомагає визначити, які операції можна виконувати з різними видами даних і які перевірки та обмеження повинні бути застосовані до цих операцій.

Тип даних — це множина доступних значень для певного поля чи змінної. Булевий тип містить два значення Boolean = {true, false}.

Основні операції (функції) над масивами (arrays) в принципі не мають типу, бо щоб виявити тип вам вже потрібна якась функція чи множина. Тому навіть в строго типізованих мовах базові операція універсальні. Основні операції над масивами в сутності не мають типу, оскільки вони можуть бути застосовані до масивів будь-якого типу даних. Однак тип масиву визначається типом його елементів, і це впливає на можливі операції, які можна виконати з масивом.

Сувора (сильна, strong) і слабка (weak) типізація, а також статична (static) і динамічна (dynamic) типізація — це основні характеристики систем типів в мовах програмування. Розглянемо кожну з цих характеристик окремо:

1. Сувора (сильна) типізація:

- В суворій системі типів, типи даних дуже важливі та строго контролюються.
- Операції над даними вимагають, щоб типи були сумісними, і компілятор (або інтерпретатор) гарантує, що ці типи відповідають очікуванням.
- Наприклад, в суворій системі типів неможливо додавати рядки до чисел без явного перетворення типів.

Сувора типізація досягається шляхом статичної типізації, виводом типів (type inference), та обмеженням на операції зі змінними різних типів.

2. Слабка типізація:

- В слабкій системі типів типи даних менш суворо контролюються, і деякі операції можуть виконуватися навіть над несумісними типами.
- Це може привести до непередбачуваної поведінки та помилок під час виконання програми.
- Наприклад, у слабкій системі типів можливо додавати рядки до чисел без явного перетворення типів, і це може привести до неправильних результатів.

Зазвичай в мові зі слабкою типізацією програміст використовує різні перевірки типів в ході виконання програми, використовуючи оператори typeof та instanceof.

3. Статична типізація:

- У статичній системі типів, типи даних визначаються на етапі компіляції, і компілятор вимагає, щоб типи були вірними під час компіляції програми.
- Перевірка типів відбувається на етапі компіляції, і це допомагає виявити багато помилок ще до виконання програми.

4. Динамічна типізація:

- У динамічній системі типів типи даних визначаються на етапі виконання програми, і операції можуть бути виконані навіть над змінними з різними типами даних.

- Перевірка типів відбувається під час виконання програми, і це дозволяє програмі бути більш гнучкою, але може також привести до помилок під час виконання.

Вибір між суворою/слабкою і статичною/динамічною типізацією залежить від конкретної мови програмування та вимог вашого проекту. Різні мови мають різні комбінації цих характеристик, і вони впливають на розробку програм та безпеку виконання коду.

Сувора динамічна типізація — це характеристика системи типів в програмуванні, яка поєднує в собі обидві сторони типізації: динамічну та сувору. Це означає, що типи даних визначаються під час виконання програми (динамічна типізація), але перевірка типів виконується строго, і допустимі операції обмежені типами даних (сувора типізація).

Основні риси суворої динамічної типізації включають:

1. Визначення типів даних під час виконання: Тобто, програма може динамічно визначити типи даних змінних або об'єктів під час виконання, зазвичай на основі значень, які вони містять.
2. Перевірка типів даних: Навіть при динамічному визначенні типів даних сувора динамічна типізація обмежує операції, які можуть бути виконані лише над змінними або об'єктами певних типів. Якщо спробувати виконати недопустиму операцію, програма зазвичай видасть помилку.

Сувора динамічна типізація часто використовується в деяких динамічних мовах програмування, таких як Python, Ruby, а також у системах типізації з інтерпретаторами, які виконують перевірку типів даних під час виконання. Вона надає більшу гнучкість, у порівнянні з суворою статичною типізацією, але при цьому забезпечує безпеку виконання коду через перевірку типів під час виконання.

Сувора типізація з виводом типів (Strong typing with type inference) — це підхід до системи типізації в програмуванні, який поєднує в собі обов'язкову декларацію типів з можливістю автоматичного виводу типів для деяких конструкцій коду.

В контексті виводу типів (type inference), певні типи можуть бути виведені автоматично, не потрібно явно вказувати їх у коді. Наприклад, якщо ви присвоюєте числове значення змінній, компілятор може автоматично визначити, що ця змінна має тип цілого числа.

Багато статично типізованих мов програмування, таких як Java чи C#, використовують номінальну перевірку типів, тоді як деякі динамічно типізовані мови, такі як Python чи JavaScript, можуть використовувати структурну перевірку типів.

Номінальна перевірка типів орієнтується на імена типів (наприклад, "int", "string", клас). Важливо, щоб типи мали однакові імена або були явно визначені програмістом.

Структурна перевірка типів орієнтується на структуру даних.

Типи вважаються однаковими, якщо вони мають однакову структуру, незалежно від їхніх імен.

Якщо ви створюєте об'єкт без вказання його класу, то це означає, що компілятор не буде явно знати тип об'єкта і тому під час порівняння використовуватиме структурну типізацію.

Качина типізація (Duck typing) схожа на структурну типізацію, але відрізняється від неї. Структурна типізація — це статична система типізації, яка визначає сумісність та еквівалентність типів за структурою типу, тоді як качина типізація є динамічною та визначає сумісність типів лише тієї частини структури типу, до якої здійснюється доступ під час виконання.

Назва терміна походить від "качиного тесту", який звучить так: "Якщо воно виглядає як качка, плаває як качка і крякає як качка, то це напевно і є качка" (Але це може бути модель качки, а не реальна качка, тому потрібно бути уважним).

Що таке TypeScript?

TypeScript — це розширення мови програмування JavaScript, яке надає статичну типізацію для JavaScript-коду.

Ось опис термінів та понять, які пов'язані з TypeScript:

1. Функції першого класу (First-Class Functions): TypeScript, подібно до JavaScript, має функції, які є об'єктами першого класу. Це означає, що функції можна передавати як аргументи, повертати з функцій і зберігати у змінних.
2. Замикання (Closure): TypeScript підтримує замикання — це властивість функцій зберігати доступ до змінних, які були створені під час їхнього створення.
3. Область видимості (Scope): TypeScript використовує області видимості для обмеження доступу до змінних та функцій.
4. Каррінг (Currying): Це техніка функціонального програмування, коли функція приймає декілька аргументів і перетворюється на послідовність функцій, кожна з яких приймає лише один аргумент.
5. Динамічна система типів (Dynamic Type System): JavaScript і TypeScript використовують динамічну систему типів, де тип змінної визначається під час виконання програми.
6. Боксинг (Boxing): TypeScript дозволяє автоматично упаковувати примітивні значення в об'єкти (бокси) при необхідності для взаємодії зі структурами даних, які очікують об'єкти.
7. Інструкція та вираз (Statement and Expression): TypeScript розрізняє між інструкціями (statements) і виразами (expressions). Вираз — це частина коду, яка повертає значення. Інструкція описує логіку програми.
8. Змінні (Variables): TypeScript використовує ключове слово `var`, `let`, та `const` для оголошення змінних. `let` та `const` були додані в ECMAScript 6 і надають більший контроль над областями видимості та імутабельністю відповідно.
9. Підняття (Hoisting): Hoisting (підняття) — це механізм, який пересуває оголошення змінних та функцій вгору в контексті їхнього області видимості перед тим, як код буде виконаний. Це означає, що ви можете викликати функцію або отримати доступ до змінної до їхнього оголошення (Це не означає, що в ній буде значення. Значення запишеться під час оголошення змінної).

Узагальнюючи, TypeScript — це мова програмування, яка додає статичну типізацію до JavaScript, що допомагає зменшити помилки під час розробки та підвищити читабельність та підтримку коду. Вона також підтримує багато з функцій і понять, які специфічні для функціонального програмування і JavaScript.

Ось приклад TypeScript-коду, який демонструє вищеописані поняття:

```
// Приклад функції першого класу та замикання
function makeMultiplier(factor: number): (x: number) => number {
    return function (x: number) {
        return x * factor;
    };
}

const double = makeMultiplier(2);
const triple = makeMultiplier(3);

console.log(double(5)); // Виведе 10
console.log(triple(5)); // Виведе 15

// Приклад області видимості та змикання
function outerFunction() {
    const outerVariable = 'Зовнішня змінна';

    return function innerFunction() {
        console.log(outerVariable);
    };
}

const inner = outerFunction();
inner(); // Виведе "Зовнішня змінна"

// Приклад каррінгу
function add(x: number) {
    return function (y: number) {
        return x + y;
    };
}

const add5 = add(5);
console.log(add5(3)); // Виведе 8

// Приклад динамічної системи типів
function addNumbers(a: any, b: any): any {
    return a + b;
}

console.log(addNumbers(5, 3)); // Виведе 8
console.log(addNumbers('5', '3')); // Виведе "53"

// Приклад боксингу
const primitiveValue: number = 42;

console.log(primitiveValue.toFixed(2)); // Працює

// Приклад вказівок (statement) та виразів (expression)
let statementExample: number;
if (true) {
```

```
statementExample = 10;  
}  
  
const expressionExample = true ? 10 : 20;  
  
console.log(statementExample); // Виведе 10  
console.log(expressionExample); // Виведе 10
```

Синтаксис TypeScript:

```
// Декларація змінних з вказанням типів
let myNumber: number = 10;
let myString: string = "Hello, TypeScript";
let myBoolean: boolean = true;

// Типи масивів
let myArray: number[] = [1, 2, 3, 4, 5];
let myStringArray: string[] = ["apple", "banana", "orange"];

// Типи об'єктів
let myObject: { key: string, value: number } = { key: "example", value: 42 };

// Функції з вказанням типів для параметрів та поверненого значення
function addNumbers(a: number, b: number): number {
    return a + b;
}

// Опціональні та залишкові параметри у функціях
function greet(name: string, greeting?: string): string {
    return greeting ? `${greeting}, ${name}!` : `Hello, ${name}!`;
}

// Інтерфейси для створення власних типів
interface Person {
    name: string;
    age: number;
}

let person: Person = { name: "John", age: 25 };

// Зв'язування типів
type Point = { x: number, y: number };

function calculateDistance(point1: Point, point2: Point): number {
    const deltaX = point2.x - point1.x;
    const deltaY = point2.y - point1.y;
    return Math.sqrt(deltaX ** 2 + deltaY ** 2);
}

// Джenerіки для створення загальнозастосовних функцій та класів
function identity<T>(arg: T): T {
    return arg;
}

class Container<T> {
    private value: T;

    constructor(value: T) {
        this.value = value;
    }

    getValue(): T {
```

```

        return this.value;
    }
}

// Enums (переліки)
enum Color {
    Red,
    Green,
    Blue,
}
let selectedColor: Color = Color.Green;

// Типи для обробки нульових та невизначених значень
let nullableNumber: number | null = null;
let undefinedString: string | undefined = undefined;

// Типи для об'єднань та перетинів
type Animal = { name: string };
type Bird = { fly: () => void };

let bird: Animal & Bird = { name: "Eagle", fly: () => console.log("Flying") };

// Типи для збору та обробки подій
type EventHandler = (event: Event) => void;

class EventManager {
    private handlers: EventHandler[] = [];

    addHandler(handler: EventHandler): void {
        this.handlers.push(handler);
    }

    triggerEvent(event: Event): void {
        this.handlers.forEach(handler => handler(event));
    }
}

// Автоматичне виведення типів (Type Inference)
let inferredNumber = 42;
let inferredString = "TypeScript is awesome";

```

Мова TypeScript є розширенням мови JavaScript, яке дає можливість структурної перевірки типів, зокрема, шляхом такої сутності як interface (fully abstract class).

Істинна філософія JavaScript: "Об'єкт -> клас, а не клас -> об'єкт".

(Клас у сенсі об'єктно-орієнтованого програмування, тобто структура, яка генерує об'єкти по шаблону. Шаблон задає клас, в сенсі, множину об'єктів, які відповідають шаблону).

В JavaScript: "Об'єкт -> клас -> об'єкт".

Це означає, що перевірка типів в JS може бути тільки структурна, а не номінальна. Щоб була номінальна перевірка типів кожен об'єкт повинен створюватися через клас, ніяких літеральних об'єктів.

В JS клас це похідна структура.

Ми можемо сконструювати клас з об'єкта прототипа та функції конструктора. В принципі, typeof ClassA === "function". Тобто, об'єкт в JS примітивніша сутність ніж клас.

Вперше класи були реалізовані в мові Симула, в якій ви не могли створити об'єкт не створюючи спершу клас.

Ключові особливості JS:

First-class functions,
Closure,
Scope,
Object literals,
Implicit boxing,

JavaScript використовує прототипне спадкування.

Прототип — твірний шаблон проєктування, який дозволяє створювати копії існуючих об'єктів таким чином, що програмний код не залежить від їх класів.

JavaScript (JS) — інтерпретована (або ЛТ) скриптована мова з слабкою системою типів й динамічною типізацією.

Стандартом для JavaScript є ECMAScript Language Specification (ECMA-262) та ECMAScript Internationalization API specification (ECMA-402).

Структура JavaScript:

1. statement and expression in JS.
2. const, var, let (hoisting, temporary dead zone, immutability).
3. data type (Primitive types: number, bigint, string, boolean, null, undefined, symbol; Reference types: object, array*, function).
4. typeof, in, instanceof.
5. arithmetic operations (+,-,*,/,%). Math object, Math.random();
6. logic operations (&&, ||, !).
7. Conditional statement (if, switch), ternary operator.
8. Loops (for, while, do-while, forEach, map, for-of, for-in, ...).
9. function declaration and function expression, arrow function, IIFE, anonymous function.
10. constructor function, this, .prototype, .__proto__;
11. class, super(), extends, this, getter, setter, instanceof.
12. Promises, async, await, timers, event loop, micro and macro tasks.
13. Generators, new Proxy.
14. new Date, RegExp, JSON.stringify.

Інструкція (Statement) — це конструкція, яка виконує певний блок коду. Зазвичай інструкції (вказівки) використовуються для визначення поведінки програми, такої як умовні конструкції (if, else), цикли (for, while), визначення функцій та інше.

Вираз (Expression) — це будь-яка конструкція, яка повертає значення. Вираз може бути чимось простим, таким як змінна чи константа, або складнішим, як арифметичний вираз або виклик функції. Вираз (Expression) — це структура коду на місце якої в програмі можна підставити значення й програма при цьому буде працювати.

Ескіз керівництва по стилю JavaScript й TypeScript кода:

1. Використовуємо константи, якщо не має сильної потреби використати, щось інше.
2. Пишемо чисті функції, але якщо потрібно все ж написати побічний ефект, тоді обгортаемо все в об'єкт. Робимо метод й зміну, яка буде містити побічний ефект.
3. Стандартизуємо формат назв змінних, функцій, класів та інших сутностей (camelCase, snake_case, Hungarian notation).
4. Типізуємо змінні.
5. Використовуємо сувере порівняння ===.
6. Дотримуйтесь принципів DRY, KISS, YAGNI.
7. Використовуємо декларативні методи, якщо відсутня необхідність в іншому. Наприклад, forEach замість for.

Змикання (closure) — це концепція в програмуванні, яка означає, що функція, яка була створена в одному контексті (зазвичай в іншій функції), зберігає доступ до змінних та об'єктів цього контексту, навіть після того, як цей контекст вийшов з області видимості. Іншими словами, змикання запам'ятовує значення змінних, доступ до яких воно мало на момент свого створення, і використовує ці значення навіть після того, як зовнішня функція завершила свою роботу.

Це дозволяє створювати більш гнучкі та потужні функції, особливо в контексті з асинхронним програмуванням або подіями. Ідея замикань є однією з основних концепцій у багатьох мовах програмування, таких як JavaScript, TypeScript, Python, та інші.

Що таке Цикл подій в JavaScript?

Віртуальна машина V8, яка виконує JavaScript програму, має Event loop (цикл подій), який контролює послідовність виконання команд мови.

Мова TypeScript також розрахована на роботу з Event loop (цикл подій).

Event loop (цикл подій) в популярних віртуальних машинах для мови JavaScript та її розширень (TypeScript) створений для того, щоб давати змогу виконувати певні команди асинхронно.

Цикл подій (Event loop) має 3 ключові поняття:

1. Стек викликів (call stack).
2. Черга задач (або черга макрозадач).
3. Черга мікрозадач (microtask queue).

Початковий код виконується послідовно. Деякі методи мови призначені для реєстрації макrozадачі, або мікрозадачі, тобто вони додають задачу у відповідну чергу.

Функції обробляються в стеку викликів.

Інтерпретатор послідовно виконує команди мови, коли він зустрічає функцію він передає її в стек викликів.

Коли десь в коді зустрічається метод, який реєструє асинхронний код, тоді додається задача в чергу макrozадач, або мікрозадач, залежно від типу.

Опрацювання черг працює наступним чином:

Цикл подій бере задачу з черги макrozадач і починає її виконувати (також залучаючи стек викликів). Перед тим, як перейди до наступної групи задач в черзі макrozадач, цикл подій повинен опрацювати всю чергу мікрозадач.

Спочатку ми опрацьовуємо синхронний код, потім всі мікрозадачі, а потім переходимо до наступної групи задач в черзі макrozадач.

Кожна черга являє собою масив, який складається з задач, які представляють групи команд.

Ось приклад порядку виконання задач в Event loop на мові JavaScript:

```
const macrotaskQueue = [()=>{console.log("initial code");}];
const microtaskQueue = [];

macrotaskQueue.push(() => {
  console.log(1);
});
macrotaskQueue.push(() => {
  console.log(2);
});
microtaskQueue.push(() => {
  console.log(3);
});
microtaskQueue.push(() => {
  console.log(4);
});
```

```

const processQueue = (queue, callback) => {
  while (queue.length > 0) {
    const task = queue.shift();

    task();

    if (callback) {
      callback();
    }
  };
};

processQueue(macrotaskQueue, () => {
  processQueue(microtaskQueue);
});

```

Є декілька способів створення асинхронних функцій в JavaScript:

```

//setTimeout додасть нову задачу в чергу макрозадач після певної затримки
setTimeout(function() {
  console.log("Task completed");
}, 1000);

// Promise створить нову мікрозадачу

let myPromise = new Promise(function(resolve, reject) {

  let isSuccess = true;

  if (isSuccess) {
    resolve("Успіх!");
  } else {
    reject("Помилка!");
  }
});

myPromise.then(function(successMessage) {
  console.log(successMessage);
}).catch(function(errorMessage) {
  console.log(errorMessage);
});

```

Що таке граматика?

Граматика — це система правил, які визначають, як буде побудована мова, як вона буде правильно вживатися і як розрізнятимуться між собою окремі слова та фрази. Граматика визначає структуру мови, синтаксис, правила словоутворення, правила сполучення слів у речення та інші аспекти мовного виразу. Вивчення граматики є важливою частиною навчання будь-якої мови, оскільки правильне використання граматичних правил допомагає зрозуміти та бути зрозумілим у мовленні і письмі. Граматика також містить правила вживання частин мови, такі як іменники, прикметники, дієслова та інші граматичні категорії.

Слово мови — це набір символів, який визначений в словнику цієї мови відповідно до граматики. Слово — це послідовність символів, об'єднаних за граматичними правилами певної мови та співвідносних з певним елементом позамовної реальності (іменники, прикметники, дієслова, прислівники, числівники).

Граматика мови L — це скінчений набір правил G, що дозволяє рекурсивно генерувати множину P всіх допустимих (значущих) виразів (рядків) L.

Приклад.

Визначимо G: якщо $q \in \{\text{"A"}, \text{"B"}\}$ і $r \in \{\text{"a"}, \text{"b"}\}$, то $q \in P$, $r \in P$, $q \times r \in P$;

Отже, $P = \{\text{"A"}, \text{"B"}, \text{"a"}, \text{"b"}, \text{"Aa"}, \text{"Ab"}, \text{"Ba"}, \text{"Bb"}\}$.

Мова L складається з множин G, P і позначається як $L(G, P)$.

x — декартів добуток.

У теорії множин, декартів добуток (прямий добуток) $X \times Y$ двох множин X та Y — це множина усіх можливих впорядкованих пар, у яких перший компонент належить множині X, а другий — множині Y. Це поняття названо на честь відомого французького математика Рене Декарта, який не застосував теорію множин, але в аналітичній геометрії використовував впорядковані пари точок, одна по x осі, друга по y осі.

У інформатиці форма Бекуса-Наура — це метасинтаксичне позначення для контекстно-вільних граматик, яке часто використовується для опису синтаксису мов, що використовуються в обчислювальній роботі, таких як мови комп'ютерного програмування, формати документів, набори інструкцій та протоколи зв'язку. Джін Саммет у своїй книзі "Мови програмування: історія й основи" (1969) активно використовувала форму Бекуса-Наура.

Приклад рекурсивного позначення у формі Бекуса-Наура:

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{number} \rangle;$

Сенс цього позначення полягає у визначенні класу об'єктів, які називаються "ідентифікаторами" (identifier). Спочатку стверджується, що кожен елемент раніше визначеного класу $\langle \text{letter} \rangle$ є ідентифікатором, потім наводяться правила генерації, які полягають у тому, що присвоєння ідентифікатору літери або цифри праворуч знову дає ідентифікатор. Тобто в наведеному вище випадку об'єкт є ідентифікатором, якщо це "літера", "літера + літера", "літера + літера + цифра", або "літера + цифра", або "літера + цифра + цифра". (A, AB, A1, A1B, A1B2, ..., але не 1, 1A, ...).

Приклад налаштування синтаксису для операції додавання (натуруальних чисел):

$\langle \text{number} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;$

$\langle \text{letter} \rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z;$

$\langle \text{variable} \rangle ::= \langle \text{letter} \rangle | \langle \text{variable} \rangle \langle \text{letter} \rangle | \langle \text{variable} \rangle \langle \text{number} \rangle;$

$\langle \text{operand} \rangle ::= \langle \text{number} \rangle | \langle \text{variable} \rangle;$

$\langle \text{addition operation} \rangle ::= \langle \text{operand} \rangle + \langle \text{operand} \rangle | \langle \text{addition operation} \rangle + \langle \text{operand} \rangle;$

Контекстно-вільна граматика — це формальна граматика $G = (N, T, P, S)$, де:

N — набір нетермінальних символів,

T — набір термінальних символів,

$S \in N$ — початковий символ,

P — набір правил виведення, який задається як $P = N \times (N \cup T)$.

Правила $(\alpha, \beta) \in P$ записуються як $\alpha \rightarrow \beta$. Ліва частина правила виведення повинна містити лише одну змінну (нетермінальний символ).

Термінальні символи — це елементарні символи мови, визначені формальною граматикою.

Нетермінальні символи (або синтаксичні змінні) замінюються групами термінальних символів відповідно до правил породження (утворення).

Сучасні мови програмування здебільшого створюються контекстно-вільними граматиками.

Канонічним прикладом контекстно-вільної граматики є відповідність дужок. Є чотири термінальні символи: $T = \{ '[', ']', '(', ')' \}$, і один нетермінальний символ S .

Правила породження такі:

1. $S \rightarrow SS$,

2. $S \rightarrow ()$,

3. $S \rightarrow (S)$,

4. $S \rightarrow []$,

5. $S \rightarrow [S]$,

У цій граматиці можна вивести таку послідовність:

$(([[[OO[[[]]]])()$].

Контекстно-залежна граматика — це формальна граматика $G = (N, T, P, S)$, де:

N — набір нетермінальних символів,

T — набір термінальних символів,

$S \in N$ — початковий символ,

P — правила виведення (породження) виду $\alpha X \beta \rightarrow \alpha \gamma \beta$ за умови:

$\alpha, \beta \in (N \cup T)$ або $\{\}$.

$X \in N$,

$\gamma \in (N \cup T)$.

Вихідними (початковими) правилами є $\alpha X \beta \rightarrow \alpha \gamma \beta$ і $\gamma \neq X$. Це означає, що нетермінальний символ X у контексті α і β буде замінено на γ . Однак, хоча довжина γ повинна бути принаймні одиниці, α і β можуть бути порожніми.

Приклад

$G = (N, T, P, S)$ з термінальними символами $T = \{a, b, c\}$, нетермінальними $N = \{B, C, H, S\}$ і правилами висновку P :

1. $S \rightarrow aSBC \mid aBC$,

2. $CB \rightarrow HB$,

3. $HB \rightarrow HC$,

4. $HC \rightarrow BC$,

5. $aB \rightarrow ab$,

6. $bB \rightarrow bb$,

7. $bC \rightarrow bc$,

8. $cC \rightarrow cc$.

Слово $aabbcc \in L$ можна отримати так:

$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aaBHBC \Rightarrow aaBHCC \Rightarrow aaBBCC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc$.

Граматика англійської мови

Ескіз граматики англійської мови з допомогою форми Бекуса-Найра:

1. **<sentence> ::= <subject> <predicate>**
2. **<subject> ::= <noun-phrase> | <pronoun>**
3. **<predicate> ::= <verb-phrase> | <verb-phrase> <object>**
4. **<noun-phrase> ::= <article>? <noun> (<adjective>)***
(**<article>?** означає, що артикль (наприклад, "the", "a", "an") може бути у фразі, але може і не бути відсутнім. Знак ***** вказує на те, що ця частина може повторюватися нуль або більше разів.)
5. **<verb-phrase> ::= <verb> | <verb> <adverb>**
6. **<object> ::= <noun-phrase> | <pronoun> | <preposition-phrase>**
7. **<pronoun> ::= <personal-pronoun> | <possessive-pronoun> | <reflexive-pronoun>**
8. **<personal-pronoun> ::= I | you | he | she | it | we | they**
9. **<possessive-pronoun> ::= my | your | his | her | its | our | their**
10. **<reflexive-pronoun> ::= myself | yourself | himself | herself | itself | ourselves | themselves**
11. **<noun> ::= <proper-noun> | <common-noun>**
12. **<proper-noun> ::= (names, specific entities)**
<proper-noun> ::= John | Mary | London | Eiffel Tower | Tuesday
13. **<common-noun> ::= (generic entities)**
<common-noun> ::= cat | dog | house | car | book
14. **<verb> ::= (action verbs, linking verbs)**
<verb> ::= run | jump | eat | write | think | create | is | are | were | am
15. **<adjective> ::= (descriptive words)**
<adjective> ::= big | red | happy | beautiful | old | new | interesting
16. **<adverb> ::= (words modifying verbs, adjectives, or other adverbs)**
<adverb> ::= quickly | quietly | very | now | here | often
17. **<preposition-phrase> ::= <preposition> <noun-phrase>**
18. **<preposition> ::= (words indicating relationships, such as "in," "on," "at")**
<preposition> ::= in | on | at | with | under | above | before | after | between
19. **<article> ::= a | an | the**

Ми можемо задавати також правила для конкретного часу з допомогою форми Бекуса-Наура. Оксфордські граматики не використовують такі формальні нотації, бо вони не зрозумілі для початківців.

Ось деякі правила використання часу (tense).

1. Ми використовуємо Present Simple для опису загального теперішнього стану, наприклад, I see. Для опису загальних фактів, наприклад, Sky is blue. Або для звичок які є зараз, наприклад, I like music, або I am happy.
2. Ми використовуємо Present Continuous для опису процесу, який зараз триває, наприклад, I am dancing. Або коли ми говоримо про строго заплановані події, наприклад, I am living tomorrow in the morning.
3. Ми використовуємо Present Perfect коли хочемо сказати, що якась дія була закінчена в минулому і стосується теперішнього. Наприклад, I have been in France.
4. Ми використовуємо Present Perfect Continuous коли хочемо сказати, що якийсь процес був розпочатий в минулому і стосується теперішнього. Наприклад, I have been reading book.

Для минулого й майбутнього часу все по типу цього, з необхідним зсувом часу.

Future Simple:

I will tell you.
I will be in France.
I will drink tea.

Future Continuous:

I will be playing football.
I will be working.

Future Continuous tense вживаємо коли хочемо вказати, що в майбутньому ми будемо в певному стані, в процесі чогось.

Future Perfect:

I will have done it.
I will have played.

Future Perfect Continuous:

I will have been watching movie.
I will have been counting stars.

Існують чотири основних типи умовних речень:

Нульове умовне речення:

(if + present simple (time adverb), ... present simple)

If you heat water to 100 degrees, it boils.

If it rains tomorrow, we stay indoors.

Перше умовне речення:

(if + present simple (time adverb), ... will + infinitive)

If it rains tomorrow, we will go to the cinema.

Друге умовне речення:

(if + past simple (time adverb), ... would + infinitive)

If I had a lot of money, I would travel around the world.

If I were born again tomorrow, I would change my life.

Третє умовне речення:

(if + past perfect (time adverb), ... would + have + past participle)

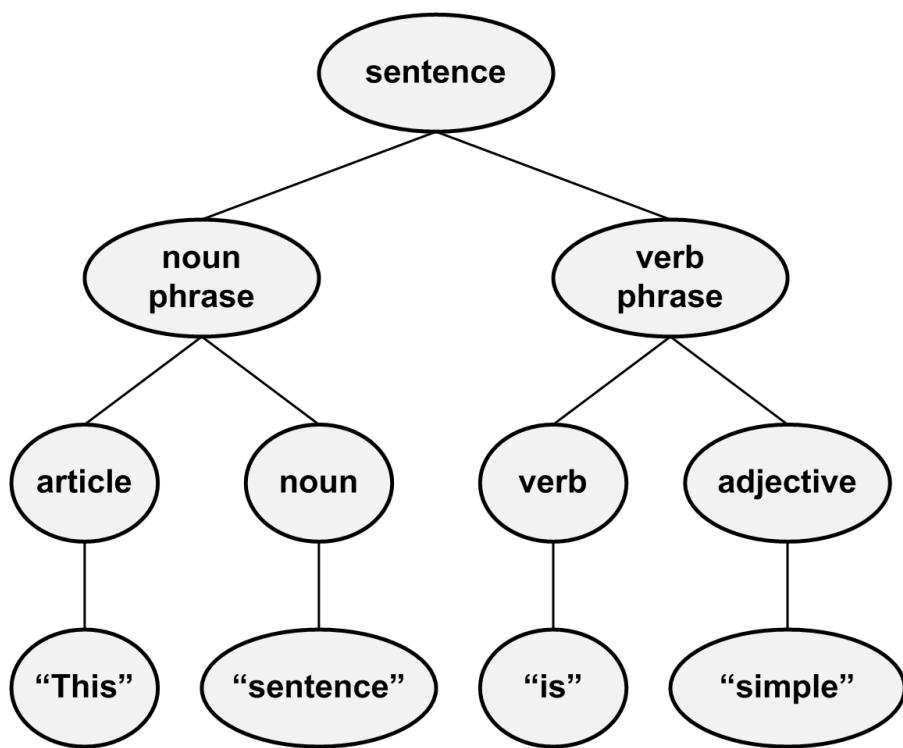
If I had gone to bed early, I would have caught the train.

Тут простий (Simple) час можна змінити на безперервний (Continuous), наприклад:

If she is still studying next week, she will be tired.

If he was sleeping, she was reading.

Синтаксичне дерево англійського речення



Що таке контекстно-вільна граматика?

Контекстно-вільна граматика (Context-Free Grammar, CFG) — це тип формальної граматики, який використовується для опису мови або синтаксичної структури різних програм, включаючи програми на багатьох програмних мовах. Цей тип граматики був введений Ноамом Хомським в 1956 році та є одним з найпоширеніших типів граматики, використовуваних в теорії формальних мов і комп'ютерних наук.

Контекстно-вільна граматика складається з чотирьох основних компонентів:

1. Нетермінали (non-terminals): Символи, які представляють синтаксичні категорії або структурні елементи мови. Нетермінали використовуються для визначення правил синтаксичного розбору.
2. Термінали (terminals): Символи, які представляють конкретні токени (слова, символи) мови. Термінали входять в рядки, які слід аналізувати.
3. Початковий символ (start symbol): Спеціальний нетермінал, з якого починається синтаксичний аналіз рядків.
4. Правила виводу (production rules): Спеціфікація того, як нетермінали можуть бути замінені на інші нетермінали або термінали. Ці правила визначають синтаксичну структуру мови.

Контекстно-вільні граматики використовуються для опису синтаксичної структури мови, такої як мови програмування, мови розмітки (наприклад, HTML або XML) і багатьох інших мов. Вони також широко використовуються в синтаксичних аналізаторах (парсерах), які визначають, чи відповідає даний рядок синтаксичним правилам мови.

Контекстно-вільна граматика — це формальна граматика $G = (N, T, P, S)$, де:

N — набір нетермінальних символів,

T — набір термінальних символів,

$S \in N$ — початковий символ,

P — набір правил виведення, який задається як $P = N \times (N \cup T)$.

Правила $(\alpha, \beta) \in P$ записуються як $\alpha \rightarrow \beta$. Ліва частина правила виведення повинна містити лише одну змінну (нетермінальний символ).

Термінальні символи — це елементарні символи мови, визначені формальною граматикою.

Нетермінальні символи (або синтаксичні змінні) замінюються групами термінальних символів відповідно до правил породження (утворення).

Сучасні мови програмування здебільшого створюються контекстно-вільними граматиками.

Канонічним прикладом контекстно-вільної граматики є відповідність дужок. Є чотири термінальні символи: $T = \{ '[', ']', '(', ')' \}$, і один нетермінальний символ S .

Правила породження такі:

1. $S \rightarrow SS$,
2. $S \rightarrow ()$,
3. $S \rightarrow (S)$,
4. $S \rightarrow []$,
5. $S \rightarrow [S]$,

У цій граматиці можна вивести таку послідовність:

$([[[OO[[[]]])([]])$.

Що таке нотація Бекуса-Наура?

Нотація Бекуса-Наура (або БНФ) — це метамова або формальна граматика, яка використовується для опису синтаксису мов програмування, мов розмітки, мов запитів та інших текстових форматів. Ця нотація була розроблена в 1960-х роках Джоном Бекусом і Пітером Науром.

Нотація Бекуса — Наура також застосовуються для опису протоколів, наприклад, для опису протоколу HTTP/1.1 у RFC 2616.

В нотації БНФ граматичні конструкції описуються за допомогою правил, які виглядають наступним чином:

<символ> ::= <вираз>,

де "<символ>" представляє називу нетермінального символу (наприклад, називу граматичної конструкції), і "<вираз>" представляє послідовність символів і/або терміналів (кінцевих символів).

Нотація БНФ стала стандартом для опису синтаксису багатьох мов програмування і мов розмітки, таких як C, C++, Java, HTML, XML та інших. Вона дозволяє формалізувати та узгоджувати правила синтаксису мови, що сприяє розробці компіляторів, інтерпретаторів та інших інструментів для обробки текстових даних.

У інформатиці форма Бекуса-Наура — це метасинтаксичне позначення для контекстно-вільних граматик, яке часто використовується для опису синтаксису мов, що використовуються в обчислювальній роботі, таких як мови комп'ютерного програмування, формати документів, набори інструкцій та протоколи зв'язку.

Джін Саммет у своїй книзі "Мови програмування: історія й основи" (1969) активно використовувала форму Бекуса-Наура.

Приклад рекурсивного визначення у формі Бекуса-Наура:

<identifier> ::= <letter> | <identifier><letter> | <identifier><number>

Сенс цього позначення полягає у визначенні класу об'єктів, які називаються "ідентифікаторами" (identifier). Спочатку стверджується, що кожен елемент раніше визначеного класу <letter> є ідентифікатором, потім наводяться правила генерації, які полягають у тому, що присвоєння ідентифікатору літери або цифри праворуч знову дає ідентифікатор. Тобто в наведеному вище випадку об'єкт є ідентифікатором, якщо це "літера", "літера + літера", "літера + літера + цифра", або "літера + цифра", або "літера + цифра + цифра". (A, AB, A1, A1B, A1B2, ..., але не 1, 1A, ...).

Приклад налаштування синтаксису для операції додавання (натуральних чисел):

1. **<number> ::= 0|1|2|3|4|5|6|7|8|9**
2. **<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z**
3. **<variable> ::= <letter> | <variable><letter> | <variable><number>**
4. **<operand> ::= <number> | <variable>**
5. **<addition operation> ::= <operand>+<operand> | <addition operation>+<operand>**

"?" - це спеціальний символ, який вказує на те, що попередній символ або група символів є необов'язковими у визначені граматики.

Наприклад, якщо ви маєте правило:

<опціональний_елемент> ::= <елемент>?

"*" - це символ, який вказує на те, що попередній символ або група символів може повторюватися довільну кількість разів (включаючи нуль разів). Наприклад, якщо ви маєте правило:

<повторюваний_елемент> ::= <елемент>*

Це означає, що **<елемент>** може повторюватися нуль або більше разів.

Приклад циклу while у формі Бекуса-Наура (BNF):

```
<statement> ::= <assignment> | <while_loop> | <other_statement>

<while_loop> ::= "while" <condition> "do" <block>

<block> ::= "{" <statement>* "}"

<condition> ::= <expression>

<expression> ::= <operand> <operator> <operand>

<operand> ::= <variable> | <constant>

<operator> ::= "<" | ">" | "==" | "!=" | "<=" | ">="

<variable> ::= [a-zA-Z]++

<constant> ::= [0-9]++
```

Форма МакКімена

Форма МакКімена — це нотація для вираження граматик. Її запропонував Білл МакКімен з Дартмутського коледжу. Це спрощена форма Бекуса-Наура зі значними пробілами та мінімальним використанням метасимволів. Дуглас Крокфорд використав форму МакКімена для опису формату JSON.

Ми можемо виразити граматику форми МакКімана у формі МакКімана.

Граматика (grammar) — це список з одного або кількох правил.

grammar
rules

Як пробіл (space) використовується код Unicode U+0020. Кодова точка Unicode U+000A (\n) використовується як новий рядок (newline).

space
'0020'

newline
'000A'

Ім'я (name) — це послідовність літер або _underbar.

name
letter
letter name

letter
'a' . 'z'
'A' . 'Z'
'_'

Відступ (indentation) — це чотири інтервали.

indentation
space space space space

Кожне правило відокремлюється новим рядком (newline). Правило (rule) має назву (name) в одному рядку з альтернативами (alternatives) відступленими під ним.

rules
rule
rule newline rules

rule
name newline
nothing
alternatives

Якщо перший рядок після назви правила "", то правило може не відповідати нічому (nothing).

nothing

""

indentation """" newline

Кожна альтернатива відступлена на окремому рядку. Кожна альтернатива містить елементи (items), які йдуть після символу нового рядка.

alternatives

alternative

alternative alternatives

alternative

indentation items newline

Елементи розділені пробілами. Елемент — це літерал або назва правила.

items

item

item space items

item

literal

name

literal

singleton

range exclude

"" characters ""

Будь-яка окрема кодова точка Unicode, крім 32 контрольних кодів, може бути розміщена в одинарних лапках. Шістнадцятковий код (hexcode) будь-якої точки коду Unicode також можна помістити в одинарні лапки. Шістнадцятковий код може містити 4, 5 або 6 шістнадцяткових цифр.

singleton

"" codepoint ""

codepoint

' ' . '10FFFF'

hexcode

hexcode

"10" hex hex hex hex

hex hex hex hex hex

hex hex hex hex

hex

'0' . '9'

'A' . 'F'

Діапазон (range) вказується як один елемент, .period та ще один елемент. Літеральні діапазони можуть супроводжуватися знаком -minus і символами, які потрібно виключити.

range

singleton space '.' space singleton

exclude

""

space '-' space singleton exclude

space '-' space range exclude

Символ, загорнутий у “подвійні лапки, може бути будь-яким кодом Unicode, крім 32 контрольних кодів і “подвійних лапок. Визначення символу показує приклад діапазону кодових точок і виключення.

characters

character

character characters

character

' ' .. '10FFFF' - """

Що таке Абстрактне синтаксичне дерево?

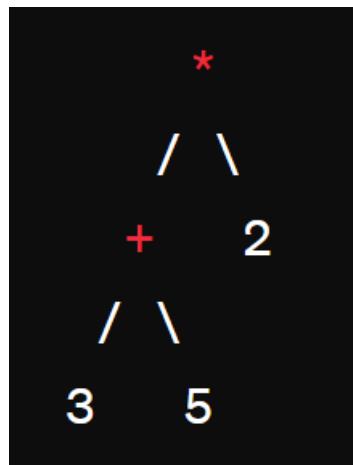
Абстрактне синтаксичне дерево (Abstract syntax tree - AST) — це структура даних, яка використовується в інформації для представлення структури програми або фрагмента коду. Це деревоподібне представлення абстрактної синтаксичної структури тексту (часто вихідного коду), написаного формальною мовою. Кожен вузол дерева позначає конструкцію, що зустрічається в тексті. Іноді його називають просто синтаксичним деревом.

Синтаксис є "абстрактним" у тому сенсі, що він не представляє кожну деталь, яка з'являється в реальному синтаксисі, а лише структурні або пов'язані зі змістом деталі. Наприклад, групувальні дужки є неявними в структурі дерева, тому їх не потрібно представляти як окремі вузли. Подібним чином синтаксична конструкція, як оператор if-умова-то, може бути позначена за допомогою одного вузла з трьома гілками.

У порівнянні з початковим кодом, AST не містить допоміжних знаків пунктуації та роздільників (дужок, крапки з комою, круглих дужок тощо).

Синтаксичні дерева є важливим інструментом у парсерах для аналізу синтаксису речень або програм.

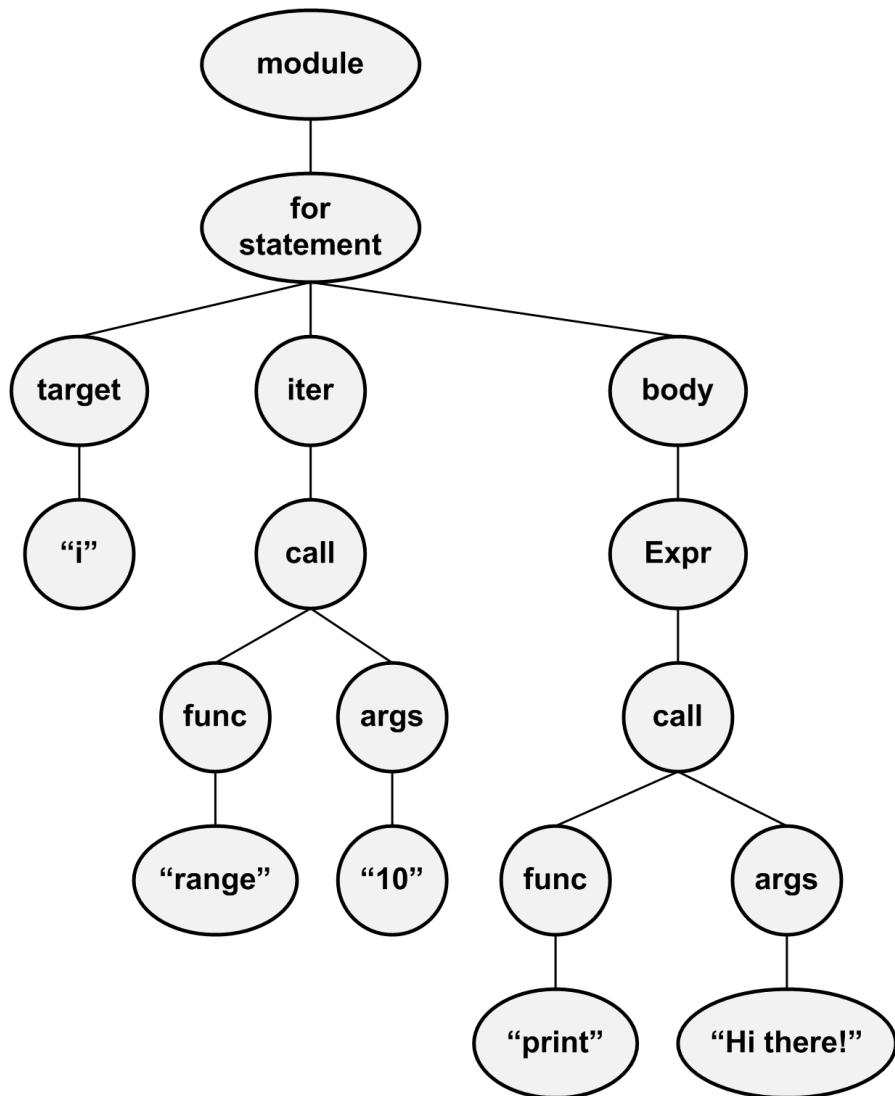
Розглянемо вираз математичного обчислення: "(3 + 5) * 2":



Приклад синтаксичного дерева для циклу for на мові Python:

```
#Код 10 разів виведе привітання “Hi there!”
```

```
for i in range(10):
    print('Hi there!')
```



Що таке польська нотація?

Польський математик Ян Лукашевич (який народився у Львові), винайшов, у 1924 році, нотацію, яка відома як нотація Лукашевича або польська нотація.

Польська нотація — це форма запису математичних виразів, де оператор розташовується перед операндами. Наприклад, у звичайному записі виразу "2 + 3" бінарний оператор (+) знаходиться між операндами (2 і 3). У польській нотації цей же вираз буде записаний як "+ 2 3", де оператор (+) знаходиться перед операндами. Це дає можливість уникнути неоднозначностей і забезпечити однозначне розуміння порядку операцій у виразах без використання дужок.

Розглянемо вираз: $3 + 4 * 2$.

У звичайному математичному записі, ми спочатку виконаємо множення, а потім додавання:

$$3 + 4 * 2 = 3 + 8 = 11.$$

Але якщо хтось забуде або помилково не врахує правило пріоритету операцій і почне здійснювати додавання перед множенням, отримаємо інший результат: $3 + 4 * 2 = 7 * 2 = 14$.

Це призводить до плутанини й може привести до неправильного результату через неправильне розуміння порядку операцій.

За використання польської нотації цей же вираз буде записаний як "+ 3 * 4 2". Таким чином, порядок операцій стає однозначним і не допускає помилок або непорозумінь.

Вираз "+ 3 * 4 2" у польській нотації означає "додати до 3 добуток чисел 4 і 2"

Мова Lisp використовує свою власну нотацію, яка називається S-виразами. Ця нотація базується на ідеї використання списків для представлення коду програми. Хоча S-вирази можуть нагадувати префіксну (польську) нотацію, оскільки оператор розташовується перед операндами, це все ж таки відрізняється від класичної польської нотації. У класичній польській нотації оператори розміщені перед операндами, але в Lisp вони розташовані у вигляді списків, де перший елемент є оператором, а решта елементів є його operandами.

Приклад Lisp:

(+ 1 2 3 4 5) // 15

(+ 1 2 3 (* 4 5)) // 26

Що таке регулярний вираз?

Регулярний вираз — це шаблон, який використовується для пошуку та відповідності тексту певним рядкам. Він використовується в багатьох програмах та мовах програмування для роботи з текстовими даними. Регулярні вирази дозволяють вам виконувати різноманітні операції з рядками, такі як пошук певного тексту, заміна тексту, вилучення підрядка і багато інших.

Регулярні вирази складаються з різних символів і конструкцій, які утворюють шаблон для пошуку. Деякі звичайні символи, які використовуються в регулярних виразах, включають кількість, діапазони символів, альтернативи, анкори (позиції на початку або кінці рядка) та інші.

Наприклад, регулярний вираз `^d{3}-d{2}-d{4}` відповідає рядку, який містить номер соціального страхування в США у форматі "###-##-####", де # - це цифри. Регулярні вирази дуже корисні при обробці тексту, валідації даних та багатьох інших завданнях, пов'язаних із текстовою обробкою.

Регулярні вирази та скінченні automati (finite automata) пов'язані через теорію формальних мов та теорію автоматів. Регулярні вирази можуть бути перетворені в скінченні automati, і навпаки, скінченні automati можуть бути використані для визначення, чи відповідає певний рядок регулярному виразу.

Скінчений автомат — це обчислювальний пристрій, який можна використовувати для розпізнавання рядків, що задовільняють певний регулярний вираз. Регулярні вирази використовуються для пошуку, виділення або перевірки текстової інформації в рядках. Якщо ви хочете створити скінчений автомат, який розпізнає рядки, які відповідають регулярному виразу "ab*c", то ось приклад опису такого автомата:

1. Стани:

- Початковий стан: q0
- Допустимий (або кінцевий) стан: q1
- Нездовільний стан: q2

2. Алфавіт: {a, b, c}

3. Переходи:

- З q0 в q1 при входному символі 'a'.
- З q1 в q1 при входному символі 'a' або 'b'.
- З q1 в q2 при входному символі 'c'.

4. Початковий стан: q0

5. Фінальний (кінцевий) стан: q1

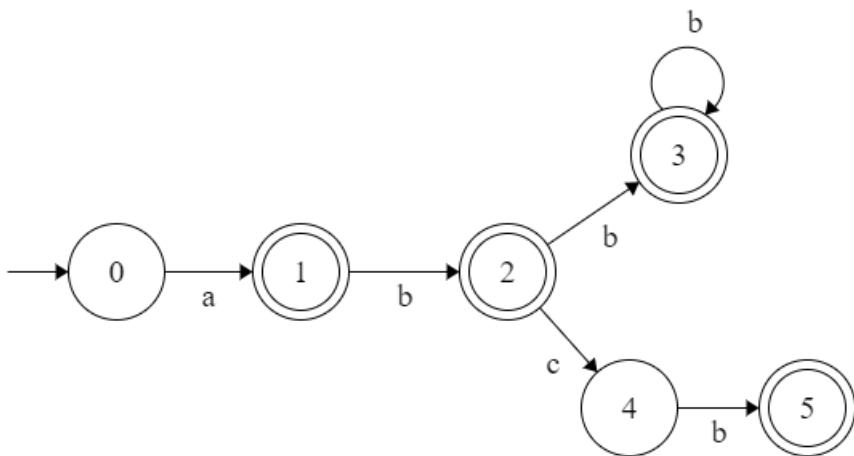
6. Переход до стану q1 після послідовності символів 'a' або 'b', які можуть повторюватися довільну кількість разів.

7. Переход до стану q2 після символу 'c'.

8. Якщо автомат завершив обробку всіх символів входного рядка у стані q1, то він приймає цей рядок як валідний за регулярним виразом "ab*c".

Отже, цей скінчений автомат реалізує регулярний вираз "ab*c" і призначений для розпізнавання рядків, які містять послідовність символів 'a', далі може слідувати будь-яка кількість символів 'b', і закінчується символом 'c'.

Традиційно малювати скінченні автомати у вигляді графа, де стани намальовані як кола, а прийнятні стани намальовані як подвійні кола. Наприклад, на малюнку зображений автомат, який відповідає регулярному виразу $a(b^*|bcb)$; зауважте, що він має чотири фінальні стани.



Ось список загальних шаблонів регулярних виразів (regex) в мові JavaScript разом із поясненнями:

1. Основна відповідність тексту:

- `'/hello/'`: Відповідає точному рядку "hello" в тексті.

2. Символи підстановки:

- `'./'`: Відповідає будь-якому одинокому символу, крім символу нового рядка.
- `'/a.b/'`: Відповідає "a", будь-якому символу і потім "b."

3. Класи символів:

- `'/[aeiou]/'`: Відповідає будь-якій голосній.
- `'/[^0-9]/'`: Відповідає будь-якому символу, який не є цифрою.

4. Кількісні обмежувачі:

- `'/a*/'`: Відповідає нулю або більше входжень "a."
- `'/a+/'`: Відповідає одному або більше входжень "a."
- `'/a?/'`: Відповідає нулю або одному входженню "a."
- `'/a{2,4}'`: Відповідає "a" повтореному від 2 до 4 разів.

5. Якорі:

- `'/^start/'`: Відповідає "start" лише на початку рядка.
- `'/end$/'`: Відповідає "end" лише в кінці рядка.

6. Межі слів:

- `'\bслово\b': Відповідає "слово" як цілому слову.

7. Альтернація:

- `/кіт|собака/': Відповідає або "кіт", або "собака."

8. Екранування символів:

- `'\d': Відповідає будь-якій цифрі (еквівалентно '[0-9]').
- `'\s': Відповідає будь-якому символу пробілу.
- `'\w': Відповідає будь-якому символу слова (букви, цифри або підкреслення).
- `'\D': Відповідає будь-якому символу, який не є цифрою.
- `'\S': Відповідає будь-якому символу, який не є пробілом.
- `'\W': Відповідає будь-якому символу, який не є символом слова.

9. Групи та збереження:

- `'(abc)+': Відповідає одному або більше входжень "abc" як групі.
- `'(\d{2})-(\d{2})-(\d{4})': Зберігає дату в форматі "дд-мм-рррр."

10. Модифікатори:

- `/шаблон/i': Виконує відповідність без урахування регістру.
- `/шаблон/g': Відповідає всім входженням у рядку (глобально).
- `/шаблон/m': Розглядає рядок як декілька рядків.

11. Перегляди вперед та назад:

- `'(?=foo)': Позитивний перегляд вперед, відповідає, якщо "foo" передує.
- `'(?<!foo)': Негативний перегляд назад, відповідає, якщо "foo" не передує.

12. Зворотні посилання:

- `'(abc)\1': Відповідає "abcabc", посилаючись на першу групу збереження.

Це лише основи, і регулярні вирази можуть стати досить складними. Вони дуже корисні для завдань, таких як перевірка, пошук та маніпулювання текстом в JavaScript.

Що таке скінчений автомат?

Скінчений автомат — це абстрактна математична модель, яка використовується для опису автоматичних процесів та обчислень. Він є дискретним автоматом, що працює у визначеному наборі станів і відповідає на вхідні події переходами між цими станами.

Скінчений автомат складається з наступних основних компонентів:

1. Множина станів (States): Це обмежений набір можливих станів, у яких може перебувати автомат. Кожен стан представляє певний контекст чи умову системи.
2. Алфавіт вхідних символів (Input Alphabet): Це набір можливих вхідних символів, які автомат може приймати. Вхідні символи спричиняють переходи між станами.
3. Функція переходів (Transition Function): Ця функція визначає, як автомат реагує на вхідні символи і переходить між різними станами. Вона вказує, який стан буде активним після обробки певного символу.
4. Початковий стан (Start State): Це початковий стан, в якому автомат знаходиться на початку обчислення або процесу.
5. Множина заключних станів (Final States): Ця множина визначає, які стани вважаються заключними або приймаючими станами. Якщо автомат перебуває в заключному стані після обробки вхідних символів, то вважається, що вхід прийнятий.

Скінченні automati використовуються для моделювання різних процесів та задач в інформатиці та теорії автоматів, такі як розпізнавання мов, синтаксичний аналіз, керування програмами, автоматизація, керування витратами енергії, керування мережами, тощо. Вони є важливою теоретичною основою для розв'язання багатьох завдань у сферах інформатики та інженерії.

Скінчений автомат (finite automaton) можна математично описати наступним чином:

1. Множина станів (Q): Нехай Q буде скінченою множиною станів. Це означає, що автомат може перебувати лише у певному скінченому наборі станів.
2. Алфавіт вхідних символів (Σ): Нехай Σ буде скінченою множиною вхідних символів. Це символи, які можуть бути прийняті автоматом.
3. Функція переходів (δ): Функція переходів $\delta: Q \times \Sigma \rightarrow Q$ визначає, як автомат переходить з одного стану в інший при обробці вхідних символів. Наприклад, $\delta(q, a) = r$ означає, що при обробці символу 'a' автомат переходить зі стану 'q' в стан 'r'.
4. Початковий стан (q_0): q_0 є початковим станом, з якого розпочинається обробка вхідних символів.
5. Множина заключних станів (F): F є множиною заключних станів, і вхід вважається прийнятим, якщо автомат знаходиться в одному з цих станів після обробки всіх вхідних символів.

Математично, скінчений автомат $(Q, \Sigma, \delta, q_0, F)$ представляє собою п'ятірку, де Q - множина станів, Σ - алфавіт вхідних символів, δ - функція переходів, q_0 - початковий стан, і F - множина заключних станів. Використовуючи цю модель, можна описати, як автомат приймає вхідні послідовності символів та переходить між станами для обробки цих символів.

Що таке логіка предикатів?

Логіка предикатів — це галузь математики та філософії, яка вивчає властивості і відношення між об'єктами в контексті математичних виразів і висловлювань. Ця логіка спрямована на аналіз і розв'язання проблем, пов'язаних з властивостями об'єктів та їх відношеннями, використовуючи предикати — функції, які приймають об'єкти як аргументи та повертають істину або хибність висловлювань.

Основні поняття в логіці предикатів включають:

1. Предикати: Вони використовуються для вираження властивостей або відношень між об'єктами. Наприклад, " $x > 5$ " - це предикат, який вказує на те, що число x більше за 5.
2. Квантори: Логіка предикатів включає квантори, такі як квантор загальності (\forall) і квантор існування (\exists), які використовуються для вираження кількості об'єктів, які задовольняють певному предикату. Наприклад, " $\forall x (x > 0)$ " висловлює твердження, що "для всіх x , x більше за 0."
3. Логічні операції: У логіці предикатів використовуються логічні операції, такі як кон'юнкція (\wedge), диз'юнкція (\vee) та заперечення (\neg), для побудови складних висловлювань.

Логіка предикатів грає важливу роль у формалізації різноманітних областей, включаючи математику, програмування, штучний інтелект та філософію. Вона допомагає аналізувати, доводити та розв'язувати проблеми, пов'язані з виразами та властивостями об'єктів у точний та систематичний спосіб.

Числення предикатів (логіка першого порядку)

Літери: A, A1, B, C, ..., Z означають предикат, який може бути істинним (1) або хибним (0).

Предикат може бути простим або складеним, простий предикат називається атомарною формулою, а складний — складеною (або комплексною) формулою.

Атомарну формулу не можна розділити на частини, які при цьому залишаться предикатами, складену формулу можна розділити на частини, які все ще залишаться атомарною формулою або складеною формулою.

Складену формулу утворюють з атомарних формул з допомогою логічних зв'язків, як-от диз'юнкція (\vee), може читатися як "або", кон'юнкція (\wedge), може читатися як "і", (імплікація) \rightarrow , може читатися як "тоді", інверсія (\neg), може читатися як "не".

Диз'юнкція (\vee) та кон'юнкція (\wedge) є бінарними (двійковими) операторами, оскільки вони вимагають двох параметрів, а \wedge b, а \vee b. (Термін "параметр" та "аргумент" ми можемо вживати як синоніми, але часто під аргументом розуміють конкретне значення яке задається параметру функції).

Заперечення (\neg) є унарним оператором, оскільки приймає лише один параметр a, тобто $\neg a$.

\vee , \wedge , \neg дотримуються законів булевої алгебри для $+$, $*$, $-$;

\vee , \wedge , \neg — базові оператори, з допомогою яких ми можемо побудувати всі інші оператори, наприклад \rightarrow , \leftrightarrow (логічна рівність), які використовуються для зручності.

Три основні операції: \vee , \wedge , \neg .

$A \rightarrow B$ те саме, що $\neg A \vee B$.

$A \leftrightarrow B$ те саме, що $(\neg A \wedge \neg B) \vee (A \wedge B)$.

$A \wedge B$ те саме, що $\neg(\neg A \vee \neg B)$;

Таблиці істинності

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

A	$\neg A$
0	1
1	0

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Мало хто знає, що грецький філософ Секст Емпірик був першим хто описав, що:
 "З істини слідує істина, а з брехні що завгодно".

Цим пояснюється логічна іmplікація (оператор іmplікація \rightarrow), яка хибна тільки коли з істини слідує брехня, але не навпаки.

A	B	$A \leftrightarrow B$
0	0	1
0	1	0
1	0	0
1	1	1

Стріла Пірса (ввів англійський вчений Чарльз Пірс)

A	B	$A \downarrow B$
0	0	1
0	1	0
1	0	0
1	1	0

З допомогою стрілки Пірса можна визначити всі інші оператори.

$$A \downarrow A = \neg A;$$

$$(A \downarrow A) \downarrow (B \downarrow B) = A \wedge B;$$

$$(A \downarrow B) \downarrow (A \downarrow B) = A \vee B;$$

Оператор \downarrow відомий як стрілка Пірса. Чарльз Пірс у неопублікованих рукописах вперше розглянув його як логічний оператор і показав, що він може виражати логічне НЕ, І та АБО. Генрі Шеффер був першим, хто опублікував опис цього оператора.

Наприклад, атомарними формулами в арифметиці (теорії чисел) є $a = b$, $a < b$, $a > b$, $a:b = d:c$, де $=, <, >$ двійкові предикати, $a, b, a:b$ є термінами.

Якщо A і B є формулами, то $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \leftrightarrow B)$, $\neg A$, $\forall x A(x)$, $\exists x A(x)$ також є формулами.

Формулу $(A \wedge (B \vee C)) \rightarrow D$ можна читати як “якщо A істинне і B або C істинне, то D істинне”.

Квантори:

$\exists x$ - "існує x ", $\forall x$ - "всі x ", $\exists!x$ - "існує тільки один x ".

\exists - існують. \forall - будь-який, усі.

Протиріччя (суперечність) – це формула, яка завжди хибна, незалежно від параметрів.

Тавтологія — це формула, яка завжди правильна, незалежно від параметрів.

Аксіоми числення предикатів є логічними тавтологіями.

Аксіоми числення предикатів (аксіоми Вайтгеда-Рассела для логіки першого порядку):

AL1. $(A \vee A) \rightarrow A$, (Закон ідемпотентності);

AL2. $A \rightarrow (A \vee B)$;

AL3. $(A \vee B) \rightarrow (B \vee A)$, (Закон комутативності);

AL4. $(A \rightarrow B) \rightarrow (C \vee A \rightarrow C \vee B)$;

Квантори (\forall, \exists):

AL5. $\forall x A(x) \rightarrow A(y)$;

AL6. $A(y) \rightarrow \exists x A(x)$;

Правила виводу

Модус поненс (MP).

AL7. MP: $(A \wedge (A \rightarrow B)) \rightarrow B$;

Модус толенс (MT).

AL8. MT: $((A \rightarrow B) \wedge \neg B) \rightarrow \neg A$;

Інші тавтології (теореми)

F1. $(A \rightarrow B) \rightarrow ((C \rightarrow A) \rightarrow (C \rightarrow B))$;

Доказ F1.

$(A \rightarrow B) \rightarrow (\neg C \vee A \rightarrow \neg C \vee B)$, (AL4);

$C \rightarrow A = \neg C \vee A$, $C \rightarrow B = \neg C \vee B$;

F2. $\neg A \vee A$, (Закон виключеного третього);

Доказ F2.

$A \rightarrow (A \vee A)$, (AL2);

$(A \vee A) \rightarrow A$, (AL1);

$((A \vee A) \rightarrow A) \rightarrow ((A \rightarrow (A \vee A)) \rightarrow (A \rightarrow A))$, (F1);

$A \rightarrow A = \neg A \vee A$, (За визначенням \rightarrow);

$A \vee \neg A$, (AL3);

F3. $A \rightarrow \neg\neg A$, (Правило подвійного заперечення);

Доказ F3.

$A \rightarrow \neg\neg A = \neg A \vee \neg\neg A$ (За визначенням \rightarrow);

$\neg A \vee \neg\neg A$ (F2).

F4. $\neg\neg A \rightarrow A$, (Правило подвійного заперечення);

Доказ F4.

$\neg A \rightarrow \neg\neg\neg A$, (F3);

$A \vee \neg A \rightarrow A \vee \neg\neg\neg A$, (F2);

$\neg\neg\neg A \vee A$, (AL3);

$\neg\neg\neg A \vee A = \neg\neg A \rightarrow A$, (За визначенням \rightarrow);

F5. $(A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$, (Закон контрапозиція);

Доказ F5.

$B \rightarrow \neg\neg B$, (F3);

$\neg A \vee B \rightarrow \neg A \vee \neg\neg B$, (AL1, AL2);

$\neg A \vee \neg\neg B \rightarrow \neg\neg B \vee \neg A$, (AL3);

$\neg A \vee B \rightarrow \neg\neg B \vee \neg A$, (F1);

$\neg A \vee B \rightarrow \neg\neg B \vee \neg A = (A \rightarrow B) \rightarrow (\neg B \rightarrow \neg A)$;

F6. $\neg(A \wedge B) \rightarrow (\neg A \vee \neg B)$, (Закон де Моргана);

Доказ F6.

$A \wedge B = \neg(\neg A \vee \neg B)$, (за визначенням \wedge);

$\neg(\neg(\neg A \vee \neg B)) \rightarrow (\neg A \vee \neg B)$, (F4).

F7. $\neg(A \vee B) \rightarrow (\neg A \wedge \neg B)$, (Закон де Моргана);

Доказ F7.

$\neg(A \vee B) \rightarrow (\neg(\neg A \vee \neg B))$, (за визначенням \wedge);

$\neg(\neg A \vee \neg B) = \neg(A \vee B)$, (F4).

F8. $A \wedge B \rightarrow B \wedge A$;

Доказ F8.

$\neg(\neg A \wedge \neg B) \rightarrow \neg(\neg B \wedge \neg A)$;

F9. $A \wedge B \rightarrow A$;

Доказ F9.

$\neg(\neg A \vee \neg B) \rightarrow A$, (за визначенням \wedge);

$\neg A \rightarrow (\neg A \vee \neg B)$, (AL2);

$\neg(\neg A \vee \neg B) \rightarrow \neg\neg A$, (F5);

$A \wedge B \rightarrow \neg\neg A$;

$A \wedge B \rightarrow A$;

Буває зручно використовувати оператори, які вказують на часові межі вірності формули. Наприклад, оператор \square означає “завжди”, а оператор \diamond означає “врешті-решт”, або “є, чи буде”. Наприклад, формула $\square P$ означає, що предикат P завжди вірний, а формула $\diamond P$ означає $\neg \square \neg P$, тобто: “ P не завжди хибне”.

Амір Пнуелі отримав премію Тюрінга за основоположну працю, що ввела поняття темпоральної логіки у комп'ютерні науки і видатний внесок у верифікацію програм та систем. Леслі Лемпорт також відомий своєю роботою з темпоральної логіки, де він представив часову логіку дій (TLA).

Кон'юнктивна нормальна форма (КНФ) в булевій логіці — нормальна форма в якій булева формула має вид кон'юнкції декількох диз'юнктів (де диз'юнктами називаються диз'юнкції декількох пропозиційних символів або їх заперечень). Кон'юнктивна нормальна форма широко використовується в автоматичному доведенні теорем.

Наступна формальна граматика описує всі формули, приведені до КНФ:

<КНФ> → <диз'юнкт>
<КНФ> → <КНФ> \wedge <диз'юнкт>
<диз'юнкт> → <літерал>
<диз'юнкт> → (<диз'юнкт> \vee <літерал>)
<літерал> → <терм>
<літерал> → \neg <терм>

де <терм> позначає довільну булеву змінну.

Приклад:

$(A \vee B \vee \neg C) \wedge (\neg D \vee E \vee F) \wedge (C \vee D)$;

Довільна булева формула може бути приведена до КНФ за допомогою наступного алгоритму:

Крок 1 : Усі логічні зв'язки виразити через кон'юнкцію, диз'юнкцію і заперечення.

Крок 2 : Скасувати всі подвійні заперечення і використати, де можливо, правила де Моргана.

Що таке комп'ютерний процесор і як його виготовляють?

Сучасний комп'ютерний процесор (CPU) — це пристрій на основі напівпровідників логічних вентилів, який може читувати дані у двійковому форматі й виконувати команди, які також представлені у двійковому форматі. Зазвичай до команд процесора належать арифметичні оператори, булеві оператори, умовні оператори, команди збереження в пам'ять і читування з пам'яті. Процесор може звертатися до зовнішньої пам'яті та до внутрішньої пам'яті, яка представлена регістрами процесора. Регістри процесора слугують для збереження проміжних результатів обчислення та в ролі прапорців (flags), які говорять про стан процесора.

Арифметичні обчислення зазвичай реалізовані за стандартом IEEE 754.

Сучасний комп'ютерний процесор — це електричний пристрій на основі логічних вентилів, які реалізовані з допомогою напівпровідників.

Основні логічні вентилі: вентиль АБО (OR), вентиль I (AND), вентиль НЕ (NOT).

З допомогою цих трьох вентилів можна створити напівсуматор, елемент, який використовується в суматорах двійкових чисел.

CISC (Complex instruction set computer) та RISC (Reduced instruction set computer) — це два різних підходи до проектування процесорів, які використовуються в архітектурі комп'ютерів. Основна різниця між ними полягає в складності набору інструкцій і способів виконання операцій.

CISC (комп'ютер зі складним набором команд) — це архітектура системи команд, в якій більшість команд є комплексними, тобто реалізують певний набір простіших інструкцій процесора або шляхом зіставлення з кожною CISC-командою певної мікропрограми, або принаймні можуть бути зведені до набору таких простих інструкцій.

CISC використовує складні інструкції, які можуть виконувати кілька операцій одночасно. Наприклад, одна інструкція може включати завантаження даних з пам'яті, арифметичну операцію та збереження результату. Це робить CISC-процесори гнучкими.

RISC, навпаки, використовує прості інструкції, кожна з яких виконує лише одну операцію. Це дозволяє їм працювати на високих частотах та забезпечує більш ефективне виконання інструкцій. Однак це може вимагати більшої кількості інструкцій для виконання складних завдань. У сучасних системах можна знайти обидві архітектури, використовуючи їх залежно від контексту та вимог.

На початку комп'ютерної індустрії, коли програми писались мовою асемблера, основною метою було забезпечення максимальної продуктивності й універсальності процесорів. Це означало, що кожна інструкція повинна працювати з операндами, розташованими як у пам'яті, так і в регістрах, або безпосередньо в інструкції.

Для досягнення цієї універсальності розробники процесорів включали в них різноманітні типи адресації, що дозволяло здійснювати доступ до даних з різних джерел. Однак це призводило до того, що для кожної операції потрібно було реалізувати кілька шляхів взаємодії з даними, що робило процесори складнішими й, зрештою, менш ефективними.

Крім того, в умовах обмежених ресурсів пам'яті, де кожен байт коштував гроші, виникла потреба в оптимізації розміру програми. Тому інструкції змінної довжини, які могли одночасно завантажувати дані з пам'яті та виконувати обчислення, були вигідними з точки зору економії місця.

Однак в середині 1970-х років наукові дослідження показали, що складні інструкції не завжди використовувалися ефективно компіляторами, багато процесорів мали надлишковий набір інструкцій, а в деяких випадках прості інструкції працювали швидше за складні. Ці відкриття спонукали до змін в архітектурі процесорів, спрямованих на спрощення і прискорення виконання інструкцій.

Інструкцій за секунду (Instructions per second, IPS) — це показник швидкості процесора комп'ютера. Для комп'ютерів зі складним набором інструкцій (CISC) різні інструкції потребують різної кількості часу, тому вимірює значення залежить від суміші інструкцій; навіть для порівняння процесорів одного сімейства вимірювання IPS може бути проблематичним.

Термін зазвичай використовується разом із метричним префіксом (k, M, G, T, P або E) для формування кіль інструкцій за секунду (kIPS), мега інструкцій за секунду (MIPS), гіга інструкцій за секунду (GIPS) і так далі.

IBM 7030, також відомий як Stretch — перший суперкомп'ютер компанії IBM, побудований на транзисторах. Був найшвидшою ЕОМ (Електронно обчислювальною машиною) у світі з 1961 до 1964 року, коли запрацювала перша машина CDC 6600.

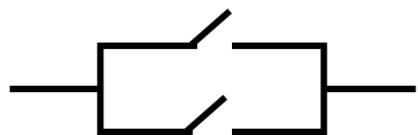
Stretch працював зі швидкістю 1.200 MIPS з частотою 3.30 MHz.

Логічні вентилі

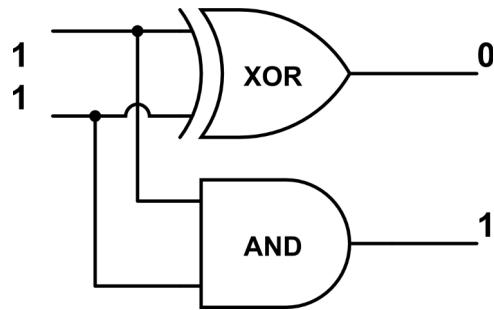
Логічні вентилі, як дріт з електричним струмом:



Вентиль — “І”. Якщо обидва контакти замкнуті, тоді по провіднику потече електричний струм, що буде означати в математичному сенсі 1, інакше 0.



Вентиль — “АБО”. Струм потече по провіднику якщо хоча б один з контактів замкнутий.



Напівсуматор, реалізований на елементах “ВИКЛЮЧНЕ АБО” та “І”. Якщо напруга подається до кожного з двох входних контактів логічного вентиля “ВИКЛЮЧНЕ АБО”, то на виході цього вентиля напруги не буде. Такий же результат буде отримано, якщо на двох входах немає напруги. Вихід “ВИКЛЮЧНЕ АБО” буде передавати напругу, тільки якщо один і тільки один з входів знаходиться під напругою (а інший ні). Логічний вентиль “І” видасть напругу на вихід, тільки якщо обидва його входні контакти знаходяться під напругою. Об’єднавши ці два елементи, можна отримати напівсуматор, завданням якого є перенесення числа в поточний або наступний регістр. Напівсуматор передає число в наступний регістр, тільки якщо два входи під напругою, інакше він записує його в поточний регістр.

Піни процесора

Конкретна архітектура 8-бітового процесора та набір його пінів може відрізнятися залежно від моделі. Розглянемо деякі типові піни, які можуть зустрітися на прикладі 8-бітового процесора Intel:

1. Vcc та GND: Це піни живлення. Vcc представляє позитивну напругу живлення (зазвичай 5 В), а GND - заземлення.
2. AD0-AD7: Це шина даних. AD0 до AD7 представляють 8 бітів даних і використовуються для передачі інформації між процесором та пам'яттю або пристроями введення/виведення.
3. A8-A15: Це шина адреси. A8 до A15 вказують адресу в пам'яті, до якої звертається процесор.
4. RD та WR: Ці сигнали керують операціями читання (Read) та запису (Write) в пам'ять або зовнішні пристрой.
5. IO/M: Сигнал, який вказує, чи є поточна операція операцією введення/виведення (I/O) або операцією пам'яті (Memory).
6. ALE (Address Latch Enable): Цей сигнал використовується для активації зовнішніх ліній адреси на шині даних. Це допомагає керувати зовнішніми пристроями під час доступу до пам'яті або введення/виведення.

7. INTR та INTA: Ці піни пов'язані з обробкою переривань. Процесор може бути перерваний зовнішнім пристроєм, і INTA використовується для приймання переривання.

8. CLK (Clock): Цей сигнал представляє тактовий сигнал і керує синхронізацією операцій процесора.

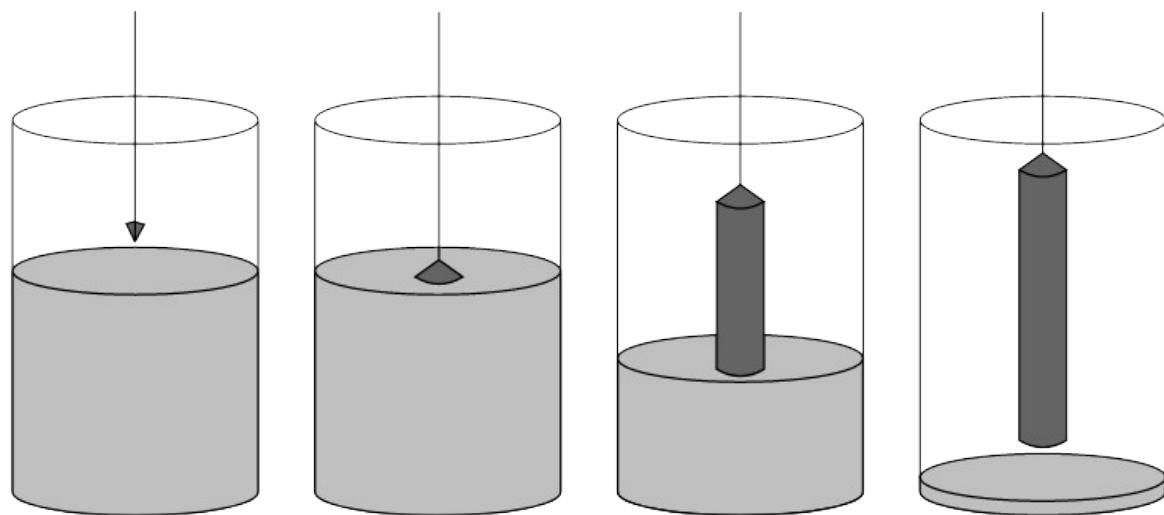
Це лише деякі з пінів, які можна знайти на 8-бітовому процесорі, такому як Intel 8085. Конкретні характеристики та функції можуть відрізнятися для інших моделей 8-бітових процесорів.

Процесори читають дані з пам'яті, виконуючи певні кроки під час своєї роботи. Ось як це відбувається:

1. Встановлення адреси пам'яті: Процесор повинен вказати адресу в пам'яті, звідки він хоче читати дані. Для цього він встановлює значення на відповідній шині адреси (наприклад, A0-A15 в разі 16-бітової шини адреси) через свої адресні піни.
2. Встановлення сигналу читання (Read): Процесор встановлює сигнал читання (наприклад, RD) на відповідний рівень, щоб показати, що він хоче виконати операцію читання з пам'яті.
3. Очікування тактового сигналу: Процесор очікує наступного тактового сигналу (CLK), який зазвичай використовується для синхронізації операцій.
4. Читання даних: Коли сигнал CLK активується в потрібний момент, дані з вказаної адреси в пам'яті передаються на шину даних (наприклад, D0-D7 в разі 8-бітової шини даних). Процесор читає ці дані через свої піни даних.
5. Обробка даних: Процесор отримує прочитані дані і може виконувати різні операції з ними відповідно до поточної інструкції.

Метод Чохральського

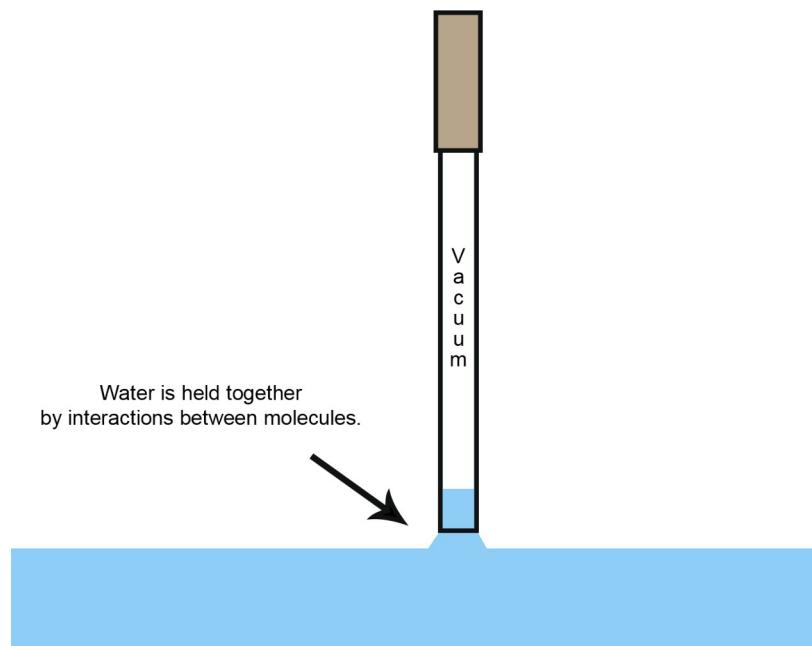
Метод Чохральського — це технологія вирощування кристалів, яка починається з вставки невеликого затравкового кристала в розплав у тиглі, витягуючи затравку вгору, щоб отримати монокристал. Сьогодні цей метод використовується для виробництва різних монокристалів, таких як кремній та германій. Діоксид кремнію (SiO_2 - скло) є електричним ізолятором, на відміну від кремнію (Si), який є напівпровідником. На початку процесу сировина поміщається в тигель циліндричної форми та розплавляється з допомогою опорних або радіочастотних електричних нагрівачів. Після того, як вихідний матеріал повністю розплавлений, затравковий кристал діаметром зазвичай кілька міліметрів занурюють зверху у вільну поверхню розплаву, і невелику частину зануреної затравки розплавляють. На межі контакту між затравкою і розплавом утворюється меніск розплаву. Потім затравку повільно витягають з розплаву і розплав кристалізується на межі розділу, утворюючи нову кристалічну частину. Під час подальшого процесу росту форму кристала, особливо діаметр, контролюють шляхом ретельного регулювання потужності нагріву, швидкості витягування та обертання кристала.



Щось подібне (до методу Чохральського) відбувається, коли взимку утворюються бурульки. Бурулька — це льодовий шип, який утворюється, коли вода, що стікає з предмета, замерзає.

Коли ви поміщаете маленький кристал на поверхню розплавленої кристалічної маси й тому трохи підіймаєте його, тоді між кристалом і поверхнею розплавленого матеріалу утворюється шлях (як міст) розплавленої маси. Коли ця маса остигає, ваш оригінальний кристал подовжується. Це можна продемонструвати з допомогою піпетки та води.

Коли піднести кінець піпетки до поверхні рідини, змочити його, а потім трохи підняти, між ним і поверхнею води утвориться шар води (водяний шлях).



Ян Чохральський (1885 – 1953) був польським хіміком, який винайшов процес Чохральського, який використовується для вирощування монокристалів і виробництва напівпровідникових пластин. Він досі використовується в більш ніж 90% всієї електроніки у світі, яка використовує напівпровідники. Метод Чохральського або процес Чохральського — це метод вирощування кристалів, який використовується для отримання монокристалів напівпровідників (наприклад, кремнію, германію та арсеніду галію), металів (наприклад, паладію, платини, срібла, золота), солей та синтетичних дорогоцінних каменів. Метод названий на честь польського вченого Яна Чохральського, який винайшов метод у 1915 році, досліджуючи швидкість кристалізації металів. Суть свого відкриття Чохральський виклав у статті “Новий метод вимірювання ступеня кристалізації металів” (1918). Зробити це можна з медом. Змочіть ложку в рідкому меді та повільно підніміть її, ви можете побачити цівку (смужку) меду, яка тягнеться від ложки до баночки з медом. Мед також може кристалізуватися завдяки природним цукрам, які він містить. Якби струмінь меду охолоджувати, він затвердів би.

Випрямляюча властивість контакту між мінералом і металом була відкрита в 1874 році Карлом Фердинандом Брауном. Вперше кристали були використані як детектор радіохвиль у 1894 році Джагадішем Чандрою Бозе в його експериментах з мікрохвильовою оптикою. Браун замінив когерер кристалічним детектором, що привело до значного прогресу в чутливості приймача того часу. Контакт між двома різномірними матеріалами на поверхні напівпровідникового кристала детектора утворює неочищений напівпровідниковий діод, який діє як випрямляч, проводячи електричний струм тільки в одному напрямку і протидіючи струму, що протікає в іншому напрямку. Кришталевий детектор складався з електричного контакту між поверхнею напівпровідникового кристалічного мінералу та металом або іншим кристалом (сульфідом свинцю (PbS) або сульфідом кадмію (CdS)). Діод Шотткі (названий на честь німецького фізика Вальтера Шотткі) — це напівпровідниковий діод, утворений з'єднанням напівпровідника з металом. Він має низьке пряме падіння напруги та дуже швидке перемикання. Кристалічний детектор, який використовувався на початку бездротової мережі, можна вважати примітивними діодами Шотткі.

Американський винахідник Лі де Форест (1873 – 1961) розробив тріод (аудіон), який згодом послужив для створення транзистора з напівпровідника в лабораторії Белла за роботами Джона Бардіна і Вільяма Шоклі (у 1956 році Бардін і Шоклі отримали Нобелівську премію “за дослідження напівпровідників та відкриття транзисторного ефекту”).

Планарна технологія

Планарна технологія в електроніці вказує на техніку виготовлення мікросхем, де всі компоненти і провідники знаходяться на одній плоскості на поверхні напівпровідникового матеріалу (зазвичай кремнію). Це включає в себе виготовлення транзисторів, конденсаторів і резисторів на одному шарі матеріалу.

Однією з ключових подій у розвитку планарної технології був винахід Джека Кілбі в 1958 році, коли він розробив першу інтегральну схему, в якій всі компоненти були розташовані на одному плоскому чіпі кремнію. Це важливо для виробництва компактних і надійних електронних пристройів.

Процес планарної технології виготовлення мікросхем включає кілька ключових кроків. Нижче наведено загальний опис процесу:

1. Початкова підготовка:

- Вибір напівпровідникового матеріалу, зазвичай кремнію, який є основним матеріалом для виробництва мікросхем.
- Початкова обробка кремнію для створення плоскої основи для майбутніх компонентів.

2. Формування оксидного шару:

- Нанесення тонкого оксидного шару (зазвичай діоксиду кремнію) на поверхню кремнієвої пластини. Цей шар використовується як діелектрик.

3. Формування полімерного шару:

- Нанесення фоточутливого полімерного шару на поверхню оксидного шару.

4. Фотолітографія:

- Використання світлоочутливого полімерного шару для створення маски зображення, яка визначає області, де будуть розміщені компоненти.

- Експонування полімеру до світла через маску, яка передає певний зразок на поверхню.

Використання маски та світлоочутливого шару для створення зразка на фоточутливому шарі.

5. Етап етчінгу:

- Використання хімічних реакцій для видалення непокритих частин оксидного шару, залишаючи тільки ті області, де є світлоочутливий полімер. Впровадження хімічного процесу для видалення частинок шару, де фоточутливий матеріал був викритий світлу. Це визначає місця, де будуть розташовані компоненти.

6. Дифузія/іонна імплантация:

- Додавання або видалення домішок в областях кремнію для створення напівпровідниківих елементів (наприклад, транзисторів).

7. Металізація:

- Нанесення тонкого металевого шару на поверхню для створення провідників, які будуть з'єднувати компоненти.

8. Формування захисного шару:

- Покриття всієї структури захисним шаром для запобігання пошкодженням і забезпечення стійкості мікросхеми.

9. Тестування та розрізання:

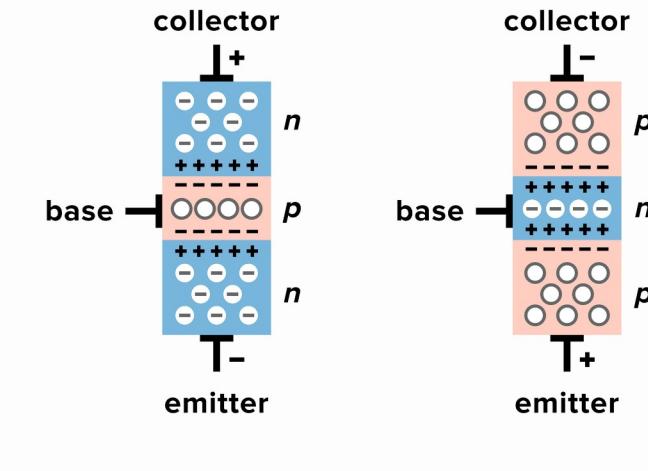
- Проведення тестів для перевірки працездатності мікросхеми.
- Розрізання великої кремнієвої пластини на окремі мікросхеми.

Цей загальний процес може включати різні етапи та технології в залежності від конкретного дизайну та виробничих можливостей.

Електронні компоненти

Біполярний транзистор

Транзистори використовуються для створення логічних вентилів (AND, OR, NOT), які в свою чергу використовуються для реалізації логічних операцій в комп'ютерних пристроях.



Біполярний транзистор (БТ) — це електронний прилад, який складається з двох типів напівпровідникового матеріалу (зазвичай, кремнію) і зазвичай використовується як елемент для підсилення або комутації сигналів в електронних пристроях.

Біполярні транзистори можна розділити на два основні типи: NPN (негативно-позитивно-негативний) і PNP (позитивно-негативно-позитивний). Кожен транзистор складається з трьох областей: емітера, бази та колектора.

1. Емітер (Emitter): Це одна з областей транзистора, з якої основні носії заряду (електрони для NPN або дірка (квазічастинка) для PNP) виходять у вакуум або напівпровідник.
2. База (Base): Це область, яка керує потоком основних носіїв заряду між емітером і колектором. Зміна напруги або струму на базі може контролювати електричний струм в колекторі.
3. Колектор (Collector): Це область, де основні носії заряду поглиблюються після того, як пройшли через базу. Колектор збирає основні носії заряду і створює вихідний струм транзистора.

Робота біполярного транзистора базується на принципі контролю струму в базі для впливу на струм між емітером і колектором. Коли транзистор працює в режимі підсилення, невеликий струм, що подається на базу, може призводити до великого струму в колекторі, забезпечуючи підсилення сигналу.

Біполярні транзистори широко використовуються в різноманітних електронних пристроях, включаючи підсилювачі, осцилятори, інтегральні схеми та інші. Вони є важливими компонентами багатьох сучасних електронних пристрій і забезпечують надійну та ефективну роботу цих пристрій.

Біполярний транзистор був винаходом американських вчених Джона Бардіна, Вільяма Бреттона та Вільяма Шоклі у 1947 році. Вони працювали в лабораторії Bell Telephone Laboratories, що є дослідницьким підрозділом компанії AT&T.

Винахід транзистора виявився революційним для електроніки, оскільки цей пристрій замінив великий і неефективні лампи, які використовувалися раніше в електронних пристроях. Транзистор став основою для розвитку нових технологій, включаючи інтегральні схеми та мікропроцесори, і відіграв важливу роль у розвитку сучасної електроніки.

Напівпровідник — це матеріал, який має провідність між металами (які мають високу провідність) і непровідниками (які мають низьку провідність). Головна особливість напівпровідників полягає в тому, що їхні властивості провідності можна легко контролювати або змінювати, зазвичай за допомогою домішок або зовнішнього електричного поля.

Однією з найважливіших властивостей напівпровідників є їхній здатність функціонувати як напівпровідникові прилади, такі як транзистори, діоди та інші електронні компоненти. Це робить їх незамінними для виробництва інтегральних схем, які використовуються у багатьох електронних пристроях, включаючи комп'ютери, телефони, телевізори та інші.

Одним із найвідоміших напівпровідників є кремній (Si, силіцій), який широко використовується в електронній промисловості. Інші напівпровідники включають германій, галій, арсенід галію та інші матеріали.

Тріод

Тріод — це електровакуумна лампа, яка складається з трьох основних елементів: катоду, аноду і сітки (електрода, що контролює потік електронів між катодом і анодом). Тріод використовується для підсилення або контролю сигналів у електричних колах. У 1906 році американський інженер Лі де Форест увів в лампу третій електрод — сітку (і, таким чином, створивши тріод).

Катод випромінює електрони, а анод притягує їх, створюючи електричний струм від катода до анода. Сітка, розташована між катодом і анодом, контролює потік електронів, що прямують від катода, і таким чином регулює струм в електричному колі. Зміна напруги на сітці може контролювати величину струму, що проходить через лампу, що дозволяє використовувати тріод як підсилювач сигналу в аналогових електричних лінійних схемах.

У вакуумних електронних лампах термоелектронна емісія відіграє важливу роль, особливо в катодах цих ламп. Основним компонентом лампи, де використовується термоелектронна емісія, є катод.

Катод у вакуумних лампах часто виготовляється з матеріалів, які добре вивільняють електрони при підвищенні температури. Зазвичай використовують метали, такі як вольфрам. Катод нагрівається до високої температури, і через термоелектронну емісію з його поверхні зриваються електрони.

Отримані електрони утворюють електронний потік, який може бути контролюваний за допомогою інших елементів лампи, таких як анод і сітка. Подавання напруги на сітку дозволяє контролювати потік електронів і, таким чином, впливати на роботу лампи як засіб зміни амплітуди або інших характеристик електричного сигналу.

Термоелектронна емісія — це явище, коли електрони зриваються з поверхні матеріалу при підвищенні його температури. Це відбувається через те, що теплова енергія збільшує кінетичну енергію частинок в матеріалі, що може привести до виходу електронів з його поверхні.

Процес термоелектронної емісії часто використовується в електроніці, зокрема в катодах електронних ламп та інших електронних пристроях. У термоелектронних приладах електрони видаляються з поверхні нагрітого катода та утворюють електронний потік. Цей принцип використовується, наприклад, у вакуумних тріодах, випромінювальних приладах та інших електронних компонентах.

За дослідження термоелектронної емісії у 1928 Овен Вільямс Річардсон отримав Нобелівську премію з фізики.

Кварцовий резонатор

Кварцеві кристали здатні виражати точний резонансний часовий сигнал. Це дозволяє їм використовуватися як елементи генераторів часу для синхронізації роботи електронних пристрій. У комп'ютерах кварцеві резонатори використовуються для створення основного тактового сигналу, який синхронізує внутрішні операції.

Кварцовий резонатор — це пристрій, який використовує властивості кварцового кристалу для стабілізації частоти. Кварц — це кристалічний матеріал, який володіє п'єзоелектричним ефектом, що означає, що він може генерувати електричні сигнали при зміні форми кристалу.

Механізм роботи кварцовового резонатора ґрунтуються на тому, що кварцовий кристал має природну резонансну частоту, яка залежить від його геометрії та фізичних властивостей. Коли кварцевий кристал піддається електричному зміщенню, відбувається механічне коливання кристалу, і це коливання стає стабільним і точним джерелом частоти.

П'єзоелектричний ефект (або п'єзоефект) — це фізичний ефект, який полягає в зміні електричної поляризації деяких матеріалів при деформації. Це означає, що при застосуванні механічного навантаження до такого матеріалу відбувається зміна його електричної поляризації, що призводить до виникнення електричної напруги.

Кварц є прикладом матеріалу, який виявляє п'єзоелектричний ефект. Коли на кварцевий кристал застосовується механічне навантаження, відбувається деформація кристалічної структури, що призводить до виникнення електричної напруги на його поверхні.

П'єзоелектричний ефект був відкритий французьким фізиком та математиком П'єром Кюрі та його братом Жаком Кюрі у 1880 році. Вони виявили, що певні кристали, такі як кварц, генерують електричний заряд при механічному деформуванні.

FPGA

Програмована користувачем вентильна матриця, або FPGA, є інтегральною схемою, яка може бути програмована для виконання будь-яких логічних функцій. Основна відмінність FPGA від звичайних мікросхем (наприклад, мікроконтролерів або процесорів) полягає в тому, що логіку FPGA можна перепрограмувати на рівні вентилів та зв'язків між ними.

Програмована користувачем вентильна матриця, ПКВМ (Field-Programmable Gate Array, FPGA) — напівпровідниковий пристрій, що може бути налаштований виробником або розробником після виготовлення. ПКВМ програмуються шляхом зміни логіки роботи принципової схеми, наприклад, за допомогою вихідного коду мовою проектування (типу VHDL), на якому можна описати цю логіку роботи мікросхеми. ПКВМ є однією з архітектурних різновидів програмованих логічних інтегральних схем (ПЛІС).

FPGA зазвичай використовуються для обробки сигналів, мають більше логічних елементів і гнучкішу архітектуру, ніж CPLD. Програма для FPGA зберігається в розподіленій пам'яті, яка може бути виконана як на основі енергозалежних осередків статичного ОЗП (Оперативний запам'ятовуючий пристрій) — у цьому випадку програма не зберігається при зникненні електроживлення мікросхеми, так і на основі енергонезалежних комірок Flash-пам'яті, що дозволяє програмі зберігатися при зникненні електроживлення. Якщо програма зберігається в енергозалежній пам'яті, то при кожному ввімкненні живлення мікросхеми її необхідно заново конфігурувати за допомогою початкового завантажувача, який може бути вбудовано і в саму FPGA.

Що таке реляційні бази даних?

Англійський вчений-інформатик Едгар Кодд (1923 – 2003) в 1969 році описав реляційну базу даних.

Реляційна база даних заснована на відношеннях (relations) і зв'язках (relationships) (або таблицях і зв'язках), як це визначив Едгар Кодд.

Основна суть реляційних баз даних в тому, що вони базуються на структурі, яка називається "відношення". "Відношення", або "релейшн", є основною структурою реляційних баз даних, власне, бази даних складаються з "відношень" (таблиць) та умовних зв'язків між ними. "Відношення", вона ж славнозвісна "таблиця", являє собою впорядковану множину (кортеж), тобто асоціативний масив. "Відношення" (таблицю), яке зберігатиме ім'я користувачів, можна записати так: $\text{Users} = \{\{\langle\text{name}, \text{John}\rangle, \langle\text{surname}, \text{Smith}\rangle\}\}$.

Якщо більше ніж один користувач тоді можна записати так: $\text{Users} = \{\{\langle\text{name}, \text{John}\rangle, \langle\text{surname}, \text{Smith}\rangle\}, \{\langle\text{name}, \text{Jack}\rangle, \langle\text{surname}, \text{White}\rangle\}\}$. Суть такого підходу в тому, що база даних має плоску структуру. Все що зберігає комірка таблиці (відношення) інтерпретується як значення саме по собі, а не як структура.

Реляційна база даних (Relational Database Management System, RDBMS) — це тип бази даних, який ґрунтуються на реляційній моделі даних. Реляційна модель даних використовує таблиці, які складаються з рядків і стовпців для зберігання та організації даних. Кожна таблиця в реляційній базі даних має набір стовпців, які представляють атрибути даних, і рядки, які представляють конкретні записи або кортежі.

Бази даних, які не є нормалізованими, не відповідають принципам ACID та не основані на рядках (rows) не є реляційними. Існують бази даних основані на колонках (Apache Cassandra).

Основні поняття реляційної бази даних включають такі елементи:

1. Таблиці: Вони представляють собою сутності або класи об'єктів, для яких зберігаються дані. Кожен рядок таблиці містить конкретний запис, а кожен стовпець таблиці відповідає певному атрибуту цих записів.
2. Ключі: Реляційні бази даних використовують ключі для ідентифікації унікальних записів в таблиці. Первинний ключ (Primary Key) індексує унікальні значення і дозволяє швидкий доступ до конкретних записів. Існують також інші види ключів, наприклад, зовнішні ключі (Foreign Key), які встановлюють зв'язки між таблицями. Таблиця може мати лише один первинний ключ; а в таблиці цей первинний ключ може складатися з одного або кількох стовпців (полів).
3. Запити: Ви взаємодієте з реляційною базою даних за допомогою мови запитів, такої як SQL (Structured Query Language). За допомогою SQL ви можете вставляти, оновлювати, видаляти та витягувати дані з таблиць.
4. Нормалізація: Це процес організації даних у таблицях для запобігання зайвій реплікації та забезпечення цілісності даних. Нормалізація допомагає уникнути аномалій та забезпечує ефективний доступ до даних.
5. Транзакції: Реляційні бази даних підтримують транзакції, які є атомарними операціями, що забезпечують цілісність даних. Транзакції визначаються ключовими властивостями ACID (Atomicity, Consistency, Isolation, Durability).

6. Запити з об'єднаннями: Реляційні бази даних дозволяють виконувати запити, які об'єднують дані з різних таблиць на основі зв'язків між ними. Це дозволяє витягувати складні набори даних.

Реляційні бази даних є дуже поширеними та використовуються в багатьох сферах, включаючи бізнес, веброзробку, наукові дослідження, data science, та інші області, де потрібно ефективно зберігати та керувати структурованими даними. MySQL, PostgreSQL, Oracle Database, та Microsoft SQL Server - це лише кілька прикладів реляційних систем управління базами даних.

Нормалізація баз даних — це процес представлення бази даних з точки зору відносин у стандартних нормальних формах, де перша нормальна є мінімальною вимогою.

Перша нормальна форма (1НФ, 1NF) — це властивість відношення у реляційній базах даних. Відношення знаходиться в першій нормальній формі тоді й тільки тоді, коли домен (область значень) кожного атрибута містить лише нероздільні значення, а значення кожного атрибута містить лише одне значення з цього домену.

Наприклад, якщо у вас є таблиця "Клієнти" з атрибутом "Телефонні номери", в якому зберігається список телефонних номерів клієнта, то це не задовільняє умовам 1НФ. Для досягнення 1НФ, кожен телефонний номер має бути окремим атрибутом, а не списком в одному атрибуті.

У контексті баз даних, термін "домен" вказує на множину всіх можливих значень, які може приймати атрибут. Іншими словами, домен — це набір допустимих значень для конкретного атрибута в межах реляційної таблиці.

Наприклад, якщо у вас є атрибут "Вік" в таблиці "Користувачі", то домен цього атрибута, наприклад, множина натуральних чисел.

ACID (Atomicity, Consistency, Isolation, Durability) — це набір властивостей, які характеризують транзакції в реляційних базах даних і забезпечують надійність та цілісність даних під час операцій з базою даних. Ось їх означення:

Atomicity (Атомарність): Ця властивість вказує на те, що транзакція є атомарною, тобто вона виконується як єдина неділіма одиниця. Це означає, що усі операції в межах транзакції виконуються повністю або анульюються. Якщо будь-яка операція транзакції не вдалася, то всі інші операції транзакції скасовуються, і база даних залишається в стані, яким була перед початком транзакції.

Consistency (Спрямованість): Ця властивість гарантує, що транзакція переводить базу даних із одного стабільного стану в інший стабільний стан. Тобто база даних залишається в суворому стані цілісності, де всі обмеження і правила цілісності зберігаються після завершення транзакції.

Isolation (Ізоляція): Ця властивість забезпечує те, що кожна транзакція виконується відокремлено від інших транзакцій. Це означає, що операції, які виконуються в межах однієї транзакції, не впливають на операції інших транзакцій. Ізоляція допомагає уникнути конфліктів і забезпечує безпеку паралельних транзакцій.

Durability (Стійкість): Ця властивість гарантує, що результати виконаних транзакцій залишаються стійкими та не втрачаються в разі аварійного відключення або перезавантаження системи. Іншими словами, після завершення транзакції її результати зберігаються навіть в умовах втрати живлення або інших непередбачуваних подій.

Забезпечення цих ACID-властивостей важливе для забезпечення надійності та цілісності даних в реляційних базах даних, особливо в ситуаціях, де важливо мати точне та надійне управління даними, таких як банківські операції, системи керування запасами, облік замовлень тощо.

Теорема CAP (Consistency, Availability, Partition Tolerance) є фундаментальним поняттям у галузі розподілених систем та баз даних. Ця теорема стосується можливостей розподілених систем та визначає, які характеристики можуть бути одночасно забезпечені в таких системах.

Основні компоненти теореми CAP:

Consistency (Узгодженість): Це вимога до системи, яка стверджує, що кожний запит до розподіленої системи повинен повернати однаковий результат, якщо запити здійснюються в однаковий час. Іншими словами, всі операції читання повинні бачити однакові дані.

Availability (Доступність): Це вимога, яка стверджує, що кожний запит до системи повинен завершитися успішно (без помилок) і без затримок. Система завжди повинна бути доступною для використання.

Partition Tolerance (Стійкість до розділення): Це вимога, яка враховує можливість втрати зв'язку між частинами розподіленої системи (розділення мережі або інші подібні проблеми). Система повинна продовжувати працювати навіть при виникненні таких роздіlenь.

Теорема CAP стверджує, що в розподілених системах можливо забезпечити одночасно не більше двох з трьох вказаних вимог. Іншими словами:

Якщо система забезпечує узгодженість і доступність, вона може не бути стійкою до розділення мережі (CP).

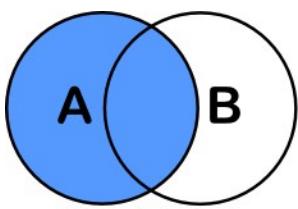
Якщо система забезпечує доступність і стійкість до розділення, вона може не забезпечувати узгодженість (AP).

Якщо система забезпечує узгодженість і стійкість до розділення, вона може не бути завжди доступною (CA).

Ця теорема допомагає вибирати правильний баланс між цими вимогами в залежності від конкретних вимог і обмежень вашого додатку чи системи.

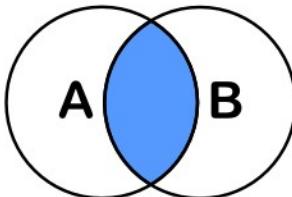
Діаграми Венна — це візуальний спосіб представлення взаємодії між множиною об'єктів чи множинами даних. У контексті баз даних, діаграми Венна можуть бути використані для відображення відносин між різними підмножинами даних чи таблиць. Це допомагає з'ясувати, які об'єкти належать до різних груп та чи є у них спільні елементи.

Ви можете використовувати запити SQL для виявлення спільних записів у різних таблицях бази даних та відображення їх результатів у вигляді діаграми Венна. Наприклад, ви можете використовувати операції об'єднання (UNION), перетину (INTERSECT) та різниці (EXCEPT) для цього.

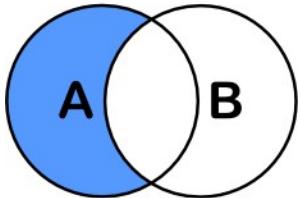


```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```

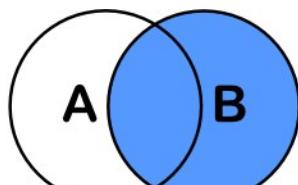
CHEATSHEET
**SQL
JOINS**



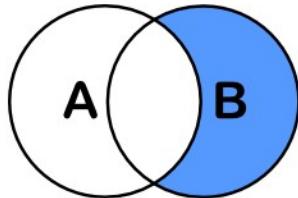
```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```



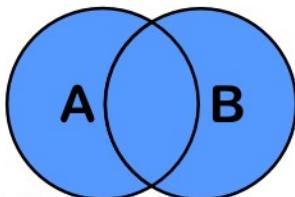
```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```



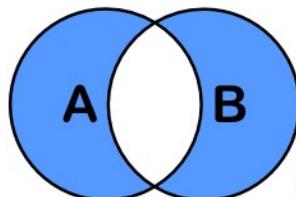
```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```



```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
```



```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
```



```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

SQL запити

SQL (Structured query language — мова структурованих запитів) — декларативна мова запитів для взаємодії користувача з базами даних.

Створити таблицю

```
-- Create a new database (if not exists)
CREATE DATABASE IF NOT EXISTS LibraryDatabase;
```

```
-- Switch to the newly created database
USE LibraryDatabase;
```

```
-- Create a table named 'Books'
```

```
CREATE TABLE IF NOT EXISTS Books (
    BookID INT PRIMARY KEY,
    Title VARCHAR(255) NOT NULL,
    Author VARCHAR(100) NOT NULL,
    PublicationYear INT
);
```

Видалити таблицю

```
DROP TABLE IF EXISTS LibraryDatabase.Books;
```

Перейменувати таблицю

```
-- Rename the 'Books' table to 'NewBooks' in the 'LibraryDatabase'
ALTER TABLE LibraryDatabase.Books
RENAME TO NewBooks;
```

Додати атрибут (стовпець)

```
-- Add a new attribute 'Genre' to the 'Books' table in the 'LibraryDatabase'
ALTER TABLE LibraryDatabase.Books
ADD COLUMN Genre VARCHAR(50);
```

```
-- Remove the 'ColumnName' column from the 'YourTable' table
```

```
ALTER TABLE YourTable
DROP COLUMN ColumnName;
```

Додати запис

```
-- Insert a new entry into the 'Books' table
```

```
INSERT INTO LibraryDatabase.Books (BookID, Title, Author, PublicationYear, Genre)
VALUES (1, 'The Great Gatsby', 'F. Scott Fitzgerald', 1925, 'Fiction');
```

Вибрать запис

-- Select a specific entry

```
SELECT Title, Author  
FROM LibraryDatabase.Books  
WHERE PublicationYear > 2000;
```

-- IS LIKE

```
SELECT * FROM your_table  
WHERE your_column LIKE 'A%';
```

Оновити запис

```
-- Update the publication year of a book with BookID 1  
UPDATE LibraryDatabase.Books  
SET PublicationYear = 2020  
WHERE BookID = 1;
```

Видалити запис

```
-- Delete the entry with BookID 1 from the 'Books' table  
DELETE FROM LibraryDatabase.Books  
WHERE BookID = 1;
```

Вибрати все

```
-- Select all entries from the 'Books' table  
SELECT *  
FROM LibraryDatabase.Books  
WHERE BookID = 1;
```

Підрахувати записи

```
-- Count the number of entries in the 'Books' table  
SELECT COUNT(*) AS NumberOfEntries  
FROM LibraryDatabase.Books;
```

Also

```
SELECT SUM(column) FROM table;  
SELECT AVG(column) FROM table;  
SELECT MIN(column) FROM table;  
SELECT MAX(column) FROM table;
```

GROUP BY

GROUP BY Clause: Used in conjunction with aggregate functions, the GROUP BY clause groups rows based on the values in specified columns.

```
SELECT column1, COUNT(*)
```

```
FROM table  
GROUP BY column1;
```

Вибір з кількох таблиць

```
-- Select title, author, and genre from the 'Books' and 'Authors' tables  
SELECT Books.Title, Authors.AuthorName, Books.Genre  
FROM LibraryDatabase.Books  
INNER JOIN LibraryDatabase.Authors ON Books.AuthorID = Authors.AuthorID;
```

Упорядкуйте записи

ORDER BY Clause: The ORDER BY clause is used to sort the result set based on one or more columns, either in ascending (ASC) or descending (DESC) order.

```
SELECT column1, column2  
FROM table  
ORDER BY column1 ASC;
```

LIMIT and OFFSET

LIMIT and OFFSET Clauses: Used for pagination, the LIMIT clause restricts the number of rows returned, and the OFFSET clause specifies the starting point.

```
SELECT column1, column2  
FROM table  
LIMIT 10 OFFSET 20;
```

Додати індекс

```
-- Create a single-column index  
CREATE INDEX index_name ON table_name (column_name);
```

У SQL транзакціями зазвичай керують за допомогою операторів BEGIN TRANSACTION, COMMIT і ROLLBACK. Ось простий приклад:

```
-- Start a transaction  
BEGIN TRANSACTION;  
  
-- SQL statements within the transaction  
UPDATE table1 SET column1 = value1 WHERE condition;  
INSERT INTO table2 (column1, column2) VALUES (value1, value2);  
  
-- Check if everything is fine, then commit the transaction  
COMMIT;  
  
-- If something went wrong, roll back the transaction  
ROLLBACK;
```

Що таке модель “сутність — зв'язок”?

Entity-Relationship Model (ER Model, модель “сутність-зв'язок”) — це концептуальна модель, яка використовується в області баз даних для опису даних та їхніх взаємозв'язків в інформаційних системах. ER-модель допомагає визначити сутності (entities), атрибути (attributes) і зв'язки (relationships) між цими сутностями. Це важлива складова процесу проєктування баз даних.

Основні концепції ER-моделі включають наступне:

1. Сутності (Entities): Сутності — це об'єкти або предмети в реальному світі, які ми бажаємо включити до бази даних. Кожна сутність має свій унікальний ідентифікатор (ключ) і може мати атрибути, що описують цю сутність.
2. Атрибути (Attributes): Атрибути — це характеристики або властивості сутностей. Вони використовуються для подальшого опису сутностей. Наприклад, якщо сутністю є "клієнт", то її атрибутами можуть бути ім'я, адреса, номер телефону і так далі.
3. Зв'язки (Relationships): Зв'язки показують, як сутності пов'язані одна з одною. Наприклад, зв'язок "має" може з'єднувати сутності "клієнт" і "замовлення", щоб вказати, які клієнти зробили які замовлення.

ER-модель зазвичай відображається за допомогою діаграм ERD (Entity-Relationship Diagram), яка містить сутності, атрибути та зв'язки між ними у вигляді графічних об'єктів.

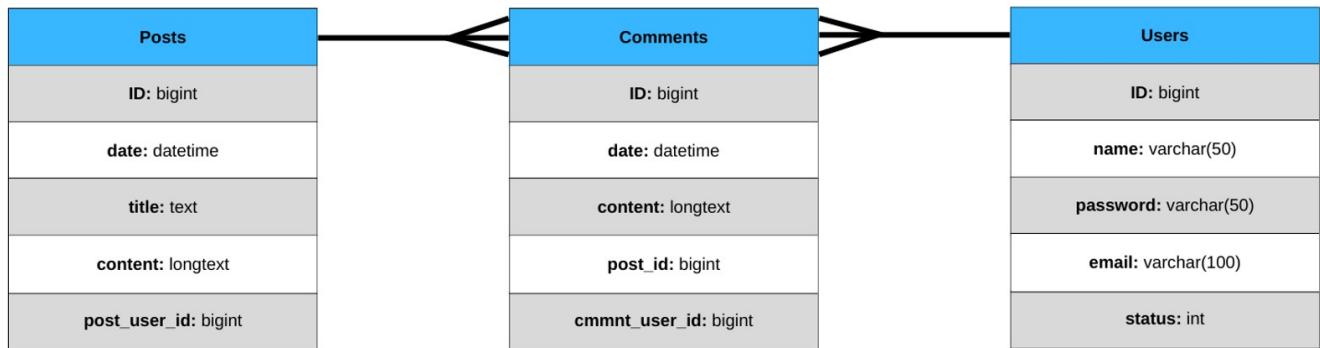
ER-модель є важливим інструментом для розробників баз даних, оскільки вона допомагає визначити структуру даних, їхні зв'язки і правила цілісності. Це полегшує створення ефективних баз даних, які можуть ефективно зберігати, оновлювати та вибирати інформацію для подальшого використання в інформаційних системах.

В ER-моделі існує кілька типів відносин (зв'язків), які використовуються для моделювання взаємозв'язків між сутностями. Основні типи відносин включають наступне:

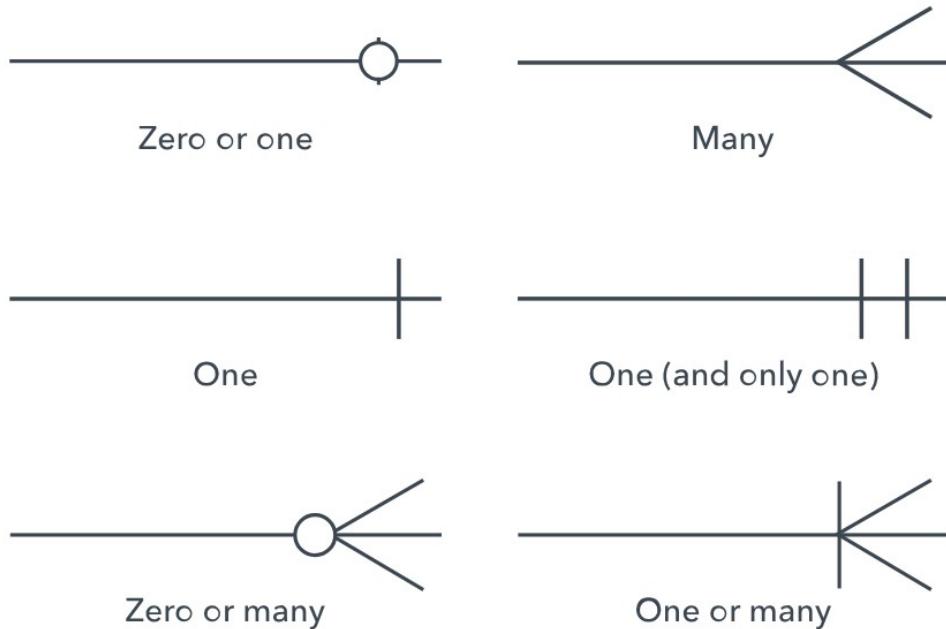
1. Один-до-одного (One-to-One): В цьому типі відносин одна сутність пов'язана з однією іншою сутністю. Наприклад, один клієнт може мати лише один рахунок, і навпаки.
2. Один-до-багатьох (One-to-Many): У цьому відношенні одна сутність пов'язана з багатьма іншими сутностями. Наприклад, один викладач може мати багато студентів, але кожен студент має тільки одного викладача.
3. Багато-до-одного (Many-to-One): Це відношення обернене до попереднього. Багато сутностей пов'язані з однією іншою сутністю. Наприклад, багато студентів можуть належати до одного факультету.
4. Багато-до-багатьох (Many-to-Many): В цьому типі відношенні багато сутностей пов'язані з багатьма іншими сутностями. Це відношення моделюється за допомогою додаткової таблиці, яка містить пари ідентифікаторів зв'язаних сутностей. Наприклад, багато студентів може бути в багатьох курсах, і багато курсів можуть мати багато студентів.
5. Самовідношення (Self-Relationship): У цьому відношенні сутність пов'язана з самою собою. Наприклад, модель користувачів соціальної мережі, де користувачі можуть бути друзями один одного.

Ці типи відносин використовуються для визначення того, як сутності взаємодіють між собою в базі даних. Коректне визначення цих відносин допомагає створювати ефективні та структуровані бази даних для зберігання та обробки інформації.

Entity-Relationship Model блога з постами (дописами). Один допис (post) може мати багато коментарів (comment), але один коментар тільки один допис. Один користувач (user) може мати багато коментарів, але один коментар тільки одного користувача, тобто автора.



Три основні зв'язки: один-до-одного, один-до-багатьох і багато-до-багатьох. Прикладом один-до-одного може бути один користувач, пов'язаний з однією поштовою адресою. Приклад “один-до-багатьох” (або “багато-до-одного”, залежно від напрямку зв'язку): книга має багато сторінок, але кожна сторінка належить тільки одній книзі. Приклад багато-до-багатьох: студенти як група пов'язані з декількома викладачами, а викладачі, у свою чергу, пов'язані з кількома студентами.



Що таке теорема CAP?

CAP-теорема (КЕП-теорема) в інформаційних технологіях визначає, що в розподілених системах неможливо забезпечити одночасно всі три наступні властивості:

Consistency (Узгодженість): Всі копії даних в системі показують однакові дані в одинаковий час, без розходження.

Availability (Доступність): Кожен запит на доступ до системи завжди отримує відповідь (успіх чи невдача) без відкладань.

Partition Tolerance (Тolerантність до розділення): Система може продовжувати працювати, навіть якщо вона розділена на частини через відмови в мережі. Тolerантність до розділення вказує на здатність розподіленої системи продовжувати працювати навіть у випадку, коли мережа досить часто розділяється або виникають збої зв'язку між складовими частинами системи. Це означає, що система може функціонувати і надалі з певними обмеженнями чи може працювати в обмеженому режимі, але не повністю виходить з ладу або втрачачі доступність для користувачів під час розділення мережі або інших аналогічних проблем.

CAP-теорема важлива в розробці розподілених систем і дуже впливає на рішення, які приймають розробники під час розробки таких систем. Вона застосовується в різних областях, включаючи:

- Бази даних:** Визначення того, як система зберігає і обробляє дані в розподіленому середовищі.
- Хмарні обчислення:** Вибір між різними конфігураціями та рівнями доступності в хмарних сервісах.
- Мережеві протоколи та архітектури:** Вплив CAP-теореми на вибір протоколів та архітектур для розподілених систем.
- Фінансові та торговельні системи:** Забезпечення консистентності та доступності в системах, які обробляють велику кількість транзакцій.
- Інтернет речей (IoT):** Управління та обробка даних в розподілених мережах IoT.

У кожному конкретному випадку розробники повинні зважити на вимоги системи та вибрати правильний баланс між цими трьома властивостями з урахуванням потреб користувачів та особливостей вимог до системи.

У більшості блокчайн систем неможливо досягнути одночасно всі три властивості CAP-теореми. Так, наприклад, багато блокчайн мереж приділяють велику увагу узгодженості та відміряній стійкості, тобто гарантують, що дані на розподілених вузлах є узгодженими та мережа стійка до розбиття. Однак це може призводити до обмеження доступності, особливо при великому навантаженні.

За словами комп’ютерного вченого з Каліфорнійського університету Берклі Еріка Брюера, ця теорема вперше з’явилася восени 1998 року. У 1999 році вона була опублікована як принцип CAP і представлена Брюером як припущення. У 2002 році Сет Гілберт і Ненсі Лінч з Массачусетського технологічного інституту опублікували формальний доказ гіпотези Брюера, перетворивши її на теорему.

Що таке блок-схема?

Блок-схема — представлення алгоритму розв'язування або аналізу задачі за допомогою геометричних елементів (блоків), які позначають операції, потік, дані тощо. (Блок-схеми описані в “Енциклопедії кібернетики” Глушкова в 1973 році. Стандартизовані в 1985 як ISO 5807).

Основні елементи



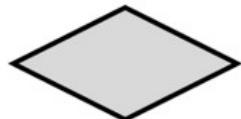
Овал представляє початкову або кінцеву точку



Паралелограм представляє вхідні або вихідні дані



Прямоугольник представляє процес



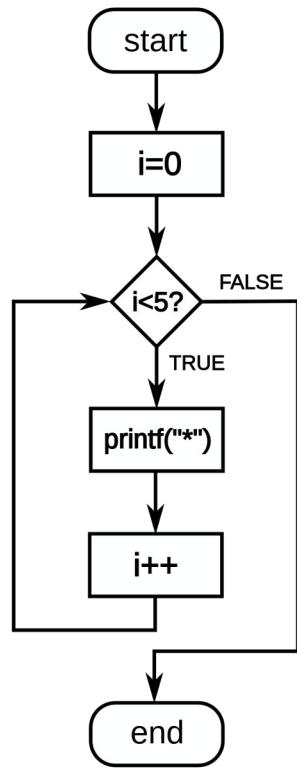
Ромб вказує на умовний оператор



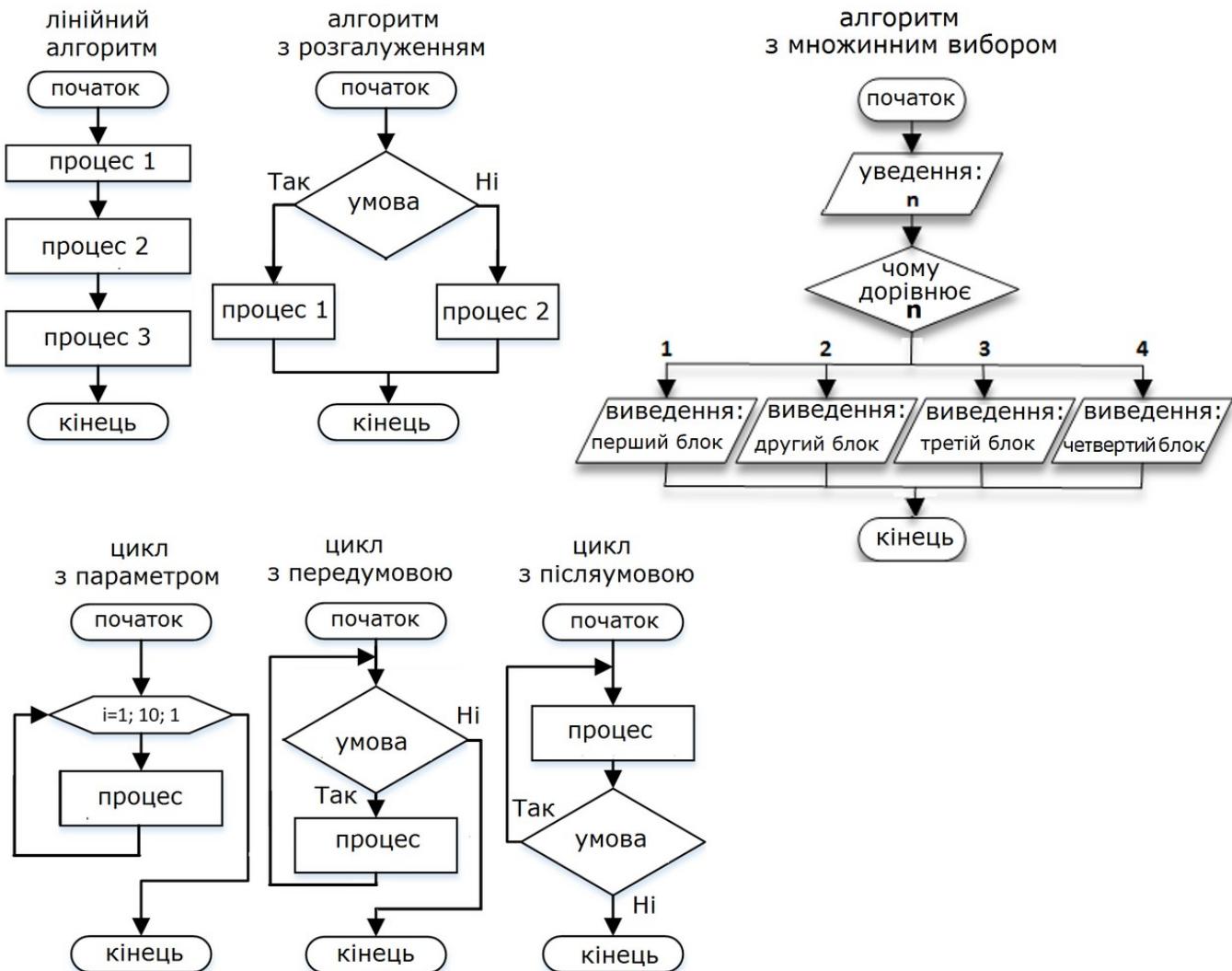
Стрілка показує зв'язки між фігурами

Проста програма у вигляді блок-схеми

```
for(var i=0;i<5;i++)  
    console.log("*");
```



Різні типи блок-схем



Математично, **цикломатична складність** структурної програми визначається за допомогою орієнтованого графу утвореного базовими блоками програми, з ребрами між двома базовими блоками якщо керування може бути передане від одного до другого (граф потоку керування програми). Наприклад, якщо початковий код не містить місць прийняття рішень таких як IF тверджень або FOR циклів, складність дорівнює 1, через наявність лише одного шляху в коді.

Що таке комп'ютерна мережа?

Комп'ютерна мережа — це система взаємозв'язаних комп'ютерів та інших пристройів (наприклад, принтерів), які з'єднані між собою з метою обміну даними, ресурсами та послугами. Ці мережі можуть бути локальними (LAN), що охоплюють невелику територію, таку як офіс або будинок, або ж широкими (WAN), які можуть охоплювати великі географічні області, наприклад, між містами чи країнами. Комп'ютерні мережі дозволяють об'єднувати ресурси, полегшують спільну роботу користувачів і забезпечують можливість комунікації та обміну інформацією.

Код Морзе

Американський художник і винахідник Семюель Морзе (1791 - 1872) розробив електричний телеграф і азбуку Морзе для нього. Принцип дії телеграфу простий: між точками А і В прокладено електричний провід, отже, якщо в точці А хтось подав електричний сигнал на дріт, цей сигнал майже зі швидкістю світла проходив по дроту до точки В, де його можна виявити. Якщо розробити спеціальний алфавіт, де є лише тире, крапки та пробіли, то з допомогою нього можна передавати повідомлення телеграфом, а саме: крапка — це електричний сигнал на короткий час, тире — на довший час, а час між ними — це відсутність сигналу на проводі. Семюель Морзе розробив такий телеграф, але була ще одна проблема, яка перешкоджала комерційному успіху телеграфу, а саме, якщо провід був дуже довгий між точками приймання і передачі сигналу, то електричний сигнал був ослаблений через електричний опір дроту (закон Ома). Потрібно було придумати, як посилити сигнал. Для посилення сигналу Морзе використав реле, яке винайшов американський вчений Джозеф Генрі (1797 - 1878). Просте електричне реле складається з електромагнітної котушки (ізольований дріт, обмотаний навколо стрижня), яка при подачі електрики може притягувати метал, тому його можна використовувати для замикання металевих контактів. Якщо на реле подається струм, то воно замикає контакт, якщо струму немає, то воно в розімкнутому стані. Таким чином, Морзе встановив точки посилення сигналу на довгих телеграфних шляхах. Електричний сигнал (повідомлення) надійшовши у точку посилення сигналу і потрапивши на реле, яке скопіювало цей сигнал в інший електричний ланцюг, в якому посилювався сигнал. Перше телеграфне повідомлення: “що Бог творитиме” (Числа 23:23), правда англійською, було передано Семюелем Морзе в 1844 році. Морзе розбагатів на своєму винаході, який поширився в Англії, Німеччині, Франції, Росії.

Фізик Ганс Ерстед виявив, що дріт, по якому протікає електричний струм, може відхилити металеву стрілку. Таким чином, котушка дроту може працювати як магніт. На цьому принципі засновані реле, які використовуються в телеграфі. Але були розроблені й так звані стрілчасті телеграфи, в яких передані повідомлення не видавалися у вигляді звукових сигналів або друкувалися на стрічці, а відображалися відхиленням стрілки в ту чи іншу сторону.

Слово “ОК” почали використовувати в телеграфних повідомленнях, щоб позначати, що все в порядку. У азбуці Морзе слово “ОК” записується як “--- .-.” (це --- і .-.), що означає “Все правильно”.

Знаменитий код лиха “SOS” азбукою Морзе записується так: “... --- ...” (три точки, три смужки, три точки). Сигнал SOS як знак лиха (кораблів чи чогось іншого) був обраний для зручності відправлення та запам'ятовування, і сам по собі нічого не означає. Коли фізики змогли передавати радіохвилі, дроти в телеграфі змогли усунути й замість цього використовувати бездротові радіоприймачі, які приймали сигнали (електромагнітні хвилі) з азбукою Морзе. У 1907 році Гульельмо Марконі створив першу постійну трансатлантичну бездротову телеграфну лінію від Кліфденса (Ірландія) до Глейс-Бей (Канада).

З допомогою радіоустановок Марконі, які були встановлені на кораблі “Титанік”, радисти змогли передати сигнали лиха CQD, а потім і міжнародний сигнал лиха SOS, що допомогло врятувати 712 людей, хоча за умови наявності достатньої кількості рятувальних човнів на судні, був шанс, що всі 2208 людей, які були на борту Титаніка під час його зіткнення з айсбергом у 1912 році на шляху до

Нью-Йорка, за 600 кілометрів від канадського острова Ньюфаундленд, були б врятовані. На “Титаніку” було досить рятувальних жилетів, але рятувальних човнів вистачило лише на половину всіх пасажирів. Уцілілих з Титаніка підібрали корабель “Карпатія”, який отримав радіоповідомлення про лихо. “Титанік” мав захист у вигляді шлюзів (воріт) від пробою під ватерлінією довжиною до 50 метрів і ширину до кількох метрів, але під час зіткнення “Титаніка” з айсбергом (брилою льоду) нижче ватерлінії (тобто в частині корабля, яка знаходиться під водою), виник розрив довжиною 90 метрів через те, що були відрівані металеві заклепки, які з’єднували металеві пластини фюзеляжу. “Пропозиції щодо покращення засобів порятунку життя на морі з’являються після великих морських катастроф, і втрата ‘Титаніка’ далеко не є винятком” (Американський журнал “Популярна механіка”, липень 1912 р.)

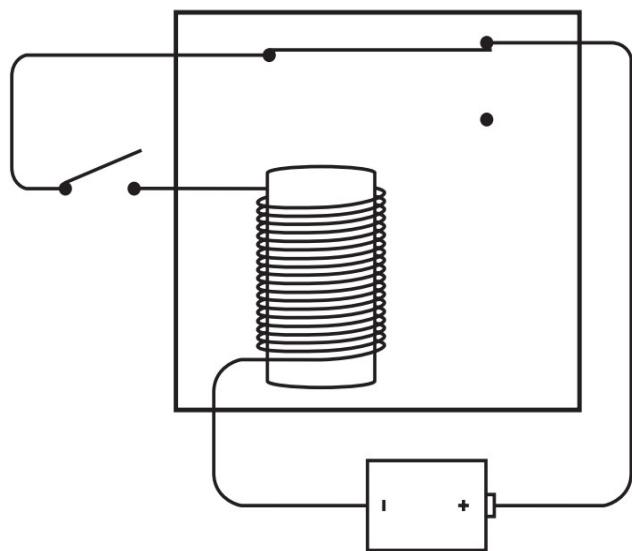
Після катастрофи Титаніка почалися розробки компактних рятувальних човнів та інших засобів, зокрема, зараз існують компактні надувні човни з балоном стиснутого газу, який може сам швидко надути човен.

Навіть наявна кількість шлюпок Титаніка заповнювалась не оптимально, одні шлюпки мали вільне місце, а інші були перевантажені, а все тому, що оптимальний розподіл людей при недостачі шлюпок потребує застосування не простого алгоритму, про який не може йти й мова в екстреному становищі. Умови задачі такі: Потрібно посадити на шлюпки максимум людей, щоб при цьому шлюпка не була перевантажена. Люди можуть сідати один одному на коліна, якщо немає місця. Звичайно, вважаємо, що людей більше ніж загальна кількість місць у шлюпках і кожна людина має свою вагу, зокрема, можуть бути люди які у два й три рази важчі ніж інші. Може бути додаткова умова: діти мають пріоритет при посадці. Потрібно знайти алгоритм який забезпечує оптимальну посадку. Наприклад, вантажопідйомність (дедвейт) шлюпки 4 умовних одиниці, вага людей [1,2,2,3]. Є всього дві шлюпки. Єдине правильне розміщення буде [2, 2], [1,3].

Історично передача голосу та даних базувалася на методах комутації каналів, як прикладом є традиційна телефонна мережа, де кожному телефонному дзвінку виділяється спеціальне, кінець до кінця, електронне з’єднання між двома станціями зв’язку. З’єднання встановлюється комутаційними системами, які з’єднують кілька проміжних ланок виклику між цими системами на час виклику.

Реле – це вимикач, який контролюється з допомогою електричного приводу. Реле використовуються там, де необхідно керувати ланцюгом з допомогою незалежного малопотужного сигналу, або коли кількома ланцюгами необхідно керувати одним сигналом. Реле вперше були використані в ланцюгах міжміського телеграфу як повторювачі сигналів: вони оновлюють сигнал, що надходить з одного ланцюга, передаючи його по іншому. Реле широко використовувалися в телефонних станціях і ранніх комп’ютерах для виконання логічних операцій. Традиційна форма реле використовує електромагніт для замикання або розмикання контактів, але були винайдені й інші принципи роботи, наприклад, у твердотілих реле, які використовують властивості напівпровідника для керування, не покладаючись на рухомі частини.

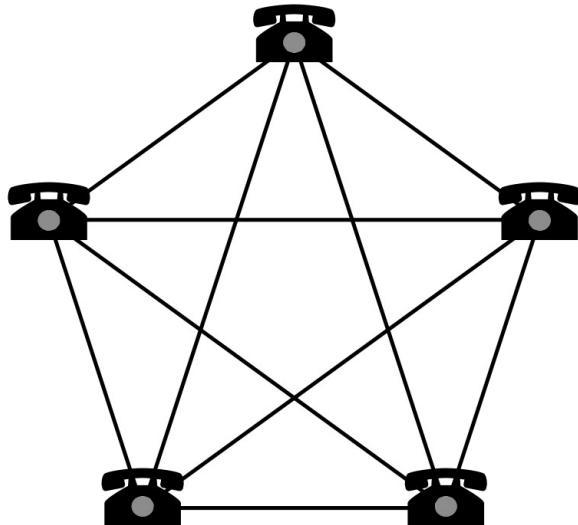
Простий спосіб перетворити електрику в рух – це електричні зумери. Подивіться на реле, підключене до вимикача та електричної батареї.



Зазвичай релейний вхід відокремлений від виходу, але тут вони утворюють кільце. Коли ви замикаєте вимикач, ланцюг стає замкненим і через нього тече струм. Струм змушує електромагніт притягувати гнучку металеву смугу. Але як тільки вона рухається, ланцюг розмикається, електромагніт втрачає свої магнітні властивості, смужка повертається у вихідне положення і, звичайно, знову замикає ланцюг. Загалом роботу цієї схеми можна описати так: поки вимикач замкнутий, металева смуга буде кидатися між двома контактами, то замикаючи, то розмикаючи ланцюг, а її рухи, ймовірно, будуть супроводжуватися шумом. Це шумне реле називається зумером. Прикріпіть молоточок до рухомої частини зумера, поставте поруч металеву чашку — ось вам електричний дзвінок.

Кількість унікальних можливих з'єднань у мережі

Два телефони можуть встановити лише одне з'єднання, п'ять - 10 з'єднань, а дванадцять - 66 з'єднань.



Кількість унікальних можливих з'єднань у мережі з n вузлами можна математично виразити як трикутне число $n(n-1)/2$.

Повнозв'язна топологія мережі — це топологія, в якій кожен вузол (або пристрій) мережі з'єднаний безпосередньо з усіма іншими вузлами. Іншими словами, кожен вузол мережі має пряме з'єднання з усіма іншими вузлами.

Повнозв'язна топологія містить $n^*(n-1)/2$ каналів зв'язку, де n — кількість вузлів. Мережі з повнозв'язною топологією відрізняються високою надійністю, оперативністю і можливістю прихованої передачі.

Якщо є N програмістів, то кількість пар програмістів $N(N-1)/2$, тобто зі зростанням числа програмістів витрати часу на взаємодію ростуть квадратично.

Протокол зв'язку

Протокол зв'язку – це система правил, яка визначає правила, синтаксис, семантику та синхронізацію зв'язку та можливі методи виправлення помилок.

Приклад протоколу

Припустимо, що двоє людей, А (Еліс) та В (Боб), вирішують зіграти в грі орел та решка, тобто вони вирішують підкинути монету і залежно від того, яка сторона підійде, виграє один з гравців, які зробили ставку на цей результат. Це імовірнісна гра. За звичайних умов, коли обидва гравці знаходяться в одній кімнаті, ця гра не є складною, достатньо лише одному з гравців кинути монету, і обидва гравці будуть стежити за результатом кидка і визначати, орлом чи решкою впала монета. Але припустимо, наші гравці знаходяться в різних містах і можуть розмовляти тільки телефоном (у них немає відеозв'язку). Проблема виникає, якщо один гравець кидає монету, інший не зможе контролювати результат цього кидка. Тобто про результат можна збрехати. Щоб забезпечити чесність і правдивість результатів цієї гри, ми можемо створити певний протокол дій, який забезпечить правильність результатів. Для цього ми замінимо підкидання монети іншою процедурою, але яка дасть подібні ймовірні результати.

Розглянемо односторонню функцію $f(x)$, яка задовольняє наступним умовам:

- 1) $f(x)$ визначено на деякому наборі цілих чисел, що містить однакову кількість парних і непарних чисел;
- 2) функція $f(x)$ така, що коли $f(x) = f(y)$, тоді x і y мають однакову парність.
- 3) функція $f(x)$ така, що при значенні $f(x)$ важко обчислити парність невідомого аргументу x .
Припустимо, що функція, яка задовольняє заданим властивостям, обрана і відома учасникам розіграшу.

Нехай А ніби підкидає монету, а В намагається вгадати результат. Тоді протокол обміну між абонентами, який вирішує проблему, складається з наступних кроків:

- a) А вибирає випадкове значення x (образно підкидає монету), шифрує x і надсилає отримане значення $f(x)$ до В.
- б) В, отримавши $f(x)$, намагається вгадати парність x і спрямовує свою згадку до А.
- в) А, отримавши відгадку від В, повідомляє В, чи вгадав він це, чи ні, і надсилає йому вибране x на підтвердження.
- г) В перевіряє, чи А обдурив його, для чого обчислює $f(x)$ і порівнює його з отриманим від А на першому кроці.

Якщо $f(x)$ збігається з тим, який В отримав від А на першому кроці а, то результат жеребкування, оголошений А на третьому кроці, передається для прийняття. Дійсно, навіть якщо А намагається обдурити й посилає В перевірити інше значення x' , для якого $f(x') = f(x)$, то x і x' матимуть однакову парність завдяки властивості 2 функції $f(x)$, і цей трюк не вдається. Це міркування ілюструє основну функцію протоколів обміну інформацією, а саме, протокол визначає, яка інформація буде вважатися правильною, а яка ні. Але цей протокол вразливий до Man-in-the-middle attack.

RSA (Rivest–Shamir–Adleman) — це криптосистема з відкритим ключем, яка широко використовується для безпечної передачі даних.

Акронім RSA походить від прізвищ Рона Рівеста, Аді Шаміра та Леонарда Адлемана, які публічно описали алгоритм у 1977 році.

RSA — це криптосистема з відкритим ключем. Він заснований на складності задачі розкладання числа на множники.

Як працює RSA.

p, q — великі прості числа.

$$n = p * q;$$

$$\phi(n) = (p-1)(q-1)$$
 — функція Ейлера числа n ;

$$e, d$$
 — натуральні числа такі, що e, d взаємно прості з $\phi(n)$ і $e * d = 1 \pmod{\phi(n)}$.

Перший абонент А генерує свій відкритий ключ $K_{pub} = \{n, e\}$ і секретний ключ $K_{priv} = \{n, d\}$.

Відкритий ключ А повідомляється всім абонентам. Будь-який інший абонент В може зашифрувати секретне повідомлення m для А, використовуючи відкритий ключ А. Отримавши зашифрований текст c , абонент А розшифрує його з допомогою свого секретного ключа.

$$\text{Шифрування: } c = E(m) = m^e \pmod{n}$$

$$\text{Дешифрування: } m = D(c) = c^d \pmod{n}$$

Доказ коректності алгоритму шифрування RSA

Теорема. Функції $E(m)$ і $D(c)$ визначають взаємно обернені перестановки множини чисел Z .

Доказ. $E(D(M)) = D(E(M)) = M^{ed} \pmod{n}$ для будь-якого $M \in Z$. Ми знаємо, що $e \neq d$ є взаємно оберненими за модулем $\phi(n)$, тобто

$$ed = 1 + k(p-1)(q-1)$$

для деякого цілого числа k . Якщо $M \neq 0 \pmod{p}$, то за Малою теоремою Ферма маємо

$$M^{ed} = M(M^{p-1})^{k(q-1)} = M * 1^{k(q-1)} = M \pmod{p}$$

З тих же причин $M^{ed} = M \pmod{q}$, і тому $M^{ed} = M \pmod{n}$ для будь-якого M .

TCP/IP

Історія розвитку зв'язку пройшла довгий шлях від телеграфу до Інтернету. Телеграф був першим засобом віддаленого зв'язку, який дозволяв передавати повідомлення шляхом кодування (азбука Морзе) та передачі сигналів через дроти. Потім з'явилися телефонні мережі, що дозволяли голосове спілкування на відстані.

Все це стало можливо завдяки дослідженням електромагнітної індукції, які проводив Майкл Фарадей, Джеймс Максвелл, Генріх Герц та інші.

У 1907 році італійський винахідник Гульельмо Марконі створив перший постійний трансатлантичний бездротовий телеграф. Марконі нобелівський лауреат у галузі фізики (1909) за роботи з бездротової телеграфії.

Американський винахідник Філо Фарнсуорт відомий завдяки винаходу електронної передавальної трубки — "диссектора", створенню на її основі електронної системи телебачення та тим що вперше в Америці 1 вересня 1928 року публічно продемонстрував передачу рухомого зображення. Згодом диссектор не витримав конкуренції з "іконоскопом" Володимира Зворікіна.

Радіомережі, такі як радіо та телебачення, забезпечували безпровідний зв'язок та масове мовлення. Створення багатокористувальників комп'ютерів з багатьма терміналами відкрило шлях до спільног доступу до обчислювальних ресурсів.

ARPANET, розроблений в 1960-х роках, вважається прародичем Інтернету, адже ця мережа мала багато вузлів, які могли обмінюватися даними.

ARPANET був розроблений агентством DARPA (Defense Advanced Research Projects Agency) Сполучених Штатів Америки. Це агентство мало за мету створення технологій, що могли б захищати країну, а також проводило дослідження в галузі передових технологій.

Протокол Ethernet був розроблений компанією Хекох (Зірокс) в середині 1970-х років. Він став стандартом для локальних мереж і визначав спосіб, яким комп'ютери та інші пристрої можуть обмінюватися даними через провідні кабелі.

Протокол Ethernet став стандартом для провідних локальних комп'ютерних мереж, а HTTP (Hypertext Transfer Protocol) став протоколом для обміну даними в World Wide Web. Всі ці кроки допомогли сформувати сучасну інфраструктуру Інтернету, яка відкрила безмежні можливості для спілкування, обміну інформацією та спільної роботи.

Протокол HTTP був розроблений командою вчених у Європейському центрі ядерних досліджень (CERN) в 1989 році. Головний розробник, Тім Бернерс-Лі, вигадав цей протокол для обміну гіпертекстовою інформацією через мережу.

Мережевий протокол описує:

- Формат повідомлення, якому застосунки зобов'язані слідувати;
- Спосіб обміну повідомленнями між комп'ютерами в контексті визначеного дії, як, наприклад, пересилка повідомлення мережею.

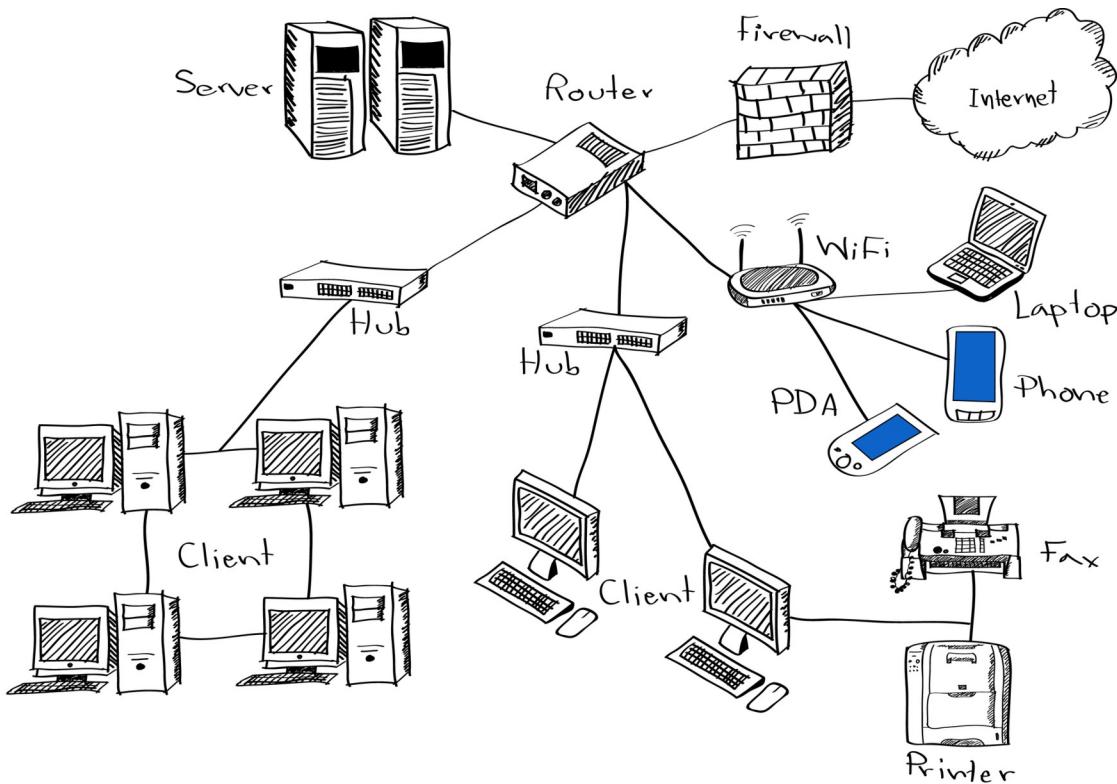
Різні протоколи найчастіше описують лише різні сторони одного типу зв'язку й, узяті разом, утворюють стек протоколів. Назви "протокол" і "стек протоколів" також вказують на програмне забезпечення, яке реалізує протоколи.

Нові протоколи для Інтернету визначаються IETF, інші протоколи — IEEE або ISO.

Специфікація протоколу може бути описана різними мовами, але найбільш поширеними є англійська та формалізована спеціальна мова, така як ASN.1 (Abstract Syntax Notation One) або XML (Extensible Markup Language). Багато стандартних протоколів, таких як HTTP, TCP/IP, SMTP, DNS і т.д., мають специфікації, які написані англійською мовою, з використанням нотації Бекуса — Наура. Такі документи зазвичай створюються організаціями, що розробляють стандарти, такими як Інтернет-інженерний та мережевий консорціум (IETF) або Міжнародна організація зі стандартизації (ISO). ASN.1 не є стандартним способом опису протоколу HTTP. Зазвичай протокол HTTP описують за допомогою текстових документів, таких як специфікація RFC (Request for Comments), що містять

опис структури повідомлень, методів запиту та відповідей, заголовків і параметрів (RFC 2616 for HTTP/1.1).

Схема мережі



LAN Network Diagram

Router (Маршрутизатор): Це пристрій, що визначає шляхи для передачі даних між різними мережами. Він приймає пакети даних і вирішує, куди і як їх переслати до їхніх пунктів призначення.

Switch (Комуутатор, Hub): Це пристрій, який використовується для з'єднання комп'ютерів в локальній мережі (LAN). Він дозволяє передавати дані між різними пристроями в мережі, враховуючи їх MAC-адреси.

Ethernet (інтернет) — найпопулярніший протокол кабельних комп'ютерних мереж, що працює на фізичному та канальному рівні мережової моделі OSI.

Ethernet використовується для з'єднання пристрій в локальній мережі (LAN). Він визначає метод передачі даних через фізичні мережеві кабелі.

Адреси Ethernet відомі як MAC-адреси (Media Access Control). Кожен мережевий адаптер (або мережевий інтерфейс) має унікальний MAC-адрес, який ідентифікує його в мережі.

Ethernet може підтримувати різні швидкості передачі даних, такі як 10 Mbps, 100 Mbps, 1 Gbps, 10 Gbps та більше, в залежності від версії та типу мережі.

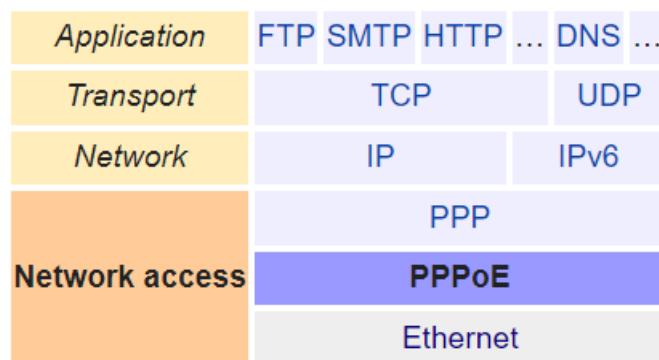
Кожен пристрій, який має мережевий інтерфейс, такий як мережева карта в комп'ютері або смартфоні, має свою унікальну MAC-адресу.

Вита пара (twisted pair) — вид мережевого кабелю, з однією або декількома парами ізольованих провідників, скручених між собою (з невеликою кількістю витків на одиницю довжини) для

зменшення взаємних наведень при передачі сигналу і покритих пластиковою оболонкою. Використовується для побудови мереж у багатьох технологіях, наприклад, Ethernet, ARCNet і Token ring.

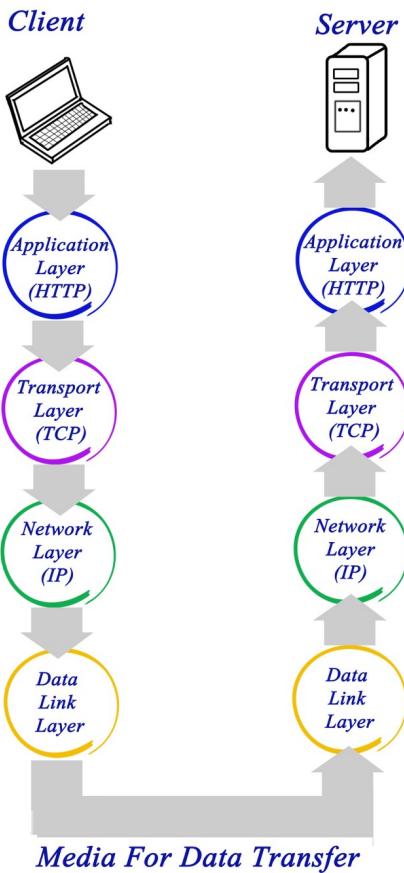
Вита пара підтримує передачу даних на відстань приблизно до 100 метрів. На більших відстанях сигнал, через загасання, не розпізнається; якщо передача даних на більшу відстань все ж необхідна, потрібно скористатися повторювачем або залучити коаксіальний кабель. Сучасний коаксіальний кабель (співвісний кабель) складається з центрального провідника, оточеного шаром діелектрика, зовнішня поверхня якого покрита обплетенням або фольгою (другим провідником) і захищеною оболонкою з пластику, що захищає кабель від дії навколошнього середовища. Також використовують оптоволоконний кабель, який передає світло.

PPPoE и TCP/IP Стек протоколів



TCP/IP є скороченням від "Transmission Control Protocol/Internet Protocol" (Протокол керування передачею/Інтернет-протокол). Це набір стандартів, який визначає спосіб, яким комп'ютери обмінюються даними в мережі Інтернет та інших комп'ютерних мережах. TCP/IP використовується для розподілення даних на пакети, адресації, маршрутизації та забезпечення їх доставки відправникам. Він містить кілька протоколів, таких як TCP, UDP, IP, ICMP, ARP та інші, які використовуються для різних цілей, таких як надійна доставка даних, маршрутизація, ідентифікація пристройів тощо. TCP/IP є основою для функціонування Інтернету та багатьох локальних мереж.

Інтернет — глобальна мережа комп'ютерів та DNS-серверів.



Розподіл протоколів за рівнями моделі TCP/IP:

- Прикладний рівень (Application Layer): HTTP, RTSP, FTP, DNS.
- Транспортний (Transport Layer): TCP, UDP, SCTP, DCCP.
- Мережевий (Network Layer) Для TCP/IP це IP
(допоміжні протоколи, на кшталт ICMP і IGMP, працюють поверх IP, але теж належать до мережевого рівня; протокол ARP є самостійним допоміжним протоколом, що працює поверх канального рівня)
- Канальний (Link Layer): Ethernet, IEEE 802.11, WLAN, SLIP, Token Ring, ATM та MPLS, фізичне середовище та принципи кодування інформації, T1, E1.

Стек протоколів TCP/IP включає чотири основні шари: фізичний, канальний, мережевий та транспортний. Цей стек використовується для передачі даних в мережі. Наприклад, TCP (Transmission Control Protocol) працює на транспортному рівні, забезпечуючи надійну доставку даних, тоді як IP (Internet Protocol) працює на мережевому рівні й відповідає за адресацію та маршрутизацію пакетів. На фізичному рівні передача даних відбувається через фізичні носії, такі як кабелі або бездротові з'єднання. Канальний рівень відповідає за передачу фреймів між пристроями в мережі, декодування та виявлення помилок.

Мережевий рівень (IP) визначає логічні адреси (IP-адреси) для пристрій в мережі та визначає оптимальний шлях для передачі даних між ними.

Транспортний рівень відповідає за керування з'єднаннями та надійною доставкою даних. TCP забезпечує гарантовану доставку та контроль потоку, тоді як UDP (User Datagram Protocol) використовується для швидкої, але менш надійної передачі.

Фізичний рівень опрацьовує електричні, оптичні або радіочастотні сигнали для передачі бітів по мережевому середовищу. Канальний рівень вирішує, як точно передати біти між пристроями через спільне середовище передачі даних, вирішуючи конфлікти та виявляючи помилки.

MAC-адреса (Media Access Control address) — це унікальний ідентифікатор, який присвоюється мережевому інтерфейсу для його ідентифікації у мережі. Кожен пристрій, який має мережевий інтерфейс, такий як мережева карта в комп'ютері або смартфоні, має свою власну унікальну MAC-адресу.

DHCP (Dynamic Host Configuration Protocol — протокол динамічної конфігурації вузла) — це стандартний протокол прикладного рівня, який дозволяє комп'ютерам автоматично отримувати IP-адресу та інші параметри, необхідні для роботи в мережі. Для цього комп'ютер звертається відповідно — до DHCP-сервера. Протокол DHCP використовується в більшості великих мереж TCP/IP.

У протоколів TCP/IP "Handshake" (рукостискання) використовується для встановлення з'єднання між двома пристроями, які спілкуються мережею. У випадку TCP (Transmission Control Protocol), що забезпечує надійну передачу даних, Handshake включає три послідовні етапи:

SYN (Синхронізація): Клієнт (наприклад, ваш комп'ютер) посилає пакет з пропозицією SYN (синхронізації) до сервера (наприклад, вебсервер), щоб започаткувати з'єднання.

SYN-ACK (Синхронізація-Підтвердження): Сервер відповідає пакетом, що містить пропозиції SYN та ACK (підтвердження), показуючи готовність до спілкування.

ACK (Підтвердження): Клієнт підтверджує отримання пакету SYN-ACK шляхом відправлення пакету з пропозицією ACK до сервера. Після цього з'єднання вважається встановленим, і обидва пристрої можуть почати обмін даними.

Цей триетапний процес Handshake забезпечує впевненість у встановленні з'єднання та надійності передачі даних, оскільки кожен етап підтверджується іншим пристроєм.

Доменне ім'я (Domain name), або Домен — частина простору ієрархічних імен мережі Інтернет, що обслуговується групою серверів системи доменних імен (DNS-серверів) та централізовано адмініструється.

Кожен комп'ютер в Інтернеті має свою власну унікальну адресу — число, яке складається з чотирьох (у протоколі IPv4) або шістнадцяти (у протоколі IPv6) байтів. Оскільки запам'ятати десятки чи навіть сотні номерів — важка процедура, то всі (чи майже всі) машини мають імена, запам'ятати які (особливо якщо знати правила утворення імен) значно легше.

DNS-клієнт звертається до кореневого DNS-сервера з вказівкою повного доменного імені; DNS-сервер відповідає клієнту, вказуючи адресу наступного DNS-сервера, який виконує обслуговування домену верхнього рівня, заданого в наступній старшій частині імені; DNS-клієнт виконує запит наступного DNS-сервера, який його надсилає до DNS-сервера потрібного піддомену і т. д., доти, доки не буде знайдено DNS-сервер, який повністю відповідає запитуваному імені IP-адреси. Сервер дає кінцеву відповідь клієнту.

Проксі-сервер (proxy — "представник, уповноважений") — це посередник між вашим комп'ютером і Інтернетом. Він працює, отримуючи запити від вашого комп'ютера і пересилаючи їх на зовнішні сервери, а потім отримуючи відповіді і пересилаючи їх назад вам. Це може бути корисно з кількох причин, таких як забезпечення анонімності, фільтрація вмісту або забезпечення безпеки, наприклад, шляхом блокування небажаних сайтів або застосунків.

Проксі може бути реалізований як програма, що запускається на комп'ютері або сервері, або як фізичний сервер. У випадку програмної реалізації, проксі-сервер може запускатися на звичайному комп'ютері або сервері і функціонувати як програма, яка обробляє мережевий трафік і виконує

відповідні операції згідно з налаштуваннями. Фізичний проксі-сервер може бути самостійним пристроєм, який спеціалізується на обробці мережевого трафіку і виконує різні функції проксі.

Міжмережевий екран, Firewall, або брандмауер, — це пристрій або програмне забезпечення, яке контролює рух мережевого трафіку між мережами з різним рівнем довіри або безпеки. Він використовується для захисту мережі від несанкціонованого доступу, контролю над вхідними та вихідними з'єднаннями, фільтрації трафіку на основі правил безпеки та іншого функціоналу, спрямованого на забезпечення безпеки мережі.

Коли користувач робить запит на домен, його браузер (вебнавігатор) використовує протокол HTTP або його захищено версію HTTPS для відправлення запиту на сервер. Запит містить метод (наприклад, GET, POST), URL домену, заголовки запиту та інші параметри.

Прийнятий сервером запит обробляється вебсервером, таким як Apache, Nginx або Microsoft IIS. Він може виконати різні операції, включаючи пошук файлів, генерацію вмісту, виконання скриптів та звертання до баз даних. Якщо запит успішно оброблено, сервер формує відповідь.

Відповідь сервера також використовує протокол HTTP або HTTPS. Вона містить статус-код, який вказує на успішність запиту або помилку, заголовки відповіді і, головне, тіло відповіді. Тіло відповіді може містити HTML-код, CSS-стилі, JavaScript-скрипти та інші ресурси, необхідні для показу сторінки в браузері.

Для виконання безпечних передач даних між клієнтом і сервером, може бути застосований протокол TLS (Transport Layer Security), який забезпечує шифрування даних та автентифікацію сторінок. Він активується при використанні HTTPS.

Крім основних протоколів HTTP і HTTPS, в процесі отримання HTML сторінки також можуть бути залучені інші протоколи та технології:

DNS (Domain Name System): Коли браузер отримує URL, він використовує DNS для перетворення доменного імені на IP-адресу сервера, де знаходиться вебсайт.

TCP/IP (Transmission Control Protocol/Internet Protocol): Ці протоколи забезпечують передачу даних через Інтернет. HTTP та HTTPS побудовані на основі TCP/IP.

HTML (Hypertext Markup Language): Це мова розмітки, яка визначає структуру сторінки. Відповідь сервера зазвичай містить HTML-код, який браузер рендерить для відображення сторінки.

Cookies і Sessions: Для зберігання та обміну даними між клієнтом і сервером можуть використовуватися куки (cookies) і сесії (sessions), які забезпечують зручність інтеракції користувача з вебсайтом.

Ось короткий приклад опису HTTP-запиту і відповіді:

HTTP-запит:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Cookie: session_id=abc123; user_id=123456
```

HTTP-відповідь:

HTTP/1.1 200 OK

Date: Sat, 15 Feb 2024 12:00:00 GMT

Server: Apache/2.4.38 (Ubuntu)

Content-Type: text/html

Content-Length: 1512

Set-Cookie: session_id=abc123;

```
<!DOCTYPE html>
<html>
<head>
    <title>Example Page</title>
</head>
<body>
    <h1>Hello, world!</h1>
</body>
</html>
```

Куки включаються в HTTP-запит у заголовку з назвою "Cookie". У цьому заголовку передається інформація про куки, які браузер має відправити на сервер. Кожен кукі представлений у форматі "назва=значення". Якщо на сервер потрібно передати декілька куки, вони розділяються крапкою з комою (;).

HTTP-куки (або просто куки) — це невеликі файли, які вебсайти зберігають на комп'ютерах користувачів. Вони використовуються для збереження інформації про користувача під час його перебування на веб-сайті або при повторному візиті на нього.

Куки можуть зберігати інформацію про те, чи ви увійшли на сайт або ні. Наприклад, вони можуть запам'ятовувати ваші дані для входу, щоб вам не доводилося вводити їх кожен раз.

Куки дозволяють вебсайтам відстежувати сесії користувачів. Це може бути корисним для аналізу того, як користувачі взаємодіють із сайтом і як їхній шлях може бути поліпшений.

Куки складаються з пар ключ-значення, де кожна пара визначає окремий атрибут.

При першому відвідуванні вебсайту сервер може відправити куки у відповіді на запит браузера.

Браузер зберігає ці куки та додає їх до кожного наступного запиту до цього домену.

MIME (Multipurpose Internet Mail Extensions – багатоцільові розширення інтернет-пошти) – стандарт, що описує передачу різних типів даних електронною поштою, а також, у загальному випадку, специфікація для кодування інформації та форматування повідомлень таким чином, щоб їх можна було пересилати через Інтернет.

Основний формат електронних повідомлень визначено у RFC 5322.

На даний час майже вся електронна пошта передається через протокол SMTP у форматі MIME.

Хоча MIME був розроблений в основному для SMTP, проте в інших стандартах широко використовуються типи вмісту, які визначені цим стандартом. Так, стандартом HTTP вимагається вказувати в заголовках MIME тип вмісту відповіді, що надсилається сервером клієнтові. Клієнт використовує отриманий MIME тип для обрання відповідної програми-переглядача, яка й покаже на екрані отримані дані. Деякі з таких переглядачів вже вбудовані в браузери, насамперед це відображення html-даних, зображень в форматах GIF, PNG, JPG, відеокліпів у форматах WebM, mp4 тощо.

На швидкість завантаження вебсторінки та її вмісту помітно впливає те, наскільки далеко користувач перебуває від сервера. Це відбувається тому, що при використанні технології TCP/IP, яка використовується для передачі інформації у мережі Інтернет, затримки при передачі інформації залежать від кількості маршрутизаторів (routers), розташованих між джерелом та споживачем вмісту. Розміщення контенту одночасно на кількох серверах засобами CDN може скоротити маршрут передачі даних мережею і прискорити завантаження сайту з точки зору користувача.

Використання CDN (Content Delivery Network) зменшує кількість хопів між маршрутизаторами, що суттєво збільшує швидкість завантаження контенту з Інтернету.

CDN

У дротових комп'ютерних мережах, включаючи Інтернет, стрібок (hop) відбувається, коли пакет передається від одного сегмента мережі до іншого. Пакети даних проходять через маршрутизатори (routers) між джерелом і одержувачем. Підрахунок стрібків означає кількість мережевих пристрій, через які дані проходять від джерела до пункту призначення.

Оскільки під час кожного стрібка виникають затримки зберігання, пересилання та інші, велика кількість стрібків між джерелом і одержувачем означає нижчу продуктивність у реальному часі.

CDN означає мережу доправлення контенту (Content Delivery Network). Це глобальна система серверів, розташованих у різних регіонах світу, які спрямовані на забезпечення швидкого та ефективного доставлення вебконтенту до кінцевих користувачів. CDN дозволяє кешувати контент на серверах, розташованих близче до кінцевих користувачів, що зменшує час завантаження сторінок та покращує продуктивність вебсайтів.

Anycast — це методологія мережевої адресації та маршрутизації, за якої одна IP-адреса спільно використовується пристроями (зазвичай серверами) у кількох місцях. Маршрутизатори направляють пакети, адресовані цьому пункту призначення, до найближчого до відправника місця, використовуючи свої звичайні алгоритми прийняття рішень, як правило, найменшу кількість мережевих стрібків. Маршрутизація Anycast широко використовується мережах доправлення (і розповсюдження) контенту (CDN), такими як вебсервери та сервери імен, щоб наблизити свій вміст до кінцевих користувачів.

Bluetooth

Дизайн бездротової специфікації Блютуз був названий на честь короля Гаральда в 1997 році, заснований на аналогії, що технологія об'єднає пристрой так, як Гаральд Блютуз об'єднав племена Данії в єдине королівство. Логотип Блютуз складається з ініціалів Гаральда у вигляді скандинавських рун Н (ᚦ) і В (ᛒ).

Блютуз — це стандарт бездротової технології, який використовується для обміну даними між фіксованими та мобільними пристроями на короткі відстані з використанням радіохвиль ультрависокої частоти в промислових, наукових і медичних діапазонах від 2,402 ГГц до 2,480 ГГц, а також для створення персональних мереж. Назва “Блютуз” була запропонована в 1997 році Джимом Кардачем з Інтел, який розробив систему, яка дозволить мобільним телефонам зв'язуватися з комп'ютерами. Під час цієї пропозиції він читав історичний роман Франса Г. Бенгтссона “Рудий Орм”.

Блютуз (Bluetooth) — це бездротова технологія, яка дозволяє пристроям обмінюватися даними через радіочастотний сигнал у короткому діапазоні. Вона використовується для з'єднання пристройв, таких як смартфони, навушники, колонки, комп'ютери та інше обладнання.

Безпека Bluetooth базується на криптографії, яка зашифрує дані, що передаються між пристроями. Низька потужність Bluetooth та сучасних радіохвиль, які використовуються для бездротового зв'язку, не мають достатньої енергії для того, щоб викликати нагрівання чи шкоду клітинам людини, навіть при тривалому використанні.

JSON

JSON (JavaScript Object Notation) — це легкий формат обміну даними, який використовується для передачі структурованих даних між програмами.

JSON використовується для передачі даних між веб-серверами та клієнтами, особливо в AJAX-запитах, які використовуються для асинхронного оновлення веб-сторінок без перезавантаження.

JSON побудований на двох структурах:

Колекція пар ім'я/значення. У різних мовах це реалізовано як об'єкт, структура, словник, хеш-таблиця, список із ключами або асоціативний масив.

Упорядкований список значень. У більшості мов це реалізовано як масив, вектор, або список.

Ось приклад формату JSON:

```
{  
  "name": "John Doe",  
  "age": 30,  
  "city": "New York",  
  "isStudent": false,  
  "grades": [85, 90, 92],  
  "address": {  
    "street": "123 Main St",  
    "zip": "10001"  
  }  
}
```

Основні правила JSON такі:

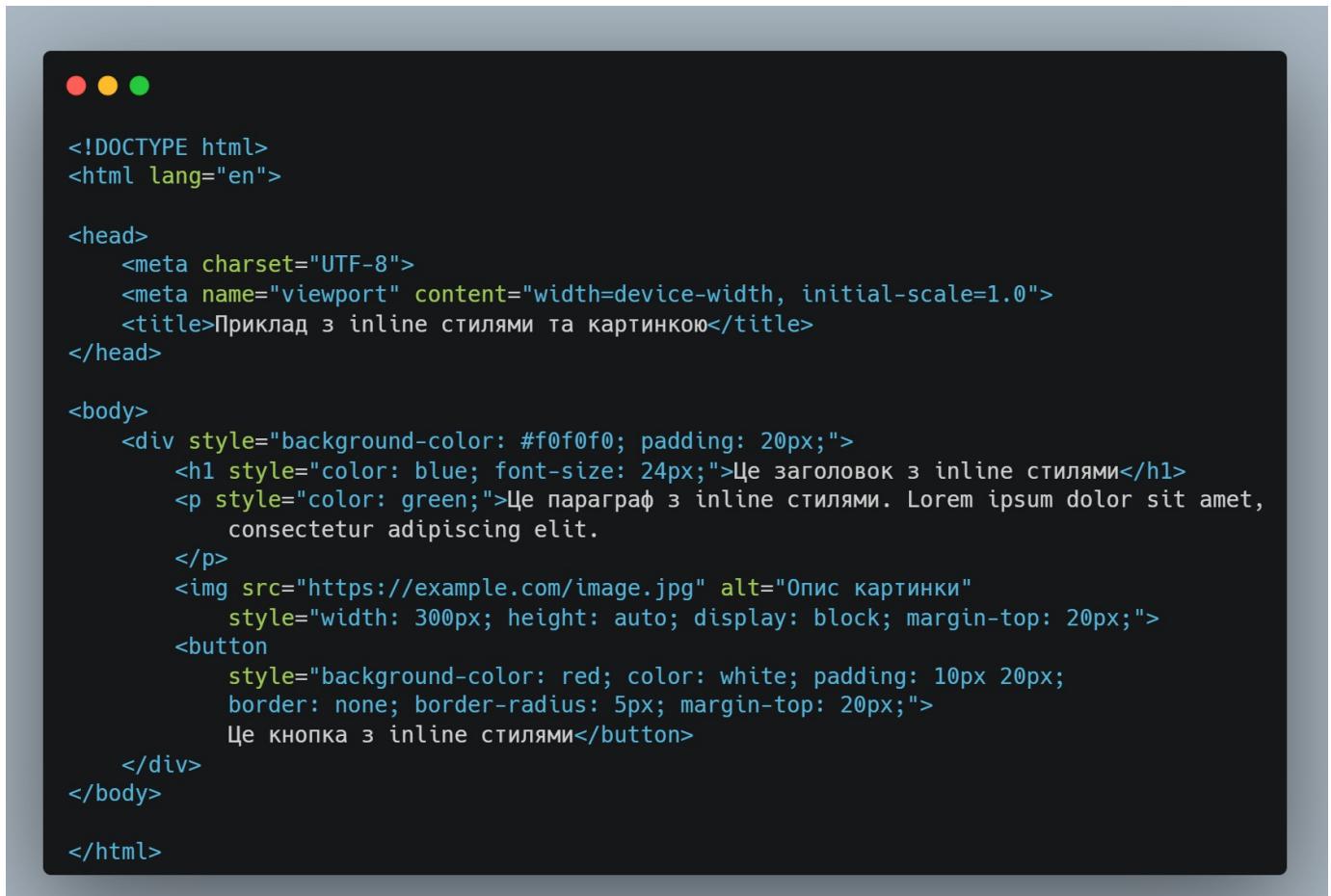
1. Синтаксична структура: JSON складається з пар "ключ: значення", розділених комами. Дані групуються в об'єкти `{}` або в масиви `[]`.
2. Типи даних: JSON підтримує рядок (string), число (number), об'єкт (object), масив (array), булеве значення (true/false) і значення null.
3. Рядкові дані: Рядкові дані мають бути в подвійних лапках, наприклад: `"**ключ**": "значення"`.
4. Числові дані: Числові дані можуть бути цілими або десятковими. Наприклад, `"**ключ**": 10` або `"**ключ**": 3.14`.
5. Масиви: Масиви визначаються за допомогою квадратних дужок `[]` і можуть містити різні типи даних, включаючи інші масиви.
6. Об'єкти: Об'єкти визначаються за допомогою фігурних дужок `{}` і містять пари "ключ: значення".
7. Коментарі: JSON не підтримує коментарі.
8. Вкладеність: JSON дозволяє вкладати об'єкти один в одного без обмежень глибини.
9. Розділювачі: Пари "ключ: значення" відокремлюються комами, а об'єкти розділяються комами.
10. Розділові знаки: У JSON немає розділових знаків на кінці об'єктів або масивів.

Ці правила визначають структуру та формат даних у JSON.

HTML

HTML, або Мова гіпертекстової розмітки (HyperText Markup Language), це стандартна мова розмітки для створення вебсторінок і вебдокументів. HTML використовується для створення структури та відображення змісту вебсторінок, таких як текст, зображення, відео, посилання та іншого контенту. Він визначає розмітку елементів на вебсторінці за допомогою тегів (h1, a, p, div, img, button, table, form, ul, span, input), які вказують браузеру, як зображати кожен елемент. HTML є однією з основних мов веброзробки, яка використовується разом з CSS (Каскадні таблиці стилів) і JavaScript для створення динамічних інтерактивних вебсайтів.

Мова гіпертекстової розмітки (HTML) була розроблена Тімом Бернерсом-Лі у 1989 році, коли він працював у ЦЕРН (Європейська організація з ядерних досліджень) в Швейцарії. Перша специфікація HTML була випущена в 1993 році. Тім Бернерс-Лі разом з іншими вченими створив HTML як частину системи для обміну та обробки документів на Інтернеті. HTML основана на метамові SGML (Standard Generalized Markup Language, 1986).



The screenshot shows a dark-themed code editor window with three circular window control buttons at the top left. The main area contains the following HTML code with inline styles:

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Приклад з inline стилями та картинкою</title>
</head>

<body>
    <div style="background-color: #f0f0f0; padding: 20px;">
        <h1 style="color: blue; font-size: 24px;">Це заголовок з inline стилями</h1>
        <p style="color: green;">Це параграф з inline стилями. Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
        
        <button
              style="background-color: red; color: white; padding: 10px 20px;
              border: none; border-radius: 5px; margin-top: 20px;">
            Це кнопка з inline стилями</button>
    </div>
</body>

</html>
```

```

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Приклад з inline стилями та картинкою</title>
</head>

<body>
    <div style="background-color: #f0f0f0; padding: 20px;">
        <h1 style="color: blue; font-size: 24px;">Це заголовок з inline стилями</h1>
        <p style="color: green;">Це параграф з inline стилями. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        </p>
        
        <button
            style="background-color: red; color: white; padding: 10px 20px;
            border: none; border-radius: 5px; margin-top: 20px;">
            Це кнопка з inline стилями</button>
    </div>
</body>

</html>

```

Основні принципи HTML включають:

1. Структура: HTML дозволяє визначити структуру вебсторінки за допомогою елементів, таких як `<html>` , `<head>` , та `<body>` . Ця структура допомагає браузеру правильно інтерпретувати контент сторінки.
2. Розмітка: HTML використовує теги для розмітки різних елементів сторінки, таких як заголовки `<h1>` до `<h6>` , абзаци `<p>` , списки `` , `` , `` , посилання `<a>` , зображення `` тощо. Це допомагає відокремлювати та організовувати різні частини контенту.
3. Атрибути: Багато тегів HTML можуть мати атрибути, які надають додаткову інформацію про елемент. Наприклад, атрибут `href` використовується для визначення посилання в тезі `<a>` , атрибут `src` - для вказівки шляху до зображення в тезі `` тощо.
4. Вкладеність: HTML дозволяє вкладати один елемент всередину іншого, утворюючи деревоподібну структуру. Це дозволяє створювати складніше форматування та структуру сторінки.
5. Семантика: HTML надає спеціальні елементи для визначення семантичної структури документа. Наприклад, `<header>` , `<footer>` , `<nav>` , `<section>` , `<article>` , `<aside>` допомагають браузерам та іншим інструментам краще розуміти структуру сторінки й покращувати доступність та індексацію контенту.
6. Кросбраузерність: HTML повинен бути написаний так, щоб він правильно відображався на різних веб-переглядачах і на різних пристроях, таких як комп'ютери, планшети та смартфони.

Стільникова мережа

Стільникова мережа — це бездротова телефонна мережа, що базується на використанні радіо- або мікрохвильових сигналів для передачі голосу та даних між мобільними пристроями.

Одна з математичних особливостей стільникових мереж — це їх структура, яка базується на геометричних принципах розміщення "клітин" чи зон покриття. Математичні моделі використовуються для визначення оптимального розміщення базових станцій у мережі, щоб забезпечити найкраще покриття при мінімальній кількості станцій. Комірка у формі правильного шестикутника має найбільшу площину серед фігур, які можуть без прогалин покрити площину.

Термін "стільникова мережа" походить від вигляду секторів покриття мережі, які нагадують клітини на стільниці. Кожна "клітина" цієї мережі функціонує незалежно, схоже на комірку або "стільник", тому виник термін "стільникова". Ця організація дозволяє мережі ефективно використовувати радіочастоти та забезпечити покриття для користувачів у різних зонах, навіть якщо вони рухаються.

Ось кілька особливостей:

Клітинна структура: Мережа розділена на сектори або "клітини", кожна з яких покриває певну територію. Це дозволяє забезпечити покриття там, де користувачі перебувають, навіть коли вони рухаються.

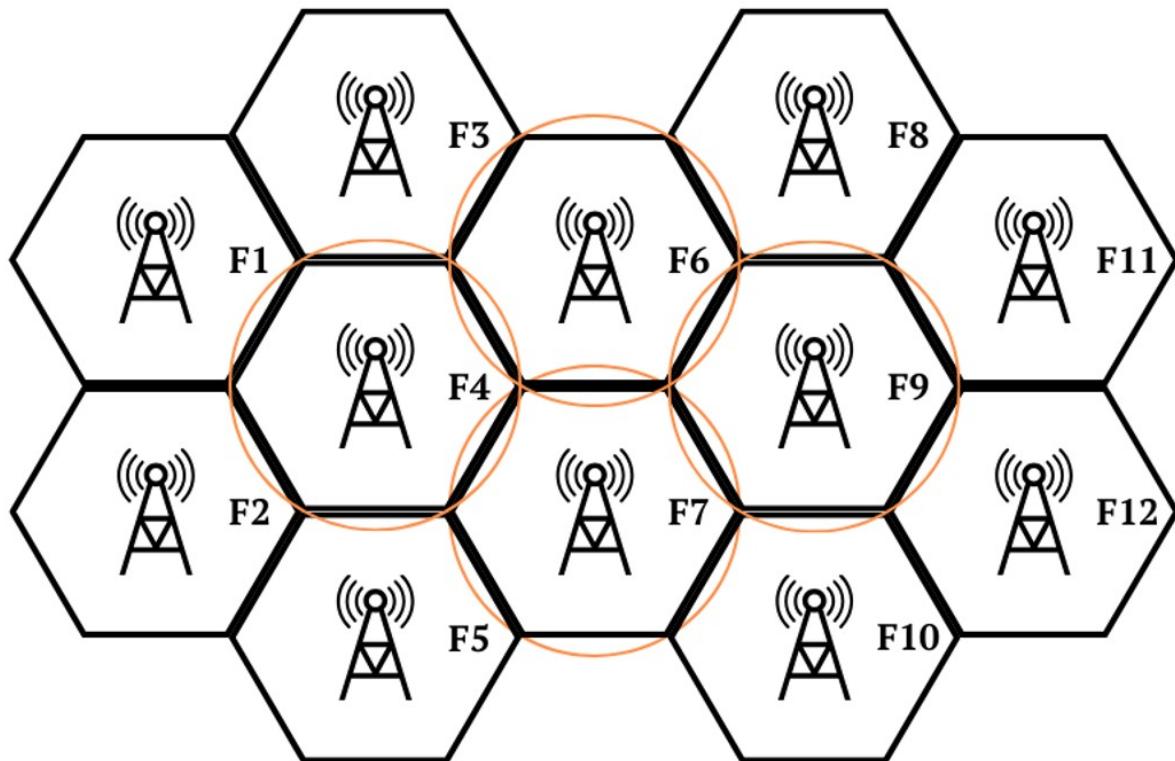
Мобільність: Одна з ключових особливостей - здатність користувачів пересуватися в межах мережі, не втрачаючи зв'язку. Коли вони переходят з однієї "клітини" в іншу, їх з'єднання автоматично під'єднується до найближчої вежі.

Частотний діапазон: Для передачі сигналів використовуються різні частоти, що дозволяє більшій кількості користувачів використовувати мережу одночасно.

Стандарти зв'язку: Існує кілька стандартів стільникових мереж (наприклад, GSM, CDMA, LTE), які визначають технічні параметри та протоколи для забезпечення зв'язку.

Зони покриття: Мережа розбита на зони покриття, кожна з яких має базову станцію, що приймає та передає сигнали.

Стільникова мережа



В 4 столітті нашої ери жив Папп Александрійський, який насамперед відомий як автор "Математичної збірки". Папп у своїй збірці описує вивчення геометрії бджолиних стільників (системи чарунок). Бджола конструює з воску стільники, в яких зберігається мед і личинки бджіл. В принципі, кожна комірка стільника повинна мати форму, що має найбільшу площину і при цьому може складатись разом з такими ж комірками, так щоб не було прогалин. Відомо, з задачі Дідона, що коло є плоскою фігурую з найбільшою площею, але ви не можете замостили колами площину так, щоб між ними не було прогалин. Папп пише у своєму творі, що комірка у формі правильного шестикутника має найбільшу площину серед фігур, які можуть без прогалин покрити площину. Саме шестикутні комірки (чарунки) використовують бджоли. Таким чином вони оптимально використовують простір для зберігання меду, бо шестикутники без прогалин покривають площину, при цьому розмір чарунки максимальний. Цю властивість правильного шестикутника використовують для побудови радіомережі. Якщо вежа (вишка) на якій знаходиться антена випромінює сигнал навколо себе, так, що сигнал покриває площину кола з певним радіусом, а вежа в центрі кола, тоді не вийде покрити такими колами цілком певну територію, бо між колами будуть пробіли. Радіовежі розташовують так, щоб вони утворювали

стільниковий зв'язок, тобто кожна вежа ніби знаходиться в центрі уявної шестикутної чарунки, а набір таких радіовеж утворює стільник.

Теорема відліків

Теорема відліків, також відома як Теорема Найквіста — Шеннона, є основним принципом в теорії сигналів і телекомунікацій. Ця теорема встановлює зв'язок між швидкістю дискретизації сигналу, його шириною смуги та максимальною частотою сигналу, яку можна правильно відтворити з цифрового сигналу.

Основні положення теореми виглядають приблизно наступним чином:

Швидкість дискретизації (частота дискретизації): Сигнал повинен бути дискретизований (вимірюваний) не менш як удвічі частіше, ніж максимальна частота сигналу, який вибирається.

$$f_{\text{dis}} \geq 2 * f_{\text{max}},$$

де

f_{dis} - частота дискретизації (швидкість дискретизації),

f_{max} - максимальна частота сигналу.

Ширина смуги (B) сигналу: Ширина смуги сигналу повинна бути меншою половини швидкості дискретизації.

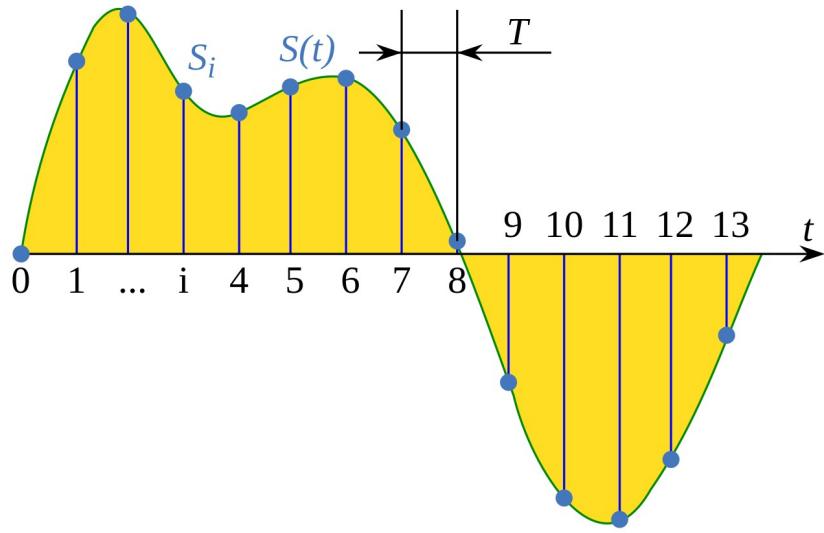
$$B \leq \frac{1}{2} * f_{\text{dis}}.$$

Ці умови важливі для того, щоб уникнути аліасингу, який може привести до втрати інформації і спотворення сигналу при його дискретизації.

Аліасинг — це явище, яке виникає при дискретизації аналогового сигналу з недостатньою швидкістю дискретизації, яка не відповідає правилам Теореми Найквіста—Шеннона. Коли сигнал недостатньо високочастотний, його високочастотні компоненти можуть спричиняти вигляд "phantom" частот, які помилково виникають при аналізі дискретизованого сигналу.

Внаслідок аліасингу низькочастотні компоненти сигналу можуть бути помилково ідентифіковані як високочастотні, що може привести до спотворення сигналу. Щоб уникнути аліасингу, необхідно забезпечити достатню високу швидкість дискретизації, яка дозволить коректно представити всі високочастотні складові сигналу. Теорема Найквіста—Шеннона визначає необхідну мінімальну швидкість дискретизації для уникнення аліасингу.

Ілюстрація дискретизації сигналу. Неперервна функція намальована зеленим кольором, а дискретна послідовність — блакитним.



T - інтервал дискретизації.

Зв'язок між інтервалом дискретизації (T) і частотою дискретизації (F_s) визначається як: $T = 1/F_s$;

Наприклад, якщо у вас частота дискретизації 10 кГц (10 000 вибірок на секунду), інтервал дискретизації буде таким: $T = 1/10\,000$;

Теорема Шеннона — Гартлі про пропускну здатність

Теорема Шеннона — Гартлі була висунута інженерами Клодом Шенноном і Ральфом Гартлі у 1948 році.

Теорема говорить про те, як швидко може бути передана інформація через канал, який піддається шумам. Основна ідея полягає в тому, що існує максимальна швидкість передачі інформації через канал із певним рівнем шуму, яку можна досягти без помилок.

Основні поняття, які використовуються в теоремі:

Пропускна здатність каналу (C, capacity): Це максимальна швидкість, з якою можна *передавати інформацію через канал без помилок*. Вимірюється у бітах на секунду.

Шум (N): Наявність шуму в каналі призводить до втрати частини інформації або до її спотворень під час передачі.

Спектральна ширина каналу (B): Це ширина діапазону частот, які може передавати канал без втрати інформації.

Теорема Шеннона — Гартлі формулюється як:

$$C = B * \log_2 (1 + S/N),$$

де C - пропускна здатність каналу (в бітах на секунду),

B - спектральна ширина каналу (в герцах),

S - потужність сигналу (в ватах),

N - потужність шуму (в ватах).

Найпростішим прикладом формулі для ширини каналу може бути формула для обчислення ширини смуги пропускання B для аналогового сигналу в радіо- або телекомунікаційних системах:

$$B = f_{\max} - f_{\min},$$

де f_{\max} — верхня частота, яку може передавати канал, і f_{\min} — нижня частота. Наприклад, якщо канал може передавати сигнали від 1000 Гц до 5000 Гц, то ширина смуги пропускання буде:

$$5000\text{Гц} - 1000\text{Гц} = 4000\text{Гц}.$$

Припустимо, що ми маємо канал зв'язку з шириною смуги пропускання $B = 10 \text{ кГц}$. Потужність сигналу S складає 100 МВт, а потужність шуму N дорівнює 1 МВт.

Отже, пропускна здатність каналу (C):

$$C = 10 \text{ кГц} * \log_2(1 + 100\text{МВт}/1\text{МВт}) = 66.7 \text{ кбіт/с.}$$

Що вище В, то більше інформації можна передати за певний період. Чим вище відношення S/N, тим сильніший сигнал щодо шуму, що дозволяє надійніше витягувати інформацію з сигналу при його прийманні. Якщо відношення сигнал-шум низьке, шум може значно спотворювати сигнал, особливо на високих частотах. У таких випадках інженери можуть обмежити використання частини смуги пропускання, щоб знизити вплив шуму на сигнал та забезпечити його правильне сприйняття приймачем.

Пропускна здатність дискретного (цифрового) каналу без шумів:

$$C = \log_2 m * Vt,$$

де m – основа коду сигналу, що використовується в каналі (кількість символів в алфавіті).

Швидкість передачі в дискретному каналі без шумів (ідеальному каналі) дорівнює його пропускній здатності, коли символи в каналі незалежні, проте m символів алфавіту рівномовірні (використовуються однаково часто). Vt - кількість символів, які можуть бути передані через канал за одиницю часу.

Коди Гемінга

Код Гемінга є кодом, який дозволяє автоматично виявити помилку, що сталася під час його передачі. Припустимо, у нас є пристрій (наприклад, комп'ютер), який сприймає програми у двійковому коді, тому його програма являє собою ланцюжок двійкового коду, наприклад, з 8 двійкових розрядів (з 8 бітів), вигляду:

10100010,
00010101,
00000001,
10101010,

При зчитуванні або передачі такого коду може виникнути помилка, в результаті якої змінюються значення кількох бітів, тобто бітів у двійкових словах. Двійкове слово в такому випадку становить вісім біт, наприклад 10100010. Код Гемінга дозволяє повністю виправити помилки в коді, якщо в кожному слові було змінено не більше одного біта, тобто не більше одного розряду в рядку з 8 розрядів.

Суть коду Гемінга полягає в тому, щоб присвоїти кожному слову контрольний біт, який ми позначаємо тире |.

10100000 | 0,
00010101 | 1,
00000001 | 1,
10101010 | 0,

Контрольний біт встановлюється таким чином, щоб загальна кількість одиниць у слові була парною або дорівнювала нулю. Таким чином, ми знаємо, що кожне слово має містити нуль або парну кількість одиниць, інакше код пошкоджений. Якщо ми оснастимо наш комп'ютер лічильником бітів, який підраховує біти за модулем два, і коли за модулем два виходить число нуль, тоді програма продовжує читати, інакше з'явиться повідомлення про помилку для виправлення коду.

Приклад “зламаного коду” Гемінга, тобто пошкодженого.

10101000 | 0, - помилка, кількість одиниць не парна.
00010101 | 1,
00000001 | 0, - помилка, кількість одиниць не парна.
10101010 | 0.

Ви можете використовувати кілька контрольних цифр, щоб відстежувати більше однієї помилки, наприклад, ви можете додати дві контрольні цифри, перша доповнить (і перевірить) одну половину двійкового слова, друга буде для перевірки другої половини слова.

Код Гемінга з двома контрольними бітами

10001010 | 10 (це означає 1000|1 та 1010|0).

“Мета обчислень – це розуміння, а не числа” (Річард Гемінг, “Чисельні методи для вчених та інженерів”) Математична формула має не тільки форму, але й зміст, або інтерпретацію. Наприклад, формула $1 + 1 = 1$ буде дивною і хибною, якщо її інтерпретувати в контексті теорії натуральних чисел,

але вона ж буде логічною в булевій алгебрі, де істина (1) + істина (1) = істина (1).

Річард Гемінг (1915 – 1998) був американським математиком та програмістом. Річард Гемінг народився в Чикаго. Будучи аспірантом, він відкрив і прочитав “Закони мислення” Джорджа Буля.

Коди Гемінга — це самоперевіряючі коди, тобто коди, які автоматично виявляють в собі помилки при передачі даних. Для їх побудови достатньо кожному двійковому слову присвоїти один додатковий двійковий реєстр і вибрати значення цього реєстру так, щоб загальна кількість одиниць у зображені будь-якого числа була, наприклад, парною. Одна помилка в будь-якому біті переданого слова змінить парність загальної кількості одиниць. Лічильники за модулем 2, які підраховують кількість одиниць, що містяться серед двійкових цифр числа, сигналізують про наявність помилок.

Американський математик Джон Тьюкі (1915-2000) відомий як автор двох комп'ютерних термінів — “софтвер” (програмне забезпечення) (1958) і “біт” (скорочення від binary digit) (1946).

Код Грэя

Код Грэя

Якщо у нас є цифровий (дискретний) пристрій зчитування, який по черзі читує кожен біт двійкового слова, наприклад, восьмирозрядне слово, протягом деякого часу t , то нам потрібно половину часу, щоб прочитати половину слова, тобто $t / 2$. Очевидно, якщо після того, як цей зчитувальний пристрій прочитає половину двійкового слова, досить швидко змінити його другу половину (тобто замінити це слово в реєстрі пам'яті або на перфокарті), то пристрій зрештою прочитати слово, що складається з двох інших. Наприклад, у нас є слово 11110000, то за час $t/2$ зчитувач прочитає 1111, і після швидкої заміни вихідного слова словом 00001111 ми отримаємо 11111111. Якщо ми маємо справу з рухомою головкою, наприклад, як у цифрових потенціометрах (енкодерах) або на магнітних жорстких дисках, то таких замін слів може бути багато за той час, коли головка читає лише одне. Код Грэя може мінімізувати ці відхилення за один крок. Уявімо, що код (звичайний двійковий) стрибає зі значення 3 \rightarrow 4, або у двійковому записі 011 \rightarrow 100. Якщо через недосконалість зчитувача ми зчитуємо перший біт з 011, а решта два зі 100, отримуємо 000 = 0 число, далеке від реальних значень. У коді Грэя не буде сторонніх значень: стрибок буде в одному біті, 010 \rightarrow 110, і ми розглядаємо або старе 010 = 3, або нове 110 = 4.

Кожен i -й біт коду Грэя G_i виражається через біти двійкового коду B_i так:

$$G_i = B_i \Delta B_{i+1},$$

де Δ — операція Виключне АБО (XOR), а біти нумеруються справа наліво, починаючи з найменшого значущого.

Число	Бінарний код	Код Грэя
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100

Коди Грэя часто застосовуються в датчиках-енкодерах. Їх використання зручно тим, що два сусідніх значення шкали сигналу відрізняються лише в одному розряді. Також вони використовуються для кодування номерів доріжок на жорстких дисках. Френк Грей (1887 – 1969) був фізиком і дослідником у Лабораторії Белла, який зробив численні інновації в телебаченні, як механічному, так і електронному, і запам'ятався кодом Грэя. Його патент 1953 року “Імпульсний кодовий зв’язок” з кодом Грэя був поданий у 1947 році.

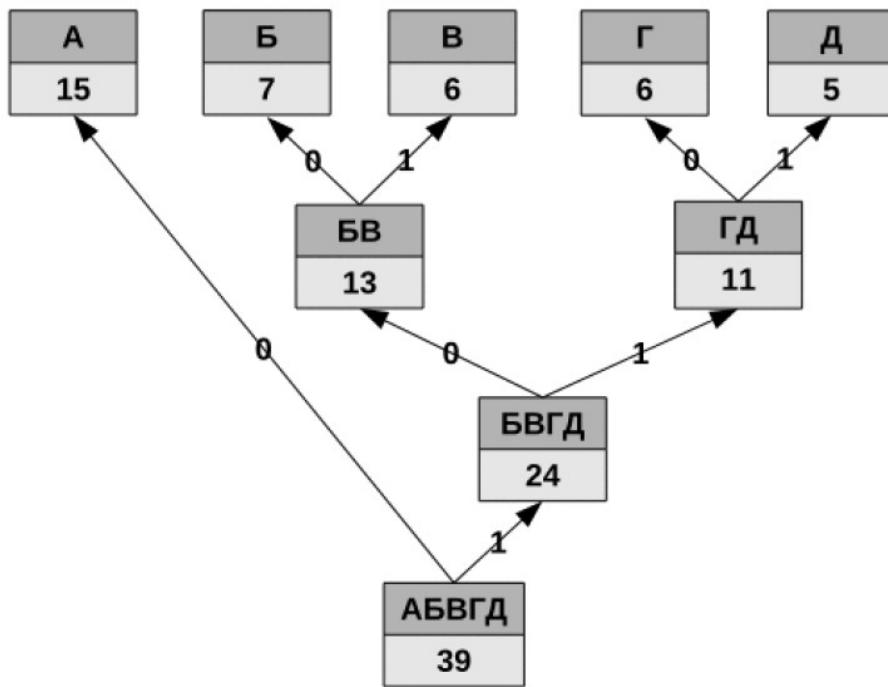
Алгоритм Гаффмана

Алгоритм Гаффмана — це алгоритм, призначений для стиснення повідомлень без втрат, після стиснення повідомлення ми отримуємо код Гаффмана. Алгоритм був розроблений аспірантом Массачусетського технологічного інституту Девідом Гаффманом під час написання ним курсової роботи та надрукований в статті 1952 року “Метод побудови кодів з мінімальною надлишковістю”.

Алгоритм заснований на наступному принципі:

1. Повідомлення надано певною мовою (скажімо, англійською).
2. Кожна літера вихідної літери зазвичай зберігається в цифровому файлі як двійкове число, двійковий код.
3. Може статися, що в тексті листа часто зустрічаються букви з найбільшою довжиною двійкового коду. (Кожна літера має власний специфічний двійковий код. Код 000 вважається меншим за 001, а 001 меншим за 011 і 111).
4. Необхідно проаналізувати повідомлення і розрахувати ймовірності появи в ньому тих чи інших літер, тобто з якою ймовірністю зустрічається та чи інша буква. (Власне, достатньо порахувати кількість кожної букви).
5. Літери, які найчастіше зустрічаються в листі, мають бути закодовані найкоротшими двійковими кодами, найрідкісніші — найдовшими.
6. Таким чином ми оптимізуємо (стиснемо) повідомлення, але текст повинен буде декодувати кінцевий адресат, для цього йому потрібно знати, з якою частотою той чи інший символ зустрічається в нашему вихідному повідомленні.

Дерево Гаффмана для присвоєння коду літері відповідно до її частоти в тексті



Итого:

A	Б	В	Г	Д
0	100	101	110	111

Дерево Гаффмана будується на основі частоти входження символів у послідовності. Основні кроки побудови:

Створення листків дерева: Кожен символ представляється як окремий листок дерева з його відповідною частотою входження.

Об'єднання символів: Обираються два символи з найнижчими частотами входження. Ці символи об'єднуються в єдиний вузол, створюючи нове піддерево, де частота цього вузла є сумою частот об'єднаних символів.

Повторення об'єднання: Цей процес повторюється, об'єднуючи на кожному кроці два символи з найнижчими частотами, поки всі символи не об'єднаються в одне дерево.

Створення бінарних кодів: Під час об'єднання символів, що відбувається у дереві, встановлюються бінарні коди для кожного символу: лівий шлях відображається як 0, правий - як 1.

Отримання дерева Гаффмана: Коли всі символи об'єдналися в одне дерево, воно і утворює дерево Гаффмана.

У кінцевому дереві листки представляють символи, а шлях від кореня дерева до кожного листка представляє код, який ідентифікує символ. Символи, які зустрічаються частіше, будуть мати меншу довжину коду, тоді як менш часті символи матимуть довші коди.

Код Гаффмана є префіксним, що означає відсутність будь-якого символу, який є префіксом іншого символу в коді.

Префіксний код в теорії кодування — код зі словом змінної довжини, що має таку властивість (виконання умови Фано): якщо в код входить слово а, то для будь-якого непорожнього рядка b слова ab в коді не існує. Хоча префіксний код складається зі слів різної довжини, ці слова можна записувати без розділового символу.

Розглянемо приклад префіксного коду за допомогою коду Гаффмана.

Для алфавіту {A, B, C, D, E} з ймовірностями {0.4, 0.3, 0.2, 0.1, 0.05}, коди Гаффмана будуть:

A: 0,

B: 10,

C: 110,

D: 1110,

E: 1111,

Це приклад префіксного коду, оскільки немає жодного коду, який є префіксом іншого. Наприклад, код для символу "A" (0) не є префіксом для будь-якого іншого коду (наприклад, "10", "110" тощо).

Математично, ми можемо сформулювати умову для множини префіксних кодів наступним чином:

$$\forall i, j \in \{1, 2, \dots, n\}, i \neq j : b_i \notin \{b_j\},$$

де $\{b_j\}$ – множина всіх префіксів b_j ,

$\{b_1, b_2, \dots, b_n\}$ – множина дозволених кодів (префіксних кодів).

Кодування довжин серій

Кодування довжин серій (Run-length encoding, RLE) — це форма стиснення даних без втрат, у якій серії даних (послідовності, в яких один і той же символ зустрічається кілька разів поспіль) зберігаються як одне значення даних і число повторів.

Таке кодування ефективно для даних, що містять велику кількість серій, наприклад, для зображень.

Простий приклад роботи алгоритму RLE (кодування довжин серій):

Нехай у нас є невелике зображення, представлене послідовністю пікселів:

[255, 255, 255, 255, 255, 0, 0, 0, 0, 255, 255, 255, 0, 0, 0, 0, 0];

З використанням алгоритму RLE ми можемо представити це зображення як послідовність пар (значення, кількість):

[(255, 5), (0, 5), (255, 3), (0, 5)];

Можна реалізувати додаткове налаштування для стискання з втратами, де два сусідні значення, які мають малу різницю, вважаються однаковими. Це називається квантуванням, і воно може допомогти ще більше зменшити розмір зображення за рахунок втрати деякої інформації про точний колір кожного пікселя. Наприклад, замість того, щоб зберігати кожен піксель точно, можна округлити кожне значення до певного діапазону, щоб зменшити кількість різних кольорів, які потрібно зберігати. Такий підхід дозволяє досягти ще більшого стиснення, але при цьому втрачається деяка якість зображення.

Під час квантування кожен піксель зображення призначається до певного кольору або відтінку з обмеженою набору доступних значень. Цей процес може бути виконаний з різними методами, такими як рівномірне квантування, кластеризація кольорів, або застосування попередньо визначених палітр.

Звичайні формати для зображень включають JPEG, який надає хороше стиснення та ідеальний для фотографій; PNG, який підтримує прозорість та відмінно підходить для веб-зображень; та GIF, що використовується для анімації. Для збереження деталей і високої якості, TIFF є відмінним вибором, особливо для професійних фотографій. У той час як SVG ідеально підходить для векторних графічних елементів, які можна масштабувати без втрати якості, ідеально для логотипів та іконок.

Які принципи розробки інтерфейсу користувача (UI/UX)?

"Розробнику варто витратити тиждень на спрошення інтерфейсу, а не змушувати тисячі користувачів витрачати хвилини на виконання складних дій. Час користувача повинен бути в пріоритеті"

(Ларрі Теслер, програміст в Xerox Park та Apple)

"Машина має підлаштовуватися під людину, а не людина під машину"

(Стів Возняк)

Щоб користуватися інтерфейсом потрібно:

1. Інструкцію для інтерфейсу (manual, tips, wizards).

До того ж дизайн інтерфейсу повинен заливати common sense та опиратися на попередній досвід користувача (інтуїцію).

2. Доступність інтерфейсу (допоможуть ARIA-атрибути, screen readers, AI). Інтерфейс повинен бути контролюваним.

Доступність (accessibility) — це насамперед доступність для людей, а не для машин. Якщо ваш сайт доступний для машини (скрінрідера, чи штучного інтелекту) це ще не означає, що він доступний для людей.

Ви можете створювати програму чи прилад для вашого кота (cat), отже вам потрібно врахувати, що коти не можуть рухати очима (вони рухають головою) і не можуть брати щось в лапи. Доступний інтерфейс для кота тоді, коли кіт здатен працювати з ним самостійно.

Рекомендації щодо доступності вебмісту (WCAG) визначають, як зробити вебміст (контент) більш доступним.

Стів Джобс висловив таку думку: Деякі люди кажуть: "Дайте клієнтам те, що вони хочуть". Але я пам'ятаю, Генрі Форд колись сказав: "Якби я запитав клієнтів, що вони хочуть, вони б сказали мені, 'Швидшого коня!'" Наше завдання — зрозуміти, що люди захочуть до того, як вони це побачать.

User stories (юзер сторіс) — це короткі описи функціонала чи можливостей з погляду кінцевого користувача. Зазвичай вони мають простий шаблон: "Як [роль користувача], я хочу [ціль] для того, щоб [причина]". Вони допомагають командам розробників програмного забезпечення зрозуміти потреби та вимоги користувачів чітко.

Сценарій користувача — це опис того, як користувач взаємодіє з системою або програмою для

досягнення певної мети чи виконання завдання. Зазвичай він описує кроки, які користувач виконує, а також можливі дії, рішення чи результати, які можуть виникнути під час взаємодії. Сценарії користувача допомагають дизайнерам та розробникам розуміти поведінку користувачів і створювати досвід, який ефективно задовольняє їхні потреби.

Ось простий приклад сценарію користувача:

Сценарій: Замовлення піци

- Крок 1: Користувач відкриває додаток піцерії.
- Крок 2: Користувач обирає "Піца" з меню.
- Крок 3: Користувач обирає "Пепероні піца" та додає її до кошика.
- Крок 4: Користувач натискає кнопку "Оформити замовлення".
- Крок 5: Користувач вводить адресу доставки та обирає спосіб оплати.
- Крок 6: Користувач підтверджує замовлення.
- Крок 7: Піцерія отримує замовлення та готує піцу для доставки.
- Крок 8: Кур'єр доставляє піцу до дверей користувача.

Цей сценарій показує послідовність кроків, які користувач виконує, щоб замовити піцу через додаток піцерії. Його можна оформити у вигляді діаграми.

A/B тестування — це метод порівняльного аналізу, який використовується в маркетингу та веброзробці для порівняння двох версій продукту чи рекламної стратегії: версії А (контрольна) та версії В (експериментальна). Під час тестування користувачі випадковим чином розділяються на дві групи, одній з яких показується версія А, а іншій — версія В. Потім аналізуються результати, щоб визначити, яка версія привертає більше уваги користувачів, генерує більше конверсій або має кращі показники ефективності. A/B тестування дозволяє здійснювати обґрунтовані рішення щодо покращення продукту чи рекламної кампанії на основі даних та експериментів.

Ларрі Теслер проводив A/B тестування комп'ютерної мишки в 1970-х роках в Xerox Parc. Він виявив, що людям зручніше працювати з мишкою та клавіатурою, ніж тільки з клавіатурою.

Комп'ютер Xerox Alto мав клавіатуру QWERTY. Американський винахідник Крістофер Шоулз створив розкладку "QWERTY" для клавіатури друкарської машинки у 1870-х роках.

Що таке кнопка?

Якщо ми маркуємо елемент інтерфейсу назвою "кнопка", тоді це номінальний підхід до розпізнавання елементів.

Але номінальний підхід до розпізнавання ролі елемента не є єдиним рішенням; натомість ми можемо використати структурну перевірку ролі (якщо елемент інтерфейсу поводиться як кнопка, він є кнопкою).

Кнопка — це елемент інтерфейсу, який використовується для виклику або активації якої-небудь функції, зазвичай шляхом натискання (клацання).

Якщо елемент інтерфейсу після натискання викликає якусь функцію, він є кнопкою. Важливо, що

кнопка спрацьовує саме на натискання, а не на інші дії.

UI (інтерфейс користувача) та UX (досвід користувача) — це два різні аспекти дизайну продукту. UI займається тим, як виглядає продукт, його естетичними аспектами та інтерактивними елементами. UX, з іншого боку, фокусується на тому, як користувачі взаємодіють з продуктом, їхніми враженнями, задоволенням та ефективністю цієї взаємодії.

Задача UI дизайнера створити UI елементи. UX дизайнер розробляє User scenarios (Сценарій користувача), тобто ланцюжки дій користувача (юзера) для отримання певного результату, та інформаційну архітектуру.

Інформаційна архітектура — це структурний дизайн інформації в системі, вебсайті або додатку, який спрямований на організацію, структурування та позначення контенту таким чином, щоб користувачам було легко знаходити інформацію. Це включає створення інтуїтивних шляхів для доступу до інформації, часто за допомогою таких технік, як категоризація, системи навігації та схеми позначення.

Приклад інформаційної архітектури:

Головна сторінка

- Новини
- Популярні пости
- Категорії
 - Політика
 - Технології
 - Мода
- Про нас
- Контакти

Категорія "Політика"

- Стаття 1
- Стаття 2
- Стаття 3

Категорія "Технології"

- Стаття 1
- Стаття 2
- Стаття 3

Категорія "Мода"

- Стаття 1
- Стаття 2
- Стаття 3

Це все можна зобразити графічно.

WYSIWYG (візівіг), тобто "What You See Is What You Get" (що ви бачите, те й отримуєте) - це тип візуального редактора. WYSIWYG (візівіг) описує інтерфейс, де користувач може бачити результати своєї роботи у відформатованому вигляді, а не у вигляді розмітки або коду. Наприклад, текстовий процесор з WYSIWYG дозволяє користувачам форматувати текст, як вони його бачать на екрані, без необхідності переглядати або редагувати код.

Один із найпоширеніших прикладів WYSIWYG - Microsoft Word. Користувачі можуть форматувати текст, вставляти графіку та таблиці, встановлювати вирівнювання тощо, і все це вони бачать у режимі, який відображає документ так, як він буде виглядати при друку або публікації, без необхідності редагувати або переглядати код.

Наприклад, додаток Adobe Dreamweaver (на ринку з 1997) має вбудований візуальний редактор, який дозволяє користувачам створювати та редагувати сторінки за допомогою інтерфейсу WYSIWYG (візівіг). Вони можуть додавати тексти, зображення, відео, кнопки та інші елементи, бачачи результати в реальному часі.

Drag-and-drop (тягни та кидай) є методом взаємодії з користувацьким інтерфейсом, який дозволяє користувачеві пересувати об'єкти на екрані, виділяючи їх та перетягуючи з одного місця на інше за допомогою комп'ютерної миші або сенсорного екрана. Наприклад, це може бути використано для пересування файлів або папок в операційних системах або для розташування об'єктів на вебсайтах або в програмах для редагування графіки.

Правила UX:

1. Мінімізуй кількість кліків (клацань) та рухів між діями. Правило ергономіки.
2. Інформуй користувача перед важливими діями (видалення, закриття, надсилання).
3. Інформуй юзера про статус виконання. Використовуйте індикатори прогресу та нотифікації (сповіщення) для відображення стану процесу.
4. Додай зручну навігацію між станами в обидва боки. Кнопку вперед та назад.
5. Додай кнопку "пропустити", "вийти".

6. Відкривай все в нових вікнах, якщо користувачу (юзеру) потенційно необхідне попереднє вікно.

Правило для посилань: Якщо посилання переходить в межах твого сайту — відкриваємо його у тій самій вкладці, якщо ж воно веде на інший ресурс — відкриваємо посилання в іншій вкладці.

Якщо перехід за посиланням може привести до втрати інформації у формі тощо, рекомендується відкривати його в новій вкладці, щоб зберегти поточний контекст користувача.

7. Не змушуйте юзера вгамовувати додаток перед користуванням. Наприклад, якщо юзер відкриває вебсторінку, а там одразу вмикається мелодія й з'являються різні віконця, які потрібно закрити перед використанням сайту, тоді користувач буде нервуватися.
8. Використовуйте фонові знання (досвід) користувача (юзера) для того, щоб зробити ваш інтерфейс інтуїтивним.

9. Якщо ви створюєте список елементів, потрібно логічно їх впорядкувати за алфавітом, за часом, за пріоритетом тощо.

10. Стандартизуйте інтерфейс, використовуйте узгоджену сітку, відступи й кольорову палітру.

11. Закон необхідної різноманітності: Закон вимагає, щоб різноманітність управлінських дій, що виходять від керуючого органу, була не менше за різноманітність можливих змін керованого об'єкта. В іншому випадку керований об'єкт вийде з-під контролю.

Правила UI:

1. Кольори не мають конфліктувати (Кольорове коло Іттена).
2. Дизайн повинен бути читабельним, чітким.
3. Не покладайтесь тільки на колір, використовуйте також текст для індикації. Пам'ятати, що під час друку вашої вебсторінки деякі кольори можуть зникнути.
4. Дизайн повинен бути консистентним. Збереження одинакового стилю та поведінки елементів інтерфейсу по всьому додатку або вебсайту. Консистентність зменшує складність.

5. Все що блимає, мерехтить, буде відволікати та розфокусовувати увагу юзера. Юзеру буде важко читати статичний текст, якщо поряд щось блимає і стрибає.

6. Групуйте елементи інтерфейсу відповідно їх призначення. Юзер повинен робити якомога менше рухів очима, коли він виконує пов'язані дії.

Коли людина виконує послідовні дії, зменшення рухів очима допомагає зберегти час та енергію, оскільки користувач не витрачає час на пошук потрібних елементів на екрані. Це може бути досягнуто, наприклад, розташуванням схожих функцій поруч, щоб користувачу було зручно та ефективно ними користуватися без зайвих переривань.

7. Вирівнюйте елементи логічно й консистентно, що юзер візуально міг групувати їх.

Наприклад, кнопки або текстові блоки можуть бути розміщені відповідно до сітки або стандартних відстаней, щоб забезпечити одинаковий вигляд та легку навігацію. Користувачу (юзеру) легше запам'ятовувати такий інтерфейс.

8. Інтерфейс не повинен контрастно *блімати більше ніж три рази за секунду*, бо це може викликати епілептичний припадок у людей зі склонністю до епілептичних припадків (WCAG2.1).

9. Закон Вебера-Фехнера: Відчуття пропорційне логарифму інтенсивності стимулювання.

Закон Міллера

Закон Міллера — це концепція, яку часто цитують у сфері дизайну користувальського досвіду (UX). У ньому стверджується, що середня людина може зберігати в робочій пам'яті лише близько 7 (плюс-мінус 2) елементів одночасно. Цей принцип має значні наслідки для розробників UX, оскільки передбачає, що інтерфейси повинні прагнути подавати інформацію в керованому та доступному вигляді, уникнути перевантаження користувачів надто великою кількістю інформації одночасно.

У UX-дизайні дотримання закону Міллера може передбачати розбиття складних завдань або інформації на менші, більш керовані частини, організацію вмісту в ієрархічний спосіб, забезпечення чіткої навігації та мінімізацію когнітивного навантаження завдяки простоті та ясності. Поважаючи обмеження робочої пам'яті, дизайнери можуть створювати інтерфейси, які є більш інтуїтивно зрозумілими, ефективними та зручними для користувача.

До такого висновку Джордж Міллер дійшов шляхом серії експериментів і спостережень. Він виявив, що людям, як правило, важко обробити та запам'ятати більше ніж сім фрагментів інформації одночасно. Наприклад, коли їх просили згадати список випадкових чисел або букв, люди зазвичай виконували найкращі результати, коли список містив близько семи елементів. Це спостереження змусило Міллера припустити, що наша робоча пам'ять обмежена, що впливає на нашу здатність обробляти та запам'ятувати інформацію.

Дослідження Міллера включали різні експерименти, включно з завданнями на об'єм пам'яті, де учасників просили згадати послідовності цифр, літер або інших стимулів. Завдяки цим експериментам він спостерігав закономірності поведінки людей, особливо відзначаючи, що обсяг їхньої пам'яті, як правило, групується навколо семи предметів.

Одним із ключових понять у галузі когнітивної науки є робоча пам'ять, яку часто називають короткочасною пам'яттю. Короткочасна пам'ять — це вид пам'яті, що характеризується дуже коротким збереженням після одноразового сприйняття і миттєвим відтворенням.

Міллер провів кілька тестів, щоб дослідити межі ємності робочої пам'яті. Один із відомих експериментів передбачав надання учасникам послідовностей випадкових чисел або літер, а потім їх просили згадати послідовність одразу після презентації. Він змінював довжину послідовностей, щоб побачити, скільки предметів учасники могли точно згадати.

В одному варіанті Міллер виявив, що учасники могли запам'ятати в середньому близько семи предметів, плюс-мінус два предмети. Це привело до формулювання “закону Міллера” або “магічного числа сім, плюс-мінус два”. Однак він також зауважив, що учасники могли згадати більше предметів, якщо предмети були організовані у значущі групи або шматки.

Міллер запропонував концепцію фрагмента або одиниці інформації, яка є зв'язною для учасника, припускаючи, що межа пам'яті становить приблизно сім знайомих фрагментів. Наприклад, літерний рядок ФБРЦРУСША можна запам'ятати без особливих труднощів, якщо його розібрати на 3 частини, кожна з яких є акронімом, що представляє американське агентство, якщо ви його знаєте: ФБР, ЦРУ та США. Ця ідея про блоки, а не обмеження приблизно до 7 елементів, може бути найважливішим конкретним внеском статті.

В експериментах Міллера учасники зазвичай згадували послідовність відразу після презентації. Міллер також досліджував здатність робочої пам'яті в контексті музичних тонів. В одному експерименті учасникам було запропоновано прослухати послідовність музичних тонів, а потім згадати їх відразу після того, як їх почули. Подібно до тестів із цифрами та літерами, Міллер виявив, що учасники можуть точно запам'ятати близько семи тонів плюс-мінус два.

Міллер проводив своє дослідження обмежень ємності робочої пам'яті переважно в 1950-х роках, коли він був професором Принстонського університету. Його основоположна стаття під назвою “Магічне число сім, плюс-мінус два: деякі обмеження нашої здатності опрацьовувати інформацію” була опублікована у 1956 році.

Чотири принципи доступності (Accessibility):

1. Сприйнятливий інтерфейс (Perceivable interface).

Користувачі повинні мати можливість сприймати представлену інформацію.

2. Інтерактивний інтерфейс (Operable interface).

Користувачі повинні мати можливість працювати з інтерфейсом (інтерфейс не може вимагати взаємодії, яку не може виконати користувач)

3. Зрозумілий інтерфейс (Understandable interface).

Користувачі повинні розуміти інформацію, а також роботу інтерфейсу користувача.

4. Надійний інтерфейс (Robust interface).

Користувачі повинні мати доступ до вмісту в міру розвитку технологій (у міру розвитку технологій і агентів користувача вміст повинен залишатися доступним)

Вміст має бути достатньо надійним, щоб його можна було надійно інтерпретувати різноманітними агентами користувача, включаючи допоміжні технології.

Витримана семантика розмітки, яку можна автоматично парсити.

Якщо щось з цього не відповідає дійсності, користувачі з обмеженими можливостями не зможуть користуватися Інтернетом («Web Content Accessibility Guidelines 2.1»).

"Надлишкове число" 12

Отже, дільники числа дванадцять: 1, 2, 3, 4, 6 і 12. Це робить його зручним для створення сітки (grid) інтерфейсу.

Наприклад, якщо вам потрібно розмістити елемент ширший, ви можете використати 6 колонок; якщо менший - 3 колонки, і так далі. Це дозволяє оптимізувати макет для різних пристрій, від комп'ютерів до мобільних пристройів.

Надлишкове число — натуральне число n , сума додатних дільників (відмінних від n) якого перевищує n . Найменшим надлишковим числом є 12.

$$12 = 10 + 2$$

													$12 = 1 \times 12$
													$12 = 2 \times 6$
													$12 = 3 \times 4$
													$12 = 4 \times 3$
													$12 = 2 \times 6$

$$1 + 2 + 3 + 4 + 6 = 16 (>12)$$

Author of image: Hyacinth; License: CC BY-SA 4.0;

Ця сітка використовується для створення структурованого та гнучкого дизайну на вебсайтах.

Що таке штучний інтелект?

“Штучний інтелект — здатність інженерної системи здобувати, обробляти, створювати та застосовувати знання та навички” (ISO/IEC TR 29119-11)

Є декілька визначень штучного інтелекту:

- Штучний інтелект — це машина, яка може приймати рішення на основі набору вхідних параметрів.
- Штучний інтелект — це машина, яка здатна вчитись, тобто аналізувати й запам'ятовувати інформацію, щоб приймати рішення в майбутньому.
- Штучний інтелект — це машина, яка здатна проходити тест Тюрінга.

Одне визначення вбачає Штучний інтелект як систему, що в змозі приймати рішення на основі вхідних даних, що відбувається у багатьох традиційних програмах.

Інше визначення підкреслює навчання як ключовий аспект штучного інтелекту. Це стосується здатності системи аналізувати та запам'ятовувати інформацію для подальшого використання у майбутніх рішеннях. Це є важливою складовою сучасних методів штучного інтелекту, таких як машинне навчання та нейронні мережі.

Третє визначення посилається на Тест Тюрінга, запропонований Алланом Тюрінгом. Цей тест має на меті визначити, наскільки система може виконувати інтелектуальні функції, які можна вважати еквівалентними людським. Це необхідно, щоб система могла взаємодіяти з людьми настільки вправно, що людина не змогла відрізнити цю систему від іншої людини під час розмови через текстовий інтерфейс.

Ян ЛеКун виділяє 4 властивості людського інтелекту:

1. Здатність пам'ятати,
2. Здатність планувати,
3. Здатність міркувати,
4. Здатність розуміти світ.

Ознакою рівня інтелекту є кількість параметрів, які здатен опрацьовувати суб'єкт для прийняття рішень, та кількісна ознака оптимальності цих рішень.

Більш розгорнуто визначення інтелекту можна сформулювати так: “Інтелект — це здатність сприймати, аналізувати (знаходить шаблони, подібності й взаємозв'язки) й використовувати набуті знання для конструювання й розв'язання різних життєвих задач. Ці здібності дають можливість ‘розуміти світ’, в певному значенні цього виразу”.

Зверніть увагу, що рівень інтелекту можна оцінювати кількісно і якісно. Якісна відмінність інтелекту проявляється тоді, коли одна особа (або прилад) може включати у свій аналіз ті параметри, які інша особа не може використовувати в принципі.

Навчання — процес набуття знань та досвіду.

“Свідомість — це процес створення моделі світу з використанням множинних циклів зворотного зв’язку за різними параметрами (наприклад, у температурі, просторі, часі та по відношенню до інших), щоб досягти мети (наприклад, знайти пару, їжу, притулок). Наприклад, найнижчим рівнем свідомості є рівень 0, коли організм нерухомий або має обмежену рухливість і створює модель свого місця, використовуючи петлі зворотного зв’язку за кількома параметрами (наприклад, температурою)” (Мічіо Кайку, "Майбутнє розуму")

Свідоме (мисляче), по Декарту, це те, що може сприймати й аналізувати інформацію, здатне вчитись й пам'ятати (Рене Декарт, "Принципи філософії", 1644).

Термін “інтелект” та “свідомість” можуть вживати як синоніми, але зазвичай свідомість ширший термін, який охоплює також інтелект.

"Я мислю, отже я існую" (Рене Декарт)

Рене Декарт вважав, що свідомість — це здатність розуміти, мислити, почувати, усвідомлювати світ навколо себе і своє власне існування.

Аврелій Августин під час диспуту з академіками використав фразу: "Якщо Я можу помилитись, отже я існую, (інакше я б не міг думати й помилитись)". Вислів: "Я знаю, що існую", є істиною сам по собі, бо його не можливо спростувати, адже той хто може виносити судження, тобто думати, повинен існувати. Сам факт того, що людина думає для неї очевидний, отже вона може твердо сказати, що вона існує. Подібний вислів зустрічається в книгах Декарта "Розсуд про метод" (1637) і "Принципи філософії" (1644).

“Довгий час багато експертів вважали, що штучний інтелект на рівні людини може бути досягнутий шляхом створення вручну досить великого набору чітких правил для маніпулювання знаннями та прийняття рішень. Цей підхід відомий як “Символічний ШІ” (Symbolic AI), і він був домінуючою парадигмою ШІ з 1950-х до кінця 1980-х років. Як бачите, підхід машинного навчання принципово відрізняється від підходу символічного ШІ.

Тоді як символічний штучний інтелект спирається на знання та правила жорсткого кодування, машинне навчання прагне уникнути цього жорсткого кодування. Отже, якщо машина не отримує чітких інструкцій щодо виконання завдання, як вона навчиться цьому? Відповідь полягає в навчанні на прикладах” (Eric D. Nielsen, Shanqing Cai, “Deep learning with javascript”, 2020)

Логічний теоретик (Logic Theorist) - це комп’ютерна програма (Символічний ШІ), написана в 1956 році Алленом Ньюеллом, Гербертом Саймоном і Кліфом Шоу. Герберт Саймон (1916 — 2001) був лауреатом премії Тюрінга та Нобелівської премії.

Логічний теоретик (Logic Theorist) був першою програмою, спеціально розробленою для автоматизованого мислення. “Логічний теоретик” довів 38 з перших 52 теорем другого розділу “Принципів математики” Альфреда Вайтгеда та Бертрана Рассела, і знайшов нові та коротші докази для деяких з них.

Перед використанням нейронної мережі дуже важливо визначити відповідну модель мережі та методи уникнення “перенавчання” (overfitting).

Ось приклад елементарної нейронної мережі, мовою JavaScript, для здійснення операції XOR. Ця мережка працює з числовими даними (1, 0).

Це нейронна мережа прямого поширення (feedforward neural network). У такій мережі інформація переміщується лише у напрямку від входних вузлів до вихідних вузлів без зворотного зв'язку. Кожен вузол (нейрон) у прихованому шарі отримує сигнали від вузлів попереднього шару, обчислює лінійну комбінацію входних сигналів з вагами та зміщенням (bias), і потім застосовує до результату нелінійну функцію активації. Результати цього обчислення передаються наступному шару, де процес повторюється, поки не досягнемо вихідного шару.

```
// Функція активації - сігмоїда
function sigmoid(x) {
    return 1 / (1 + Math.exp(-x));
}

// Похідна функції активації
function sigmoidDerivative(x) {
    return x * (1 - x);
}

class NeuralNetwork {
    constructor(inputNodes, hiddenNodes, outputNodes) {
        this.inputNodes = inputNodes;
        this.hiddenNodes = hiddenNodes;
        this.outputNodes = outputNodes;

        // Випадкова ініціалізація ваг
        this.weightsInputHidden = Array(this.hiddenNodes)
            .fill()
            .map(() => Array(this.inputNodes).fill().map(() => Math.random() - 0.5));
        this.weightsHiddenOutput = Array(this.outputNodes)
            .fill()
            .map(() => Array(this.hiddenNodes).fill().map(() => Math.random() - 0.5));

        // Зсуви
        this.biasHidden = Array(this.hiddenNodes).fill().map(() => Math.random() - 0.5);
        this.biasOutput = Array(this.outputNodes).fill().map(() => Math.random() - 0.5);
    }

    // Метод навчання
    train(inputs, targets, learningRate) {
        // --- Feedforward ---

        // Вхідні дані -> Прихований шар
        const hiddenInputs = this.weightsInputHidden.map(weights =>
            weights.reduce((sum, weight, i) => sum + weight * inputs[i], 0)
        );
        const hiddenOutputs = hiddenInputs.map(sigmoid);
```

```

// Прихований шар -> Вихідний шар
const finalInputs = this.weightsHiddenOutput.map(weights =>
    weights.reduce((sum, weight, i) => sum + weight * hiddenOutputs[i], 0)
);
const finalOutputs = finalInputs.map(sigmoid);

// --- Backpropagation ---

// Похибки на виході
const outputErrors = finalOutputs.map((output, i) => targets[i] - output);

// Похідна функції активації для вихідного шару
const outputGradients = finalOutputs.map(sigmoidDerivative);

// Зміна ваг між прихованим і вихідним шарами
const weightsHiddenOutputDeltas = outputErrors.map((error, i) =>
    hiddenOutputs.map(output => output * error * outputGradients[i] * learningRate)
);
this.weightsHiddenOutput = this.weightsHiddenOutput.map((weights, i) =>
    weights.map((weight, j) => weight + weightsHiddenOutputDeltas[i][j])
);

// Похибки прихованого шару
const hiddenErrors = this.weightsHiddenOutput.reduce((errors, weights, i) =>
    errors.map((error, j) => error + weights[j] * outputErrors[i]),
Array(this.hiddenNodes).fill(0)
);

// Похідна функції активації для прихованого шару
const hiddenGradients = hiddenOutputs.map(sigmoidDerivative);

// Зміна ваг між вхідним і прихованим шарами
const weightsInputHiddenDeltas = hiddenErrors.map((error, i) =>
    inputs.map(input => input * error * hiddenGradients[i] * learningRate)
);
this.weightsInputHidden = this.weightsInputHidden.map((weights, i) =>
    weights.map((weight, j) => weight + weightsInputHiddenDeltas[i][j])
);
}

// Метод передбачення
predict(inputs) {
    // Вхідні дані -> Прихований шар
    const hiddenInputs = this.weightsInputHidden.map(weights =>
        weights.reduce((sum, weight, i) => sum + weight * inputs[i], 0)
    );
    const hiddenOutputs = hiddenInputs.map(sigmoid);

    // Прихований шар -> Вихідний шар
    const finalInputs = this.weightsHiddenOutput.map(weights =>
        weights.reduce((sum, weight, i) => sum + weight * hiddenOutputs[i], 0)
    );
    const finalOutputs = finalInputs.map(sigmoid);
}

```

```

        return finalOutputs;
    }
}

// Вхідні дані та очікувані виходи для функції XOR
const xorInputs = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
];
const xorTargets = [
    [0],
    [1],
    [1],
    [0]
];

const inputNodes = 2;

// Створення мережі
const nn = new NeuralNetwork(inputNodes, inputNodes + 1, 1);

// Навчання мережі
for (let i = 0; i < 100000; i++) {
    const index = Math.floor(Math.random() * xorInputs.length);
    nn.train(xorInputs[index], xorTargets[index], 0.1);
}

// Передбачення
xorInputs.forEach(input => {
    console.log(`Input: ${input} Output: ${nn.predict(input)}`);
});

```

Гра в імітацію (Тест Тюрінга)

Гра в імітацію, або імітаційна гра, — це спосіб оцінки штучного інтелекту шляхом співставлення його поведінки з поведінкою людини. Найбільш відомий приклад — Тест Тюрінга, який можна розглядати як форму імітаційної гри.

У такій грі людина взаємодіє з двома "суперниками" через текстовий інтерфейс: один — це інша людина, а інший — система штучного інтелекту. Задача полягає у тому, щоб людина не могла відрізнити відповіді штучного інтелекту від відповідей іншої реальної людини.

Гра в імітацію — це спосіб перевірити, наскільки добре програма штучного інтелекту може подавати себе за людину під час спілкування через текстовий інтерфейс. Її мета — створити ситуацію, де суддя не може відріznити відповіді іншої людини від відповідей програми.

CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart - Повністю автоматизований публічний тест Тюрінга для розрізнення комп'ютерів і людей) — це метод, який використовується для визначення, чи користувач є людиною чи комп'ютерною програмою.

CAPTCHA (Кептча) містить завдання, які легко вирішувати людині, але складно для автоматизованих систем, наприклад, розпізнавання тексту на зображенні, розпізнавання звуку тощо. Це допомагає запобігти автоматизованому надходженню спаму та зламам.

Коли ви реєструєтесь на вебсайті, може з'явитися CAPTCHA, щоб забезпечити, що новий акаунт (обліковий запис) створюється реальною людиною, а не автоматизованою програмою.

Млин Лейбніца

У своїй книзі 1714 року “Монадологія” Лейбніц описує мисленнєвий експеримент, який тепер можна інтерпретувати як аргумент проти можливості штучного створення мислячої істоти. Цей експеримент називається “Млин Лейбніца”, а його суть полягає в наступному: “Якби була машина, сконструйована таким чином, щоб думати, відчувати та сприймати, її можна було б уявити як збільшенну в розмірах, зберігаючи при цьому ті самі пропорції, щоб можна було зайди в ній, як у млин. Якщо це так, ми повинні, досліджуючи її внутрішню частину, знаходити лише частини, які діють одна на одну, і ніколи нічого, що могло б пояснити сприйняття. Отже, сприйняття потрібно шукати в простій субстанції, а не в складі чи в машині” (“Монадологія”, 16).

За Лейбніцом, навіть якщо ми змогли б створити машину, яка могла б симулювати поведінку свідомої істоти, ми все одно не могли б вважати її справжнім мислителем.

Велика мовна модель (LLM від large language model), як штучний інтелектуальний агент, може взаємодіяти з вами, формулювати відповіді та виконувати завдання, але її "мислення" й "сприйняття" відрізняється від того, яке мають люди.

Мозок людини

Людський зір дискретний, тобто людина сприймає обмежену кількість кольорів та променів світла. Людське око сприймає три кольори, а їхні комбінації породжують всі інші кольори. Люди сприймають обмежену кількість кольорів через три типи колбочок, які реагують на різні довжини хвиль світла. Це утворює основний колірний спектр - червоний, зелений і синій. Людський зір інертний, тобто деякі зображення можуть зникати з затримкою, хоча насправді світлові промені від них вже не потрапляють на сітківку. Око має систему фокусування. Світлові промені від предмета потрапляють в око і проектируються на сітківку на якій нервові клітини, рецептори, зокрема, палички та колбочки. Зображення проєктується на сітківку відображене по горизонталі та вертикалі. Тобто, якщо ви дивитесь на людину, то голова людини потрапить на нижню частину вашої сітківки, а права рука на ліву частину сітківки. Інтерпретація зображення відбувається в мозку людини досить складним чином, наприклад, на сітківці людини є сліпа зона, тобто місце яке не сприймає зображення, але людина не помічає цю пляму завдяки тому, що мозок її згладжує, приирає. Людина також не помічає прогалини між своїми очима, там де розміщений ніс. Крім того, мозок отримує зображення з обох очей і це дозволяє їйому аналізувати відстань до об'єктів, тобто людина має стереоскопічний зір. Зоровий нерв від правого ока йде в задню ліву частину мозку, а від лівого ока нерв веде до правої задньої частини мозку. В мозку людини були виявлені зони, які відповідають за певні здібності, наприклад за мову, зір, координацію. Те що ми називаємо свідомістю людини також має певну локацію в мозку, наприклад, людина може бути свідомою навіть, якщо в неї відсутній мозочок (церабелум, який в нижній частині потилиці). Відкриття математичних співвідношень між струнами свідчило про те, що існують певні закони слухового сприйняття. Пізніше, це породило уявлення про мозок, як апарат, що обробляє певні коливання і сприймає їх як гарні, тобто консонанси, а інші, як негарні, тобто дисонанси, або навіть шум.

Зона Брука — ділянка кори головного мозку, названа на честь французького антрополога і хірурга Поля Брука, що відкрив його в 1865 році. Центр розташований у задньонижній частині третьої лобової звивини лівої півкулі (у правшів), роботою якого забезпечується моторна організація мовлення і переважно пов'язана з фонологічними та синтаксичними кодифікаціями.

Мозок складається з нейронів.

Нейрон — це основна структурна та функціональна одиниця нервової системи, яка відповідає за передачу інформації у вигляді нервових імпульсів. Нейрони є основними будівельними блоками мозку та інших частин нервової системи людини та інших тварин.

Нейрони з'єднані між собою за допомогою великої кількості відгалужень, які називаються аксонами та дендритами. Дендрити приймають сигнали від інших нейронів чи від зовнішніх джерел, тоді як аксони передають сигнали до інших нейронів чи ефекторних клітин (таких як м'язи чи залози).

Одна з ключових властивостей нейронів - це їхні здатності генерувати електричні сигнали та передавати їх через синапси (переходи між нейронами) за допомогою хімічних сигналів. Цей процес є основою для передачі інформації у нервовій системі.

Нейромедіатори — це хімічні речовини, які використовуються для передачі сигналів в синаптических щілинах, де взаємодіють нейрони. Це спеціальні речовини, які дозволяють імпульсам передавати інформацію з одного нейрона на інший або на ефекторні клітини (наприклад, м'язи чи залози).

Коли електричний сигнал досягає кінця аксона (процесу виходу з нейрону), нейромедіатори вивільнюються в синаптичній щілині. Ці речовини потім зв'язуються з рецепторами на дendirитах чи тілі іншого нейрона, спричинюючи передачу сигналу від одного нейрона до іншого.

Деякі звичайні нейромедіатори включають ацетилхолін, серотонін, дофамін та глютамат. Кожен з цих нейромедіаторів має свої унікальні функції та ролі в регулюванні нервової системи. Нейромедіатори важливі для контролю різноманітних фізіологічних та психологічних процесів, таких як настрій, сон, апетит, рух та інші.

Відомо, що пам'ять людини залежить від стану мозку. Різні захворювання, травми, алкоголь можуть спричиняти амнезію.

Людина має систему, яка відповідає за умовні рефлекси та інтуїцію. Ця система необхідна, щоб робити швидкі рішення зазвичай на основі неповної кількості інформації.

Інтуїція може бути розглянута як спосіб розуміння або відчуття ситуації без явного аналізу. Вона базується на попередньому досвіді та несвідомих обробках інформації.

Умовні рефлекси, у свою чергу, — це автоматичні реакції організму на певні стимули, які набулися в результаті попереднього досвіду. Вони дозволяють швидко реагувати на певні ситуації без необхідності обдумувати кожен крок.

Ці механізми є важливими для ефективності та адаптації до оточення, особливо в умовах, коли немає часу на ретельний аналіз інформації. Вони допомагають людині швидко приймати рішення та реагувати на змінні ситуації.

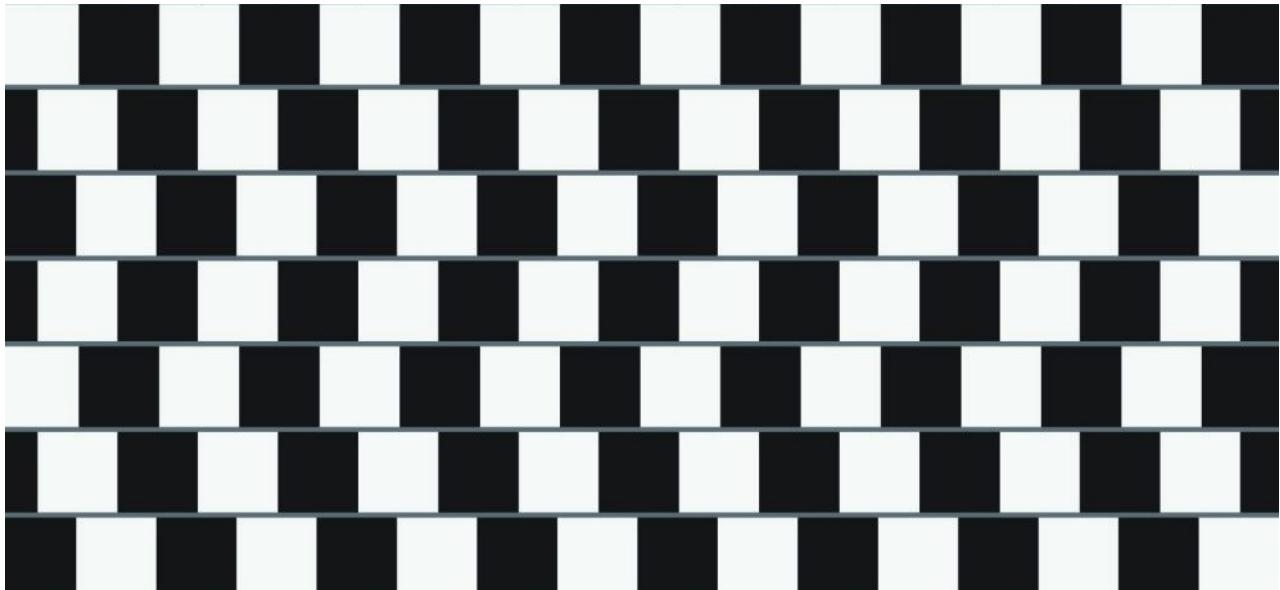
Парадокс Моравека — це концепція в штучному інтелекті та робототехніці, названа на честь Ганса Моравека, дослідника в області робототехніки. Вона висвітлює цікаве спостереження: завдання, які людина вважає легкими, такі як впізнавання облич, ходьба або навіть мовлення, часто є складними для виконання комп'ютерами, тоді як завдання, які людина вважає складними, такі як складні математичні обчислення або абстрактне мислення, можуть бути відносно простими для комп'ютерів. Ганс Моравек запропонував, що цей парадокс виникає через те, що завдання, які складні для комп'ютерів, зазвичай вимагають високорівневого мислення і абстракції, які є недавніми еволюційними досягненнями людини і не є значущою частиною нашої еволюційної історії. З іншого боку, завдання, які легкі для комп'ютерів, такі як базова сенсорна сприйнятливість та рухові навички, є функціями, які люди розвивали протягом мільйонів років і глибоко вкорінені в нашій біології.

Серед вчених, які вивчали мозок були Камиль Гольджі, Сантьяго Рамон-і-Кахаль, Іван Павлов, Отто Леві, Егас Моніз, Роджер Сперрі, Джон Екклс, Конрад Лоренц, Торстен Візел, Пол Лотербур, Деніел Канеман.

Оптичні ілюзії

Теза Декарта про те, що почуття оманливі, цілком обґрунтована. Існують зорові ілюзії (Оптична ілюзія).

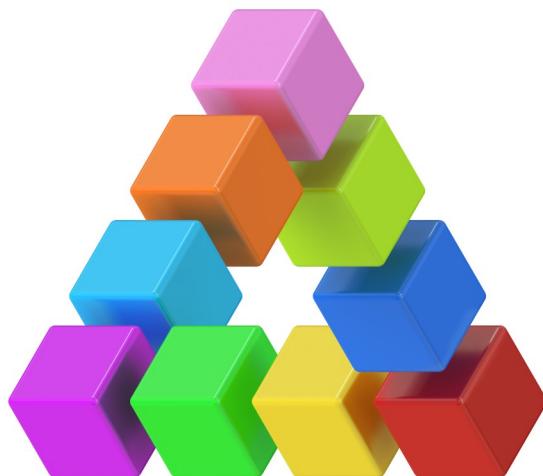
Тут горизонтальні лінії паралельні.



Трикутник Пенроуза — це трикутний об'єкт, оптична ілюзія, що складається з об'єкта, який може бути зображеній на малюнку в перспективі, але не може існувати як об'ємний об'єкт.

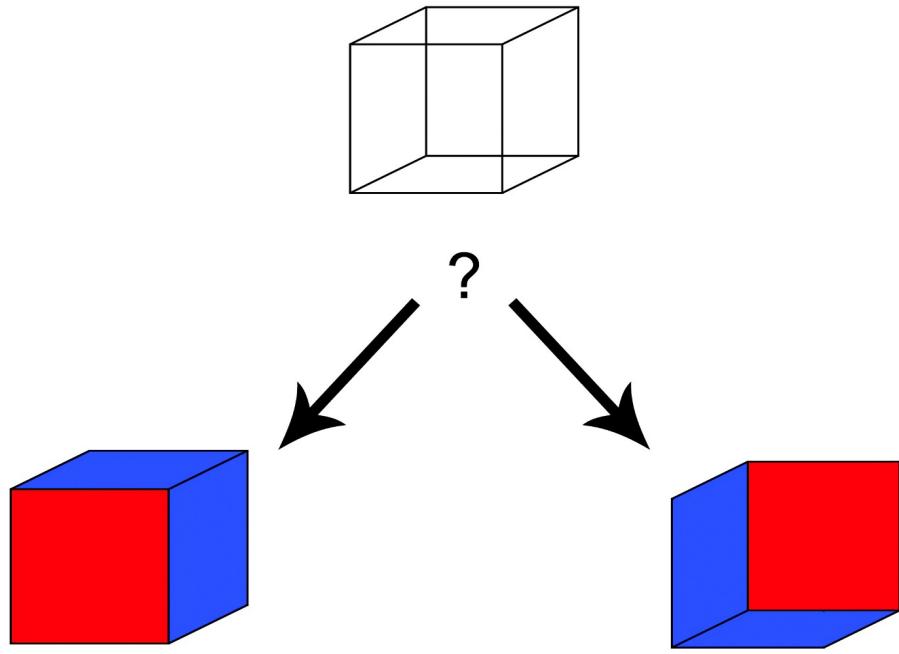


За певних припущенів трикутник Пенроуза є нібіто суперечливою фігурою, але водночас можна побудувати реальну тривимірну фігуру, проекція якої буде вписуватися в трикутник Пенроуза. При цьому реальна тривимірна фігура відрізняється від того, як інтуїція інтерпретує початковий об'єкт. Він був виявлений у 1934 році шведським художником Оскаром Рейтерсвердом, який зобразив його у вигляді набору кубиків. Ця фігура стала широко відомою після публікації в 1958 році в “Британському журналі психології” статті про неможливі фігури англійського психіатра Лайонела Пенроуза та його сина, математика Роджера Пенроуза (згодом лауреата Нобелівської премії з фізики 2020 року). Також в цій статті неможливий трикутник був зображений в найзагальнішому вигляді — у вигляді трьох балок, з'єднаних один з одним під прямим кутом. Під впливом цієї статті в 1961 році голландський художник Мауріц Ешер створив одну зі своїх знаменитих літографій “Водоспад”.



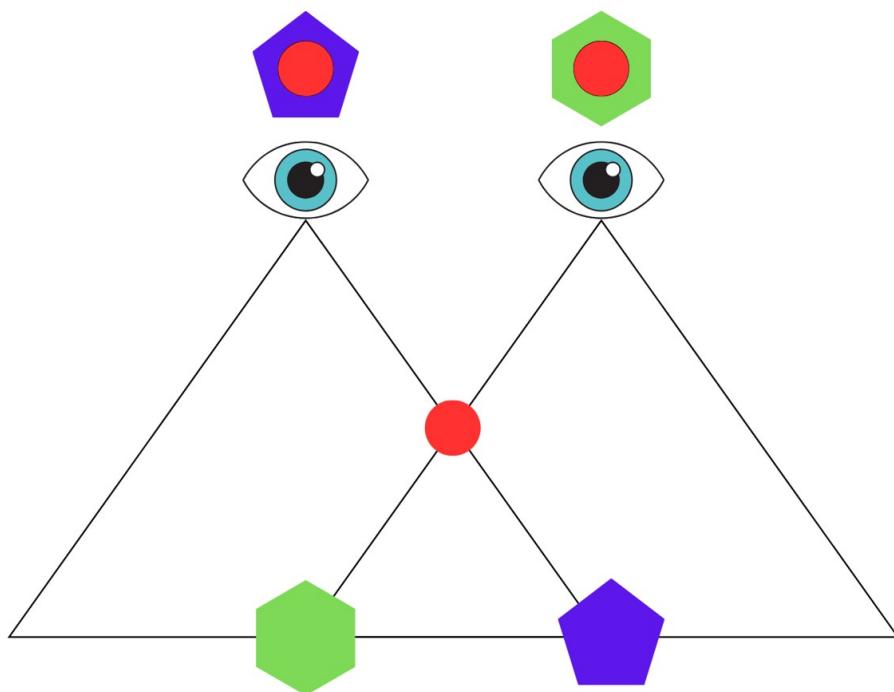
Теорема. Неможливо створити універсальний комп'ютерний алгоритм, який би аналізував двовимірне зображення предмета й видавав дані про орієнтацію цього предмета в просторі. Доказ методом контрприкладу: Куб Неккера є амбівалентним зображенням, яке має декілька інтерпретацій. Тому роботи (машини зі штучним інтелектом) потребують камер які можуть аналізувати тривимірне світлове поле, а не просто двовимірне зображення. Крім того, можна опиратись на стереоскопічний (бінокулярний) зір.

Куб Неккера — це зображення, яке вперше було опубліковано як ромбоїд у 1832 році швейцарським кристалографом Луї Альбертом Неккером. Це просте каркасне креслення куба без візуальних підказок щодо його орієнтації.



Бінокулярний зір — це зір двома очима, при якому в мозку зображення зливається в єдиний образ. Завдяки бінокулярному зору можна визначати відстань до предмета, взаємне розташування предметів.

Паралакс — видиме зміщення або різниця орієнтації об'єкта, що розглядається з двох різних позицій. Що далі розташований об'єкт, то менше змінюється його візуальна позиція. Що більші відстань до об'єкта, або що більша відстань між точками спостереження (база), то більший паралакс.



Закон Вебера — Фехнера

Бел — одиниця вимірювання різниці рівнів звукової гучності або потужності та потужності інших фізичних величин.

Число Бел дорівнює десятковому логарифму відношення інтенсивності I , відповідної вимірюваному рівню гучності, до інтенсивності I_0 довільного рівня, тобто:

$$L = \lg(I / I_0);$$

Таке логарифмічне визначення потрібне тому, що людське вухо реагує на підвищення гучності нелінійно, а за законом Вебера-Фехнера.

Умовно вважають нульовим рівнем гучності той, якому відповідає інтенсивність звуку на один порядок нижча (в десять разів) за поріг чутності. Отже, рівень гучності $L = 1$ Б тоді, коли інтенсивність I в порівнянні з початковою I_0 зростає в 10 разів. Вживають частіше одиницю, меншу за Бел в 10 разів — децибел (дБ). Гучність людської розмови приблизно 6 белів, тобто 60 децибелів.

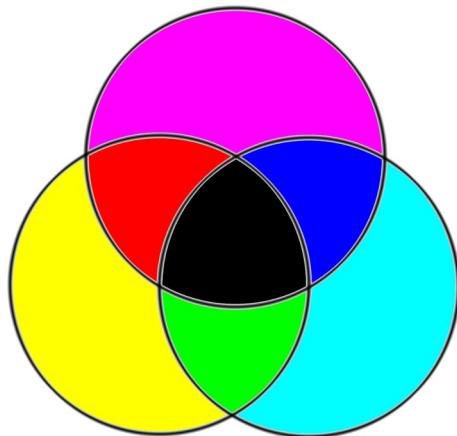
Закон Вебера — Фехнера — психофізіологічний закон, що описує сприйняття різних фізичних величин органами чуттів. Ернст Вебер стверджував, що "мінімальне посилення стимулу, яке приведе до помітного посилення відчуття, пропорційне попередньому стимулу". Наприклад, чим ви багатіші, тим більше має бути прибуток, щоб ви відчули приріст грошей. Коли в тебе один долар й тобі дають ще один, тоді для тебе це значне збагачення, а коли в тебе мільйон доларів і тобі дають один долар, тоді для тебе це невідчутно в фінансовому плані. Коли перед вами натовп народу ви не помітите, якщо одна людина буде додана, але коли перед вами дві людини, то ви однозначно помітите третю, якщо вона приєднається.

Лінійний градієнт

Джеймс Кларк Максвелл (1831 - 1879) піонер кількісної теорії кольору (RGB), автор триколірного принципу кольорової фотографії.

RGB (абревіатура від red - червоний, green - зелений, blue - синій) — це адитивна колірна модель, яка описує спосіб кодування кольору для відтворення кольору з допомогою трьох кольорів, які зазвичай називають основними. Вибір основних кольорів обумовлений фізіологічними особливостями сприйняття кольору сітківкою ока людини. Комбінуючи три промені світла, червоний, зелений і синій, можна відтворити всі кольори, які бачить людське око. Варто зазначити, що маляри мають інший набір так званих базових кольорів, оскільки оперують не світлом, а пігментами, світловідбивачами. Три основні кольори художника — жовтий, червоний і синій.

Print Spectrum

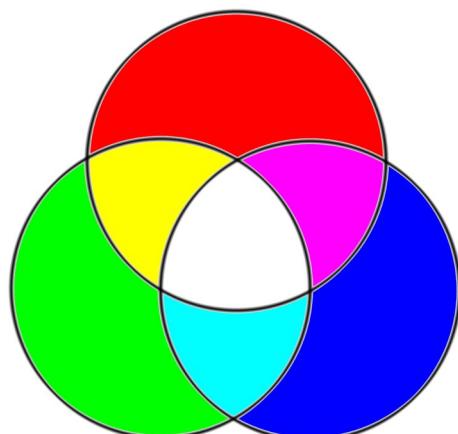


CMYK

Cyan
Magenta
Yellow
Black (k)

RGB

Red
Green
Blue



Light Spectrum

Адитивне змішування кольорів (змішування світлових променів). Основні кольори - червоний, зелений і синій. Модель RGB є адитивною, де кольори додаються до чорного. При відсутності випромінювання — немає кольору — чорний, змішавши всі три в певній пропорції — дає білий. Якщо колір екрану, освітленого кольоровим прожектором, позначено в RGB як (r_1, g_1, b_1) , а колір того самого екрана, освітленого іншим прожектором, є (r_2, g_2, b_2) , то при освітленні двома прожекторами, колір екрану буде позначено як $(r_1 + r_2, g_1 + g_2, b_1 + b_2)$. Зображення в цій колірній моделі складається з трьох каналів. Змішуючи основне випромінювання, наприклад, синій (B) і червоний (R) отримаємо пурпурний (M), зелений (G) і червоний (R) - жовтий (Y), зелений (G) і синій (B) - блакитний (C). При змішуванні всіх трьох основних випромінювань виходить білий колір (W). Джеймс Максвелл запропонував адитивний синтез кольору як спосіб отримання кольорових зображень у 1861 році.

Лінійний градієнт між двома кольорами представляє собою плавний перехід від одного кольору до іншого.

Лінійний градієнт між двома кольорами визначається як перехід від одного кольору до іншого кольору у вигляді плавної зміни кольорів на всьому шляху між ними. Це можна представити як інтерполяцію між початковим і кінцевим кольорами в просторі RGB (червоний, зелений, синій).

Ось математичний приклад лінійного градієнту між двома кольорами у просторі RGB, де ми хочемо плавно перейти від червоного $(255, 0, 0)$ до зеленого $(0, 255, 0)$: Задамо кількість кроків чи точок нашого градієнту. Наприклад, якщо ми хочемо 100 кроків, то $n = 100$.

Розрахунок кольору для кожного кроку (i):

$$R(i) = R_1 + (R_2 - R_1) * i / n;$$

$$G(i) = G_1 + (G_2 - G_1) * i / n;$$

$$B(i) = B_1 + (B_2 - B_1) * i / n;$$

У 1916–1918 рр. німецький фізико-хімік Вільгельм Оствальд (1853 — 1932) опублікував книги “Кольоровий праймер”, “Кольорознавство”, а також “Атлас кольорів”. Ці публікації встановили зв’язки між різними візуальними кольорами. Оствальд представив їх як тривимірне представлення колірного простору, який є топологічним твердим тілом, що складається з двох конусів. Одна вершина конуса чисто біла, а інша чисто чорна. Вісім основних кольорів представлені вздовж сторін двох конусів. У цьому представленні кожен колір є сумішшю білого, чорного та восьми основних кольорів. Таким чином, є три ступені свободи, які представляють кожен колір. Таке представлення кольорів було важливим раннім кроком до їх систематизації, замінивши сприйняття кольору людським оком об’єктивною системою. З часом досягнення Оствальда в галузі кольорознавства стали частиною системи кольорів HSL і HSV.

Розмивання Гауса

Розмивання Гауса — це метод фільтрації зображення за допомогою функції Гауса, який призводить до розмивання зображення. Даний ефект широко використовується в графічних програмах, як правило, для зменшення зашумленості зображені та зниження деталізації.

Алгоритм розмивання Гауса:

1. Картинка у вигляді масиву пікселей.
2. Вираховуємо таблицю ваг (ядро) відповідно до функції розподілу Гауса та радіусу.
3. Кожному пікселю в таблиці присвоюємо значення, яке рівне сумі значень всіх пікселей в його околиці попередньо помножених на вагу з таблиці ваг (ядро), яка поділена на суму всіх ваг з таблиці.

Ось формула двовимірного розподілу Гаусса:

$$f(x, y) = (1 / (2 * \pi * \sigma_x * \sigma_y)) * \exp(-0.5 * ((x - \mu_x)^2 / \sigma_x^2 + (y - \mu_y)^2 / \sigma_y^2));$$

У цій формулі:

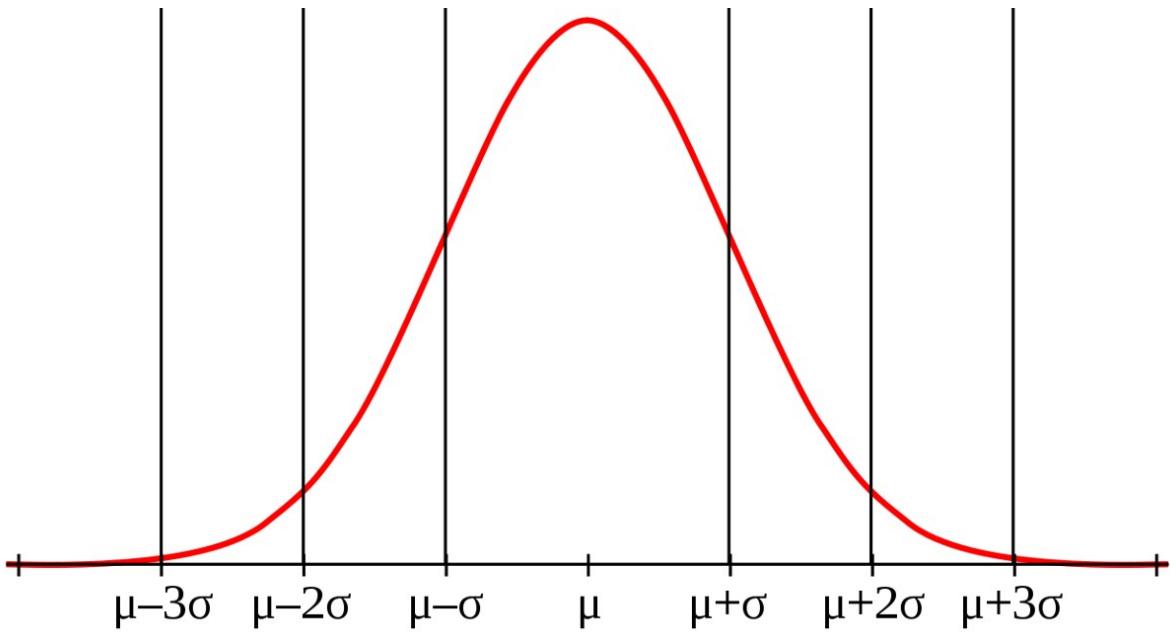
(x, y) - координати точки,
 μ_x, μ_y - середні значення (середні координати),
 σ_x, σ_y - стандартні відхилення,
 π - число пі (приблизно 3.14159).

Ця формула визначає двовимірний гауссівський розподіл, де значення функції залежить від відстані точки (x, y) від середніх значень (μ_x, μ_y) та стандартних відхилень (σ_x, σ_y) . Функція зазвичай використовується для моделювання розподілу інтенсивності або яскравості в графіці та обробці зображень.

Спрощена формула на JavaScript:

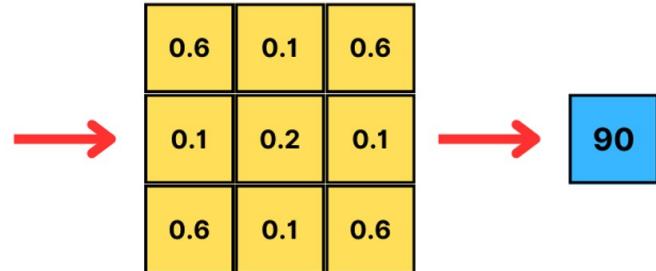
$$(1 / (2 * Math.PI)) * Math.exp(-(Math.pow(x, 2) + Math.pow(y, 2))/2);$$

Розподіл Гаусса (Нормальний розподіл)



Приклад таблиці ваг (не точно по формулі Гауса):

*	*	*	*	*	*
*	10	20	30	*	*
*	20	30	40	*	*
*	30	40	50	*	*
*	*	*	*	*	*
*	*	*	*	*	*



JavaScript код розмивання Гаусса:

```
function gaussianBlur(imageData, radius) {
    const pixels = imageData.data;
    const width = imageData.width;
    const height = imageData.height;

    const kernel = [];
    for (let i = -radius; i <= radius; i++) {
        for (let j = -radius; j <= radius; j++) {
            const distance = Math.sqrt(i * i + j * j);
            const weight = Math.exp(-(distance * distance) / (2 * radius * radius));
            kernel.push({ i, j, weight });
        }
    }

    for (let y = 0; y < height; y++) {
        for (let x = 0; x < width; x++) {
            let r = 0;
            let g = 0;
            let b = 0;
            let weightSum = 0;

            for (const { i, j, weight } of kernel) {
                const neighborX = x + i;
                const neighborY = y + j;

                if (neighborX >= 0 && neighborX < width && neighborY >= 0 && neighborY < height) {
                    const index = (neighborY * width + neighborX) * 4;
                    r += pixels[index] * weight;
                    g += pixels[index + 1] * weight;
                    b += pixels[index + 2] * weight;
                    weightSum += weight;
                }
            }

            const currentIndex = (y * width + x) * 4;
            pixels[currentIndex] = Math.round(r / weightSum);
            pixels[currentIndex + 1] = Math.round(g / weightSum);
            pixels[currentIndex + 2] = Math.round(b / weightSum);
        }
    }
}
```

Множення на ширину (width) у виразі neighborY * width є необхідним, оскільки дані пікселів у масиві зазвичай зберігаються у одновимірному форматі, навіть якщо саме зображення є двовимірним. Кожен рядок пікселів на зображенні відповідає послідовному набору значень у одновимірному масиві.

Множення номеру рядка (neighborY) на ширину зображення дає зміщення до початку цього рядка у одновимірному масиві.

Множення на 4 враховує той факт, що кожен піксель у масиві, як правило, складається з чотирьох значень: одне для каналу червоного кольору, одне для каналу зеленого кольору, одне для каналу синього кольору та одне для каналу альфа (що представляє прозорість). Це відомо як формат RGBA.

Множення зміщення на 4 дозволяє правильно переходити по масиву, рухаючись від одного пікселя до наступного з точки зору цих чотирьох значень.

Простий метод знаходження контурів зображення

У нас є картинка, яка являє собою таблицю пікселів, тобто картинка у вигляді багатовимірного масиву чисел. Числа вказують на значення кольорів пікселя (r,g,b).

Щоб отримати вертикальні контури того, що зображене на картинці, необхідно пройтись по кожному ряду пікселів (зверху вниз картинки) та відняти кольорові значення лівого пікселя від його сусіднього правого пікселя.



Для того, щоб виділити вертикальні контури на зображенні в форматі масиву пікселів, можна використати простий підхід обробки зображень за допомогою віднімання значень кольорів між сусідніми пікселями по горизонталі. Цей процес можна здійснити за допомогою таких кроків:

- Проходження через кожен рядок пікселів у зображенні.
- Для кожного пікселя в рядку (крім останнього) відніміть значення кольору лівого пікселя від його правого сусіда.
- Збережіть абсолютне значення результату, щоб показати величину зміни кольору.

Опціонально, можна застосувати певний поріг, щоб ігнорувати незначні зміни і підсилити видимість важливіших контурів.

Цей підхід дозволяє виділити місця, де існують значні горизонтальні перепади кольорів, що часто відповідає вертикальним лініям чи контурам на зображенні.

Також для подібних цілей можна застосовувати Оператор Собеля.

Що таке нейрона мережа?

Нейромережа являє собою по суті набір функцій, які називаються шарами мережі, кожен шар приймає параметри від попереднього шару (функції). Самі функції шарів складаються з багатьох інших елементарних функцій, які називаються нейронами. Така мережа приймає вхідні параметри й проганяє їх через всі шари. Одні шари фільтрують інформацію, інші аналізують та класифікують, записуючи результат в пам'ять.

Реалізувати певну обробку даних на основі нейромережі в деяких випадках може бути легше ніж на основі класичних алгоритмів. Класичні алгоритми працюють з формалізованими структурами, які буває важко витягнути з набору інформації. Наприклад, щоб зробити класичний алгоритм, який буде визначати по картинці чи на ній зображеній стілець, потрібно по перше виділяти стільці з картинки, а по друге вказати всі види стільців. Замість цього, використовуючи нейромережі, можна надати нейромережі велику кількість картинок зі стільцями, які вона класифікує, а потім зможе знаходити подібні предмети на інших картинках.

Мозок людини побудований на основі нейромереж, що робить його стійкішим і водночас простішим, бо класичний алгоритм може давати зовсім інші результати, якщо якийсь крок в ньому було трішки змінено.

Вектори використовуються в нейромережах для подання даних, їх розподілу та обчислень. Нейрони — базові обчислювальні одиниці нейромереж, які обробляють вхідні дані та передають їх далі.

Лінійна регресія в нейромережах застосовується для прогнозування значень на основі лінійних відносин між вхідними та вихідними даними.

Персепtron — це проста форма штучної нейронної мережі. Він складається з одного чи декількох шарів нейронів, які приймають вхідні дані, обчислюють їх та передають вихідні значення. Це базова модель нейромережі, яка використовується для класифікації та простих рішень у задачах машинного навчання.

Персепtron складається з набору вхідних вузлів, кожен з яких має вагу, яка множиться на вхідні дані. Ці суми зважених вхідних значень потім проходять через функцію активації, наприклад, ступеневу функцію (функція Хевісайда) або сигмоїду, для видачі вихідного сигналу. Після цього вихідні значення можуть передаватися іншим шарам персептрона або вважатися кінцевим результатом для задачі класифікації чи регресії. У залежності від архітектури та функцій активації, персепtron може вирішувати прості завдання, але для складніших завдань потрібні складніші нейронні мережі.

Навчання з учителем — це процес, коли модель машинного навчання вивчає відношення між вхідними даними та їх відповідними вихідними мітками (наприклад, класифікація чи прогнозування) на основі навчального набору даних, де кожен приклад має відомі відповіді.

У навчанні з учителем модель отримує пари вхідних даних разом із відомими правильними відповідями. Наприклад, якщо ми навчаемо модель розпізнавати числа на зображеннях рукописного написання, кожне зображення цифри (вхідні дані) має позначення числа, яке воно представляє (відповідь). Модель навчається визначати закономірності між цими зображеннями та відповідями, щоб у майбутньому здійснювати передбачення для нових, невідомих зображень.

Персептрон Розенблата

“Суть теоретичної моделі полягає в тому, що це система з відомими властивостями, які легко піддаються аналізу, яка, за гіпотезою, втілює істотні риси системи з невідомими або неоднозначними властивостями” (Френк Розенблат, “Персептрони й теорія мозку”)

Персептрон — це мережа передачі, що складається з генераторів сигналів трьох типів: елементів сприйняття (S), асоціативних елементів (A) та реактивних елементів (R). Функції генерації цих елементів залежать від сигналів, які виникають або десь всередині мережі передачі, або, щодо зовнішніх елементів, від сигналів, що надходять від зовнішнього середовища. Фактично, персептрон можна подати у вигляді графа, де вершини — це сенсори одного з трьох типів, а ребра графа — це зв'язки між цими сенсорами.

Сенсорний елемент S - це елемент, який приймає сигнал на вході та надає інший сигнал на виході, згідно з певними правилами, тобто вихідний сигнал елемента S - це функція його вхідного сигналу. Вихідний сигнал елемента S передається внутрішнім елементам мережі, наприклад, до елементів A, R або іншого елемента S.

Асоціативний елемент A - це елемент, схожий на елемент S, але його вихідний сигнал є функцією від (тобто залежить від) кількох входів. Таким чином, елемент A є функцією кількох змінних, а елемент S - функцією однієї змінної. Вихідний сигнал елемента A передається внутрішнім елементам мережі, наприклад, елементу S, R або іншому елементу A.

Реакційний (або реагуючий) елемент R подібний до елемента A, лише він відправляє свій вихідний сигнал до зовнішнього середовища у відношенні до мережі, тим самим не безпосередньо впливаючи на стан мережі.

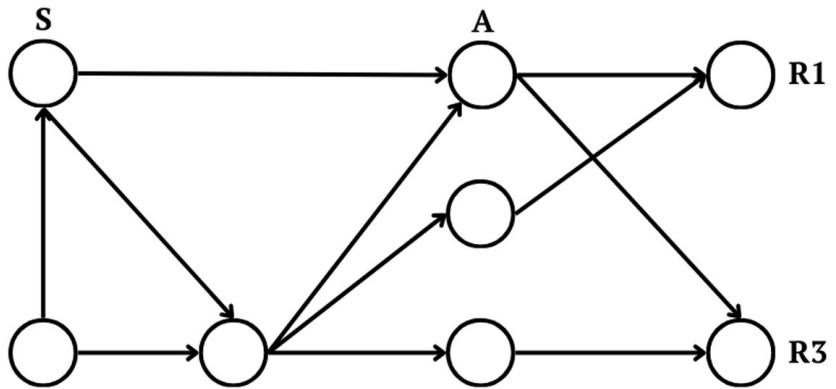
Матриця взаємодії для мережі (матриця ваг), яка складається з елементів типу S, A, R, є матрицею, елементи якої є коефіцієнтами з'єднання для всіх пар елементів (тобто кількість зв'язків між двома елементами типу S, A, R). Якщо немає зв'язку від одного елемента до іншого елемента, то коефіцієнт взаємодії між ними дорівнює нулю.

Активний стан мережі у момент часу t визначається набором сигналів, випущених усіма генераторами сигналів мережі в цей самий момент часу t.

Отже, персептрон — це мережа, яка складається з елементів типу S, A, R, змінною матрицею взаємодії V, що визначається послідовністю минулих станів активності мережі.

Персептрон називається простим, якщо він відповідає 5 вимогам:

1. Він містить лише один R елемент (вихід), до якого під'єднані всі A елементи.
2. Система містить тільки послідовні зв'язки від S елементів до A, від A елементів до R.
3. Матриця взаємодії не змінюється в часі, тобто стала, наперед задана (pre-trained).
4. Всі зв'язки спрацьовують за одинаковий час.
5. Всі елементи (S, A, R) являються функціями від алгебраїчної суми своїх входів.



Система керування підкріпленням персептрону — це будь-яка система (або механізм), зовнішня персептрону, яка служить для зміни матриці взаємодії (матриці ваг) персептрону відповідно до правил обраної системи підкріплення.

Суть роботи персептрана з системою підкріплення полягає в наступному:

- Вплив зовнішнього середовища.
- Реакція персептрану на цей вплив.
- Позитивна або негативна реакція системи керування підкріпленням на реакцію персептрану.
- Якщо відповідь системи керування підкріпленням позитивна, то персепtron посилює матрицю взаємодії, тобто збільшує значення (ваги) на активних елементах. Якщо негативна, то послаблює.

Отже, як кажуть, відбувається "перебудова нейронів". Елемент відіграє роль штучного нейрона.

Персепtron Mark I - перша реалізація моделі персептрана. У 1957 році Лабораторія аeronавтики Корнелла успішно завершила симуляцію роботи персептрана на комп'ютері IBM 704, а через два роки, 23 червня 1960 року, в Корнельському університеті був продемонстрований перший нейрокомп'ютер Mark-1, який міг впізнавати деякі літери англійського алфавіту.

Теорема існування універсальних персептранів

Френк Розенблат довід (в книзі “Персептрани й теорія мозку”), що структура елементарного персептрана дозволяє побудувати систему, яка може давати розв’язання будь-якої задачі класифікації відносно своїх входів. Елементарний персепtron має прихований шар з елементів А.

Але, якщо персепtron має тільки елементи S, які напряму з’єднані з R, тоді він не може вирішувати навіть досить елементарні завдання.

Ми вчимо персепtron подаючи на вхід дані та вказуючи очікуваний результат. Але якщо персепtron складається з вхідних елементів S, які через матрицю ваг з’єднані з вихідним елементом R, тоді ми не зможемо налаштувати матрицю ваг так, щоб виконувалась операція XOR.

Дані (вхідні, вихідні):

{input: [1,1], output: 0},
{input: [0,0], output:0},

{input: [1,0], output:1},
{input: [0,1], output:1},

Матриця ваги повинна бути помножена на вхідні дані та видати коректну відповідь на вихід.

1	0
---	---

*

x
y

= 1;

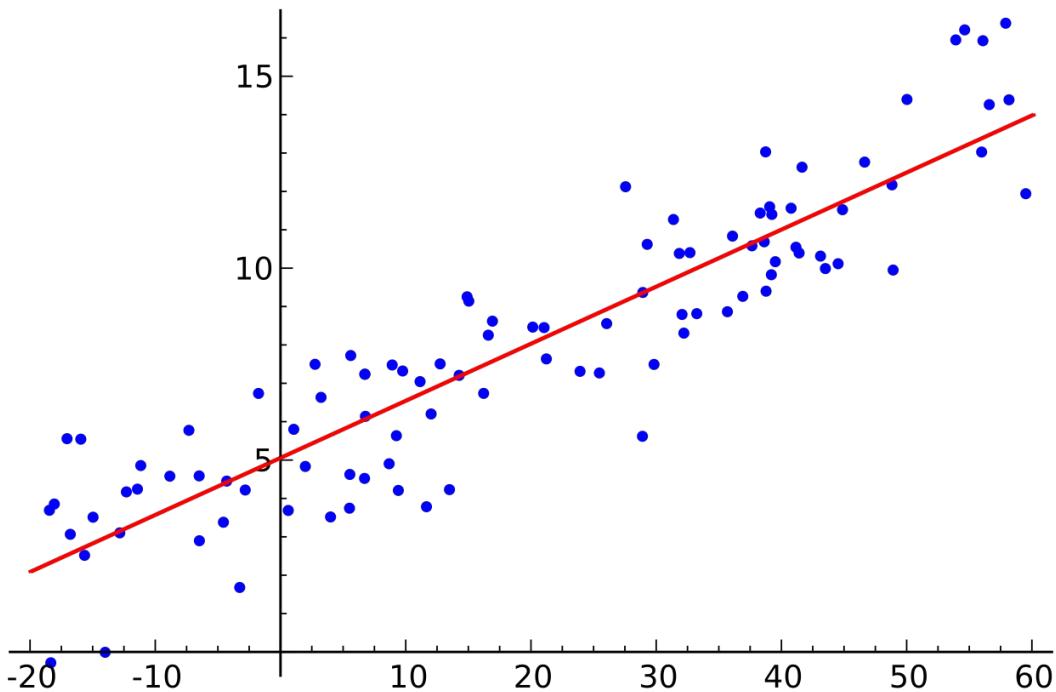
Очевидно, що ви не зможете підібрати x та y в матриці ваг, так, щоб вона реалізувала оператор XOR.

Для імплементації цього потрібно багатошаровий персепtron (нейромережу) з прихованими шарами.

Лінійна регресія

Лінійна регресія — це статистичний метод для моделювання залежності між змінною, яку називають залежною змінною, та однією чи кількома змінними, відомими як незалежні змінні. Мета полягає у прогнозуванні значень залежної змінної на основі значень незалежних змінних.

Приклад простої лінійної регресії з однією незалежною змінною



Логістична регресія відрізняється від лінійної регресії тим, що замість передбачення числового значення відгуку, вона передбачає ймовірність виникнення події, яка має два можливих результати (наприклад, так/ні, успіх/неуспіх). Логістична регресія використовує логістичну функцію (сигмоїду) для класифікування даних.

Логістична регресія використовує логістичну функцію для передбачення ймовірності виникнення події, що дозволяє моделі виражати ймовірність у межах від 0 до 1.

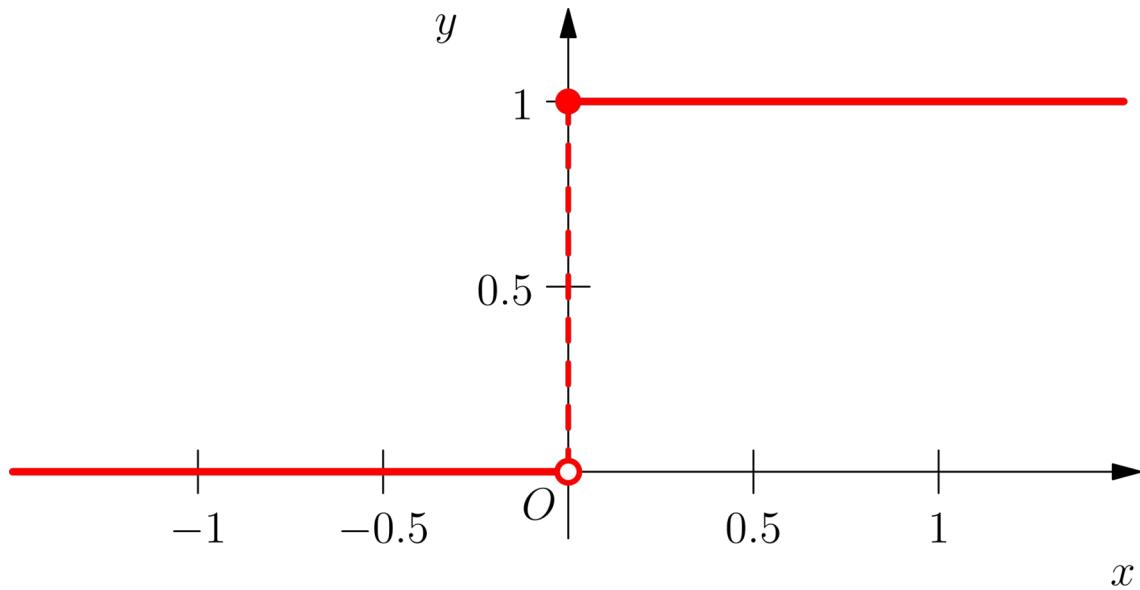
Функція Хевісайда

Для того, щоб функція була диференційованою в певній точці, вона повинна бути неперервною в цій точці й мати ліміт послідовності в цій точці, а крім того буде визначена для всіх значень. Просто кажучи, щоб функція була диференційованою вона повинна бути гладкою (плавною). Саме тому сигмоїда диференціюється і близька до функції Хевісайда, яка має розрив і не диференціюється в нулі, її (сигмоїду) використовують як альтернативу функції Хевісайда.

Функція Хевісайда — це функція дійсної змінної значення якої рівне 0 для від'ємних значень аргументу і рівне 1 для додатних значень аргументу.

Наприклад функція $|x|$ не диференціюється, бо має прямий кут в 0.

Гладка функція або неперервно-диференційовна функція — це функція, що має неперервну похідну на всій області визначення.



Сигмоїда

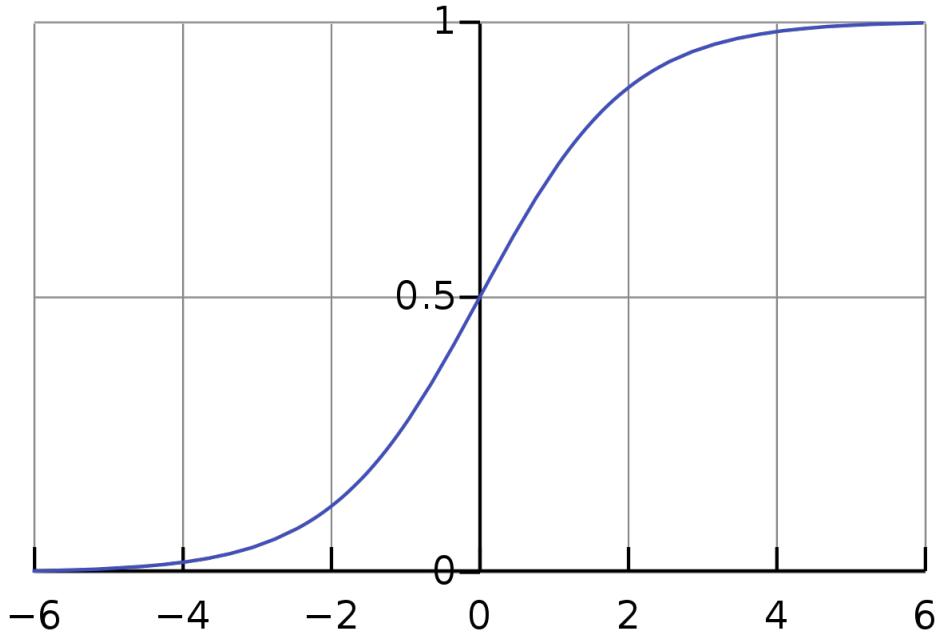
Сигмоїда — це неперервно диференційована монотонна нелінійна S-подібна функція, яка часто застосовується для згладжування значень деякої величини.

Формула сигмоїдної функції виглядає наступним чином:

$$\text{sigma}(x) = 1 / (1 + e^{-x})$$

Тут e - основа натурального логарифма, а x представляє вхідне значення або зважену суму вхідних значень в моделі. Ця функція перетворює будь-яке дійсне число в діапазон між 0 і 1.

Сигмоїдна функція має різноманітні застосування, особливо у сферах, пов'язаних з машинним навчанням і штучним інтелектом.



DBSCAN

Цифрове зображення це таблиця точок (пікселів).

Знак на цифровій картинці – це певний набір точок, який можна виділити із загальної картинки за кольором, або за просторовим розміщенням.

Таким чином, розклавши картинку на шари за кольором, можна шукати знак на цих шарах або їх комбінаціях. Потрібно просто перевірити чи підходить знак через певний еталон. Для розпізнавання знаків на цифрових зображеннях використовують різні алгоритми та їх комбінації, зокрема, алгоритм кластеризації (DBSCAN), алгоритм Джарвіса для обведення, фільтрацію, розмиття (наприклад Розмивання Гауса), Метод Монте-Карло.

Якщо у вас є набір точок в декартової системі координат і ви хочете всі скуччення точок виділити в конкретну групу (кластер), тоді для цього можна використовувати алгоритм DBSCAN.

DBSCAN (density-based spatial clustering of applications with noise) — алгоритм кластеризації даних, який запропонували Мартін Естер, Ганс-Петер Крігель, Йорг Сандер та Сяовей Су в 1996 році.

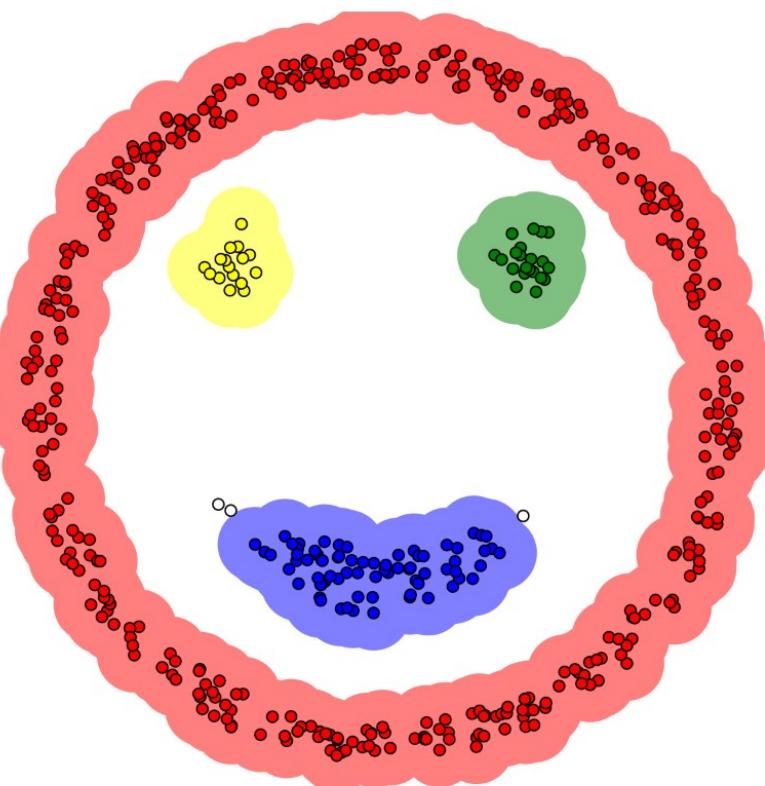
Зверніть увагу DBSCAN це не алгоритм агрегації, а саме алгоритм маркування точок на належність до певної групи. Цей алгоритм корисний для обробки зображень, наприклад, для групування точок на зображеннях, які, ймовірно, складають якийсь символ. Також DBSCAN використовують у галузі машинного навчання, коли машина групует дані й присвоює їм певні категорії. Поодинокі точки, які не входять в жодну групу, алгоритм вважає шумом.

DBSCAN досить трудомісткий алгоритм по часовій складності. DBSCAN має складність у гіршому випадку $O(n^2)$.

Результат DBSCAN трішки залежить від порядку точок в масиві.

Ось кроки алгоритму DBSCAN:

1. Переберіть всі точки й визначте, які з них ключові точки, відповідно до радіусу r та мінімуму сусідів minN . Точка вважається ключовою, тобто осередком, якщо вона має щонайменше minN сусідів в межах радіусу r , тобто сусідів, які не далі ніж довжина r .
2. Переберіть всі ключові точки (осередки) й розподіліть їх по кластерах. Дві ключові точки (осередки) належать одному кластеру, якщо вони перебувають в межах відстані r один від одного.
3. Переберіть всі інші (не ключові) точки $й$, якщо біля них в межах радіуса r є ключова точка, приєднайте їх до кластера ключової точки в межах r . Після приєднання, точка не стає осередком $й$, відповідно, не буде впливати на процес перевірки по додаванню наступних точок. Момент: Якщо біля звичайної точки в межах радіуса r є дві ключові точки, тобто осередку, тоді ця точка приєднається до осередку (точніше кластера), який буде перевірятись першим, тобто має менший порядковий номер в масиві.
4. Поодинокі точки, які не увійшли в жоден кластер при minN не дорівнює 0, будуть вважатися шумом, тобто прибрані з розгляду.



Що таке Перенавчання?

Перетренування, Перенавчання (overfitting) в машинному навчанні — це ситуація, коли модель навчається настільки добре на навчальних даних, що вона "запам'ятує" їх занадто точно. Замість того, щоб виявляти загальні закономірності, модель враховує навіть найдрібніші деталі та шум в навчальних даних. Як результат, коли ця модель застосовується до нових даних, які вона не бачила раніше, її прогнози можуть бути неточними.

Наприклад, якщо модель занадто детально підлаштовується до навчальних даних, враховуючи навіть випадкові відмінності або шум, це може привести до того, що вона не зможе коректно узагальнити свої знання на нові дані. Це, в свою чергу, може привести до поганої продуктивності моделі на реальних або тестових даних.

Перетренування можна уявити як ситуацію, коли вчений занадто детально описує умови досліду, враховуючи навіть випадкові або непотрібні деталі. Це призводить до того, що він не може коректно узагальнити знання на нові, раніше не бачені дані. Це схоже на ситуацію, коли експерт аналізує деталі експерименту настільки докладно, що він може пропустити загальні закономірності або відхилитися від головних тенденцій, фокусуючись на дрібницях.

У статистиці та машинному навчанні ансамблеві методи використовують кілька алгоритмів навчання, щоб отримати кращу прогностичну продуктивність, ніж можна було б отримати від будь-якого зі складових алгоритмів навчання окремо.

Випадковий ліс (Random forest) - це ансамбль моделей дерев рішень (ансамблевий метод), які навчаються на випадкових підвибірках навчальних даних та випадкових підмножинах ознак. Одна з переваг випадкового лісу полягає в тому, що він здатний добре узагальнюватись до нових даних та уникати перетренування (перенавчання, overfitting).

Це досягається завдяки використанню багатьох дерев рішень, кожне з яких навчається на різних підмножинах даних та ознак. Коли потрібно зробити прогноз для нового прикладу, кожне дерево випадкового лісу робить свій прогноз, і результат обчислюється шляхом голосування або середнього значення прогнозів кожного дерева.

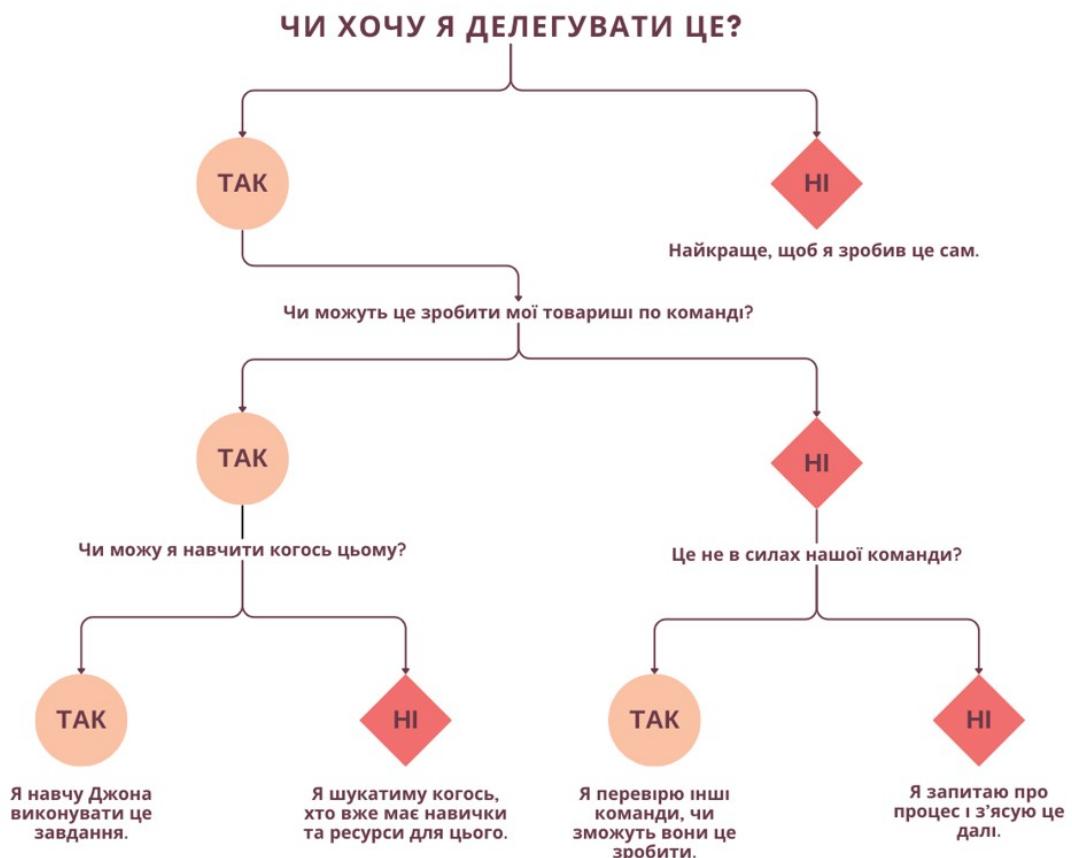
Таким чином, випадковий ліс може бути ефективним методом уникнення перетренування, оскільки він використовує ансамбль моделей та різноманітність даних для зменшення ймовірності "запам'ятування" навчальних даних.

Випадковий ліс, з одного боку, може допомогти уникнути перетренування, оскільки він використовує ансамбль моделей, які навчаються на різних підмножинах даних та ознак. З іншого боку, він може допомогти виявити важливі закономірності в даних, оскільки кожне дерево рішень аналізує дані з різних перспектив. Такий підхід зменшує ризик того, що модель буде надто віддана деталям навчальних даних і не зможе правильно узагальнити свої знання на нові дані.

Почати з найзагальніших та найбільш очевидних закономірностей в машинному навчанні може бути важливою стратегією для уникнення перенавчання. Розуміння основних принципів та патернів в даних дозволяє побудувати більш стійку та універсальну модель, яка краще узагальнюється на нові дані. Починаючи з простих моделей, що відображають загальні закономірності, ми створюємо міцний фундамент для подальшого розвитку, зменшуючи ризик перенавчання на незначних деталях тренувальних даних. Такий підхід допомагає побудувати більш ефективні та загальні моделі, які можуть успішно застосовуватися до різних завдань в машинному навчанні.

Дерево рішень — це графічна модель прийняття рішень, що відображає послідовність рішень та їх наслідки. Воно являє собою деревоподібну структуру, де кожен вузол представляє рішення, а кожне ребро — наслідок цього рішення. Дерева рішень широко використовуються в області машинного навчання як модель класифікації та регресії.

ДЕРЕВО УХВАЛЕННЯ РІШЕНЬ



Що таке згорткова (конволюційна) нейронна мережа?

Згорткова (конволюційна) нейронна мережа (CNN) — це тип штучної нейронної мережі, яка зазвичай використовується для обробки та аналізу великих обсягів даних, таких як зображення. CNN була спеціально розроблена для виявлення та вивчення ієрархічних ознак у вхідних даних за допомогою операції згортки.

(LeNet — згорткова нейронна мережа, запропонована Яном ЛеКуном зі співавторами у 1989 році.)

Зазвичай, з кожним шаром згорткової мережі використовуються операції згортки та підбору (pooling), які призводять до зменшення просторових розмірів даних.

Операція згортки застосовує фільтри для виявлення локальних ознак у вихідних даних, а операція підбору зменшує розмірність, виділяючи ключові ознаки. Цей процес дозволяє мережі зосередитися на більш важливих аспектах даних та робить модель менш чутливою до варіацій у вихідних даних.

Основні компоненти CNN:

1. Шари згортки (convolutional layers): Ці шари використовують фільтри (ядра) для виявлення різних ознак у вхідних даних. Згортка дозволяє мережі автоматично вивчати просторові ієрархічні ознаки, такі як краї, текстири та форми.
2. Шари підбору (pooling layers): Ці шари використовуються для зменшення розмірності зображення та виділення ключових ознак. Зазвичай використовується операція максимального підбору (max pooling), яка обирає максимальне значення з певного регіону.
3. Повнозв'язані шари (fully connected layers): Після кількох шарів згортки та підбору, зазвичай використовуються повнозв'язані шари для класифікації чи регресії вихідних даних.

CNN ефективно використовується для завдань, пов'язаних з обробкою зображень, таких як розпізнавання облич, класифікація об'єктів, сегментація зображень та багато інших. Вона також може застосовуватися до інших типів даних, таких як аудіо чи відео, де важлива просторова структура ознак.

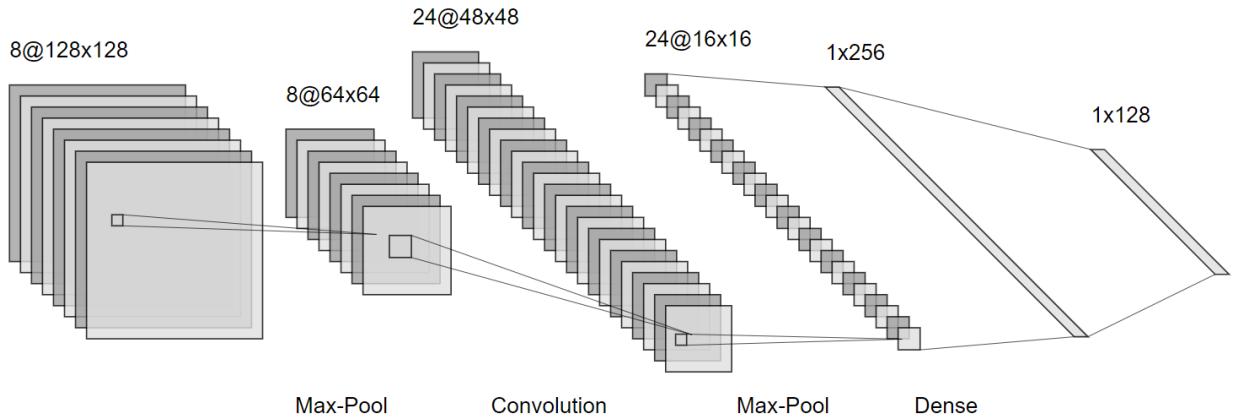
Операція згортання (convolution) є ключовою складовою згорткових нейронних мереж (CNN). Ця операція використовується для виявлення локальних шаблонів чи ознак у вихідних даних. Зазвичай вона застосовується до вхідного зображення чи іншого виду даних, наприклад, аудіосигналу.

Операція згортання включає в себе ядро (фільтр), яке переміщується по вхідних даних. Для кожного положення ядра вхідні значення помножуються на відповідні ваги ядра, і результати додаються, щоб отримати значення вихідного шару. Цей процес повторюється для всіх положень ядра, що веде до створення карти ознак.

Головна ідея за операцією згортання полягає в тому, щоб модель можна було вчити розпізнавати локальні шаблони чи ознаки, такі як краї, текстири або форми, використовуючи невеликі фільтри, які переміщаються по всьому вхідному зображеню. Це дозволяє мережі автоматично визначати та вивчати важливі ознаки вхідних даних.

Операція пулінгу (pooling) є важливою частиною згорткових нейронних мереж (CNN). Ця операція використовується для зменшення розмірності вихідних даних з попереднього шару, зберігаючи при цьому ключові інформаційні ознаки. Типова операція пулінгу - це максимальний пулінг (max pooling) або середній пулінг (average pooling).

На зображені показані шари згорткової нейромережі. Об'ємні шари вказують на збільшення вектора даних (Кожен піксель може бути представлений вектором (наприклад, RGB)). Якщо ви використовуєте п'ять компонент для представлення пікселя (x, y, r, g, b), де (x, y) - це координати пікселя, а (r, g, b) - компоненти кольору (червоний, зелений, синій), то ви отримуєте вектор розміром 5 для представлення кожного пікселя.

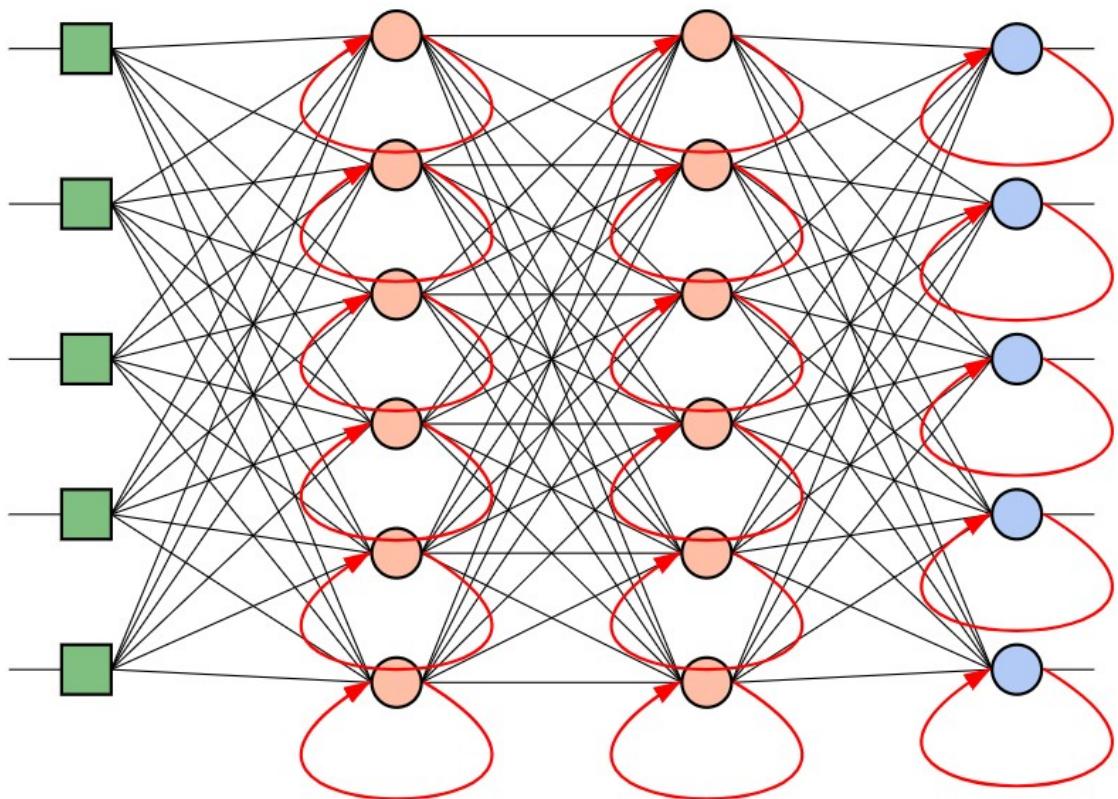


Головна різниця між глибокими нейронними мережами (DNN) і згортковими нейронними мережами (CNN) полягає в обробці даних. DNN ефективні для завдань, де важлива взаємодія між різними частинами даних, тоді як CNN частіше використовуються для обробки зображень чи відео, враховуючи просторову структуру.

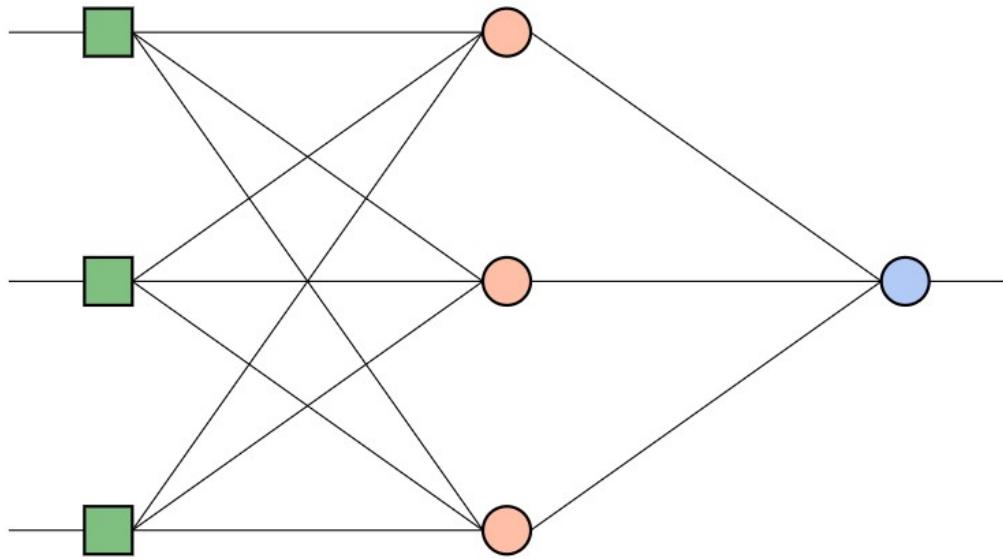
Глибока нейронна мережа (DNN) — це тип штучної нейронної мережі, яка складається з багатьох шарів нейронів, де кожен нейрон взаємодіє з нейронами на наступному шарі. Вони використовуються для розв'язання завдань машинного навчання, таких як класифікація, регресія або генерація даних.

Рекурентні нейронні мережі (RNN) є підтипом глибоких нейронних мереж (DNN). Основна відмінність полягає в їхній здатності обробляти послідовні дані, зберігаючи певний стан або пам'ять про попередні входні елементи (тобто враховувати послідовність отримання даних на вході). RNN використовуються для завдань, де важлива часова залежність, така як обробка мови тощо. Таким чином, RNN можна розглядати як один із варіантів DNN, спеціалізований на роботі з послідовними даними.

RNN



DNN (FNN)



Feedforward neural network (FNN) — це простий тип глибоких нейронних мереж, в якому інформація рухається вперед, без зворотного зв'язку. Це базовий архітектурний принцип багатьох глибоких нейронних мереж, включаючи DNN. У feedforward мережах інформація проходить через шари в одному напрямку, від входних до вихідних шарів, без збереження стану або зворотного зв'язку між шарами. Таким чином, DNN є формою feedforward мережі, а RNN додає можливість роботи з послідовностями за рахунок зворотного зв'язку.

Навчання з вчителем (supervised learning) — це метод машинного навчання, при якому модель навчається на основі пар вхідних-виходних даних (навчальний набір), де вихідні дані (мітки або цільові значення) вже відомі. Модель намагається знайти зв'язок між вхідними та вихідними даними, щоб у майбутньому здати приймати правильні вихідні прогнози на нових, раніше не бачених даних. Цей підхід широко використовується для задач, таких як класифікація та регресія.

У навчанні без вчителя модель працює з невідомими мітками, намагаючись знайти природні структури чи закономірності в даних.

Функція активації в нейромережах визначає виходи нейронів певного шару. Сигмоїда — це одна з таких функцій активації, яка перетворює вхідні значення в діапазон між 0 і 1. Це корисно для задач бінарної класифікації, де ці значення можна трактувати як ймовірності.

Сигмоїда (sigmoid) і функція Хевісайда (Heaviside step function), можуть використовуватися в нейромережах, але зазвичай сигмоїда використовується в задачах бінарної класифікації через свою гладку форму та те, що вона генерує значення в діапазоні між 0 і 1, які можна трактувати як ймовірності. Функція Хевісайда дає значення 0 або 1, що може бути корисним для деяких інших завдань, але вона не гладка, що може ускладнити процес оптимізації нейромережі під час навчання.

ReLU (Rectified Linear Unit) — це тип функції активації, який широко використовується в штучних нейронних мережах. Вона визначається як $\max(0, x)$, де x - вхідний сигнал. Функція ReLU нулює від'ємні значення та лише передає позитивні значення без змін. Це допомагає вирішити проблему зникнення градієнта в глибоких нейронних мережах.

Які принципи розробки продукту?

Проектування продукту — це складний і багатоетапний процес, що включає в себе багато аспектів, від архітектурного дизайну до розробки та впровадження. Розглянемо кілька важливих принципів та інструментів, які можна використовувати під час проектування продукту:

DRY (Don't Repeat Yourself): Цей принцип закликає уникати дублювання коду в програмному продукті. Це допомагає зменшити витрати на обслуговування та полегшує зміни в програмі.

Принцип DRY був сформульований Енді Хантом і Дейвом Томасом у їхній книзі “Прагматичний програміст”.

KISS (Keep It Simple, Stupid): Простота — це ключ до успіху. Дизайн продукту має бути максимально простим і зрозумілим для розробників та користувачів.

YAGNI (You Ain't Gonna Need It): Не додавайте зайвий функціонал або складність до продукту, якщо ви не впевнені, що це потрібно. Це допомагає уникнути зайвого збільшення обсягу роботи.

Separation of Concerns: Розділення обов'язків — це принцип, який рекомендує розділити систему на окремі компоненти або модулі, кожен з яких відповідає за свою конкретну функцію. Зокрема, **MVC (Model-View-Controller):** Це архітектурний шаблон, який розділяє програму на модель (дані), представлення (інтерфейс користувача) та контролер (логіка додатку).

Перед тим, як робити рефакторинг, покрайте код автоматичними тестами.

Рефакторинг змінює програми невеликими кроками, тому, якщо ви припуститеся помилки, легко знайти, де помилка, бо автоматичні тести її виявлять.

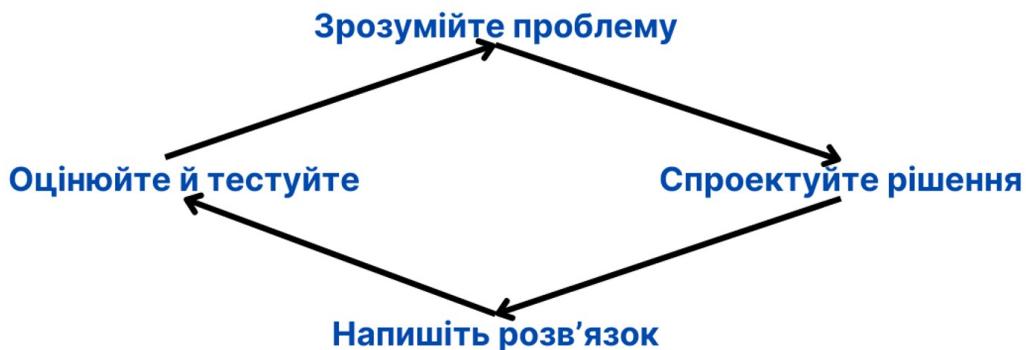
Які принципи декомпозиції задачі?

1. Висхідний підхід (bottom-up approach) — це метод аналізу, який починається з окремих компонентів або найдрібніших деталей і розвивається в загальну картину (систему). У розробці програмного забезпечення висхідний підхід передбачає спочатку створення менших модулів або компонентів, а потім їх інтеграцію в більші системи (структур). Практична відмінність в тому, що в цьому підході розробник одразу починає ділити все на компоненти й потім складати з них більшу систему. Цей підхід сумісний з TDD (Розробка через тестування) та модульним тестуванням, але в ньому більше ризику порушити принцип "YAGNI" та не помітити "за деревами ліс".
2. Низхідний підхід (top-down approach) – це метод розв'язання проблеми або метод аналізу, який починається з загального контексту (системи), а потім розбиває його на менші деталі (компоненти). У розробці програмного забезпечення низхідний підхід включає спочатку визначення високорівневої структури або архітектури, а потім її розкладання на менші модулі (компоненти). Метод сумісний з принципом "YAGNI", але підвищує шанс порушити принцип "DRY" та ускладнити модульне тестування. На відміну від висхідного підходу, тут увага зосереджена на першому впровадженні модулів вищого рівня, часто починаючи з тих, які є критичними для функціональності системи.
3. "Розділяй та володарюй" — це техніка розв'язання задач, яка передбачає розбиття складної задачі на менші, легші задачі, розв'язаннякої задачі окремо, а потім комбінування рішень для розв'язання початкової задачі. Цей підхід зазвичай використовується в розробці алгоритмів та інформації, але він має застосування в різних областях.
4. "YAGNI" означає You Ain't Gonna Need It ("Вам це не знадобиться"). YAGNI — це принцип у розробці програмного забезпечення та гнучких методологіях, який радить не додавати функціональні можливості до системи, поки це не буде необхідно. По суті, це заохочує розробників уникати впровадження функцій або можливостей на основі спекулятивних майбутніх вимог або передбачуваних майбутніх потреб.
5. "DRY" означає Don't Repeat Yourself ("Не повторюйся"). DRY — це принцип розробки програмного забезпечення, спрямований на зменшення повторення коду в програмній системі. Принцип DRY передбачає, що кожна частина знання або логіка в системі повинна мати єдине, однозначне представлення. Іншими словами, слід уникати дублювання коду, коли це можливо. Порушення принципу DRY може породити антипатерн "хіургія дробовиком" (Мартін Фаулер).

Хіургія дробовиком — антипатерн, коли кожного разу, коли ви вносите зміни, вам доводиться вносити багато дрібних редагувань у багато різних класів. Коли зміни повсюди, їх важко знайти, і легко пропустити важливу зміну.

Як проєктувати продукт?

Проєктування продукту — це комплексний процес, який можна розділити на кілька ключових кроків.



Ось як ви можете проєктувати продукт, використовуючи абстракцію, дослідження та порівняння:

1. Abstraction (Абстрагування для моделювання):

- Визначте основні цілі та завдання продукту. Розгляньте, як він вирішуватиме конкретні проблеми або задоволити потреби користувачів.
- Створіть високорівневий огляд архітектури продукту. Розгляньте, які компоненти будуть входити в систему та як вони взаємодіятимуть між собою.
- Визначте ключові властивості продукту, такі як безпека, продуктивність, масштабованість тощо.

2. Research (Дослідження технологій):

- Дослідіть технології, які можуть бути використані для створення вашого продукту. Оберіть технологічний стек, який найкраще підходить для ваших потреб.
- Порівняйте різні підходи та можливі архітектури для вашого продукту. Вибір правильної архітектури може суттєво вплинути на якість та ефективність продукту.
- Порівняйте витрати та ризики різних варіантів реалізації продукту.

3. Comparison (Порівняння з конкурентами):

- Вивчіть конкурентів та їхні продукти. Визначте, що робить їхні продукти успішними або невдалими.
- Проведіть дослідження ринку та аудиторії. Дізнайтесь про потреби та проблеми вашої цільової аудиторії.

Створення прототипу або моделі — це важлива частина процесу проєктування продукту. Прототипи та моделі допомагають команді розробників та дизайнерів краще розуміти, як буде виглядати та працювати продукт перед тим, як розпочати фактичний розвиток.

Ці кроки є лише початком процесу проєктування продукту. Після їх виконання вам слід перейти до детального проєктування, розробки, тестування та впровадження продукту. Також важливо пам'ятати, що проєктування продукту — це ітеративний процес, і ви можете вносити зміни на кожному етапі розробки, враховуючи отримані відгуки та змінюючи плани відповідно до потреб та змін в ситуації на ринку.

"Розділяй та володарюй" (divide and conquer) в інформатиці — важлива парадигма розробки алгоритмів, що полягає в рекурсивному розбитті розв'язуваної задачі на дві або більше підзадачі того ж типу, але меншого розміру, і комбінуванні їх розв'язків для отримання відповіді до вихідного завдання. Розбиття виконуються доти, поки всі підзавдання не стануть елементарними.

У 1637 році Рене Декарт опублікував свою працю "Міркування про метод", в якій описав основні правила розв'язання наукових питань, основними з яких є:

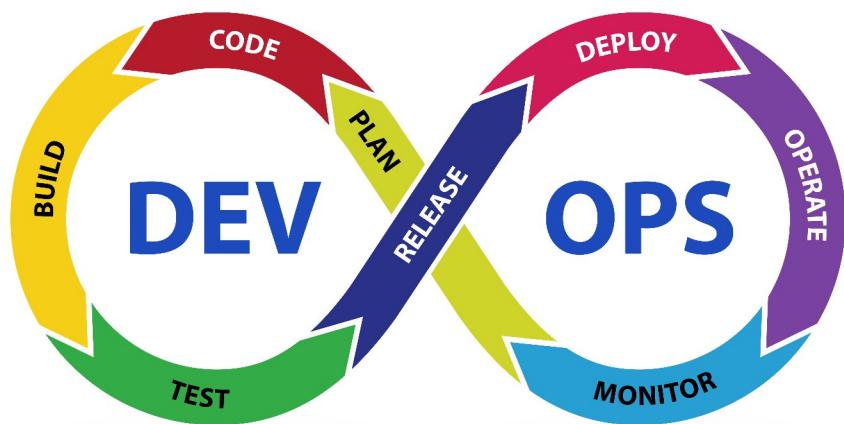
1. Розділіть великі завдання на підзадачі.

Подібне правило в програмістів називається "розділяй та володарюй", і воно означає розділення алгоритму (набору дій) на більш дрібні алгоритми, які разом дають повноцінний результат.

2. Не сприймайте нічого за дійсне, якщо це неочевидно, тобто без достатніх знань (достатньої основи).

Професійний програміст повинен знати, що в технології основне, а що другорядне, похідне. Це робиться з допомогою техніки під назвою деструктуризація (розділення на складові). Сутність, яка не піддається деструктуризації є базовою.

Щоб забезпечити ефективну розробку та підтримку великого програмного продукту, важливо розділити його на модулі та процедури, використовуючи такі концепції як мікросервісна архітектура, модульна архітектура, separation of concerns і ООП. Це подібно до того, як у великих приладах окремі блоки можуть бути виготовлені в різних країнах та різними командами, але потім об'єднані в одне ціле.



Про які тести має піклуватися саме розробник?

“Все, що може піти не так, піде не так” (Закон Мерфі)

Програмний продукт = код + тести + документація (+ ліцензія + підтримка).

Програмний продукт, в середньому, в три рази дорожчий ніж просто debugged code, — писав Фред Брукс.

Тест — один або кілька тестових випадків (test case). Тестовий випадок — це набір **передумов**, введених даних, **дій**, очікуваних **результатів**, розроблених на основі **умов тестування**. Тестування — процес виконання тестів.

У процесі тестування програмного забезпечення тестуються як функціональні, так і нефункціональні вимоги. Функціональні вимоги визначають, що система повинна робити, тоді як нефункціональні вимоги визначають якість або характеристики, які повинні бути притаманні системі, такі як продуктивність, надійність, безпека тощо.

Програміст обов'язково має виконувати тільки один вид тестів — мануальні End-to-end тести. Але розробник також має перевіряти чи немає витоку пам'яті (memory leak) й подібного. Все інше залежно від ситуації та проекту. Але загалом програміст має писати автоматичні модульні тести (Модульне тестування, Unit testing), інтеграційні тести та мануальні тести, або автоматизовані End-to-end тести. Стрес-тести необхідні для додатків, які повинні витримувати високі навантаження (багато даних та запитів).

End-to-end тести (E2E) — це метод тестування програмного забезпечення, коли весь процес від початку до кінця тестується як єдиний блок, включаючи всі інтерактивні компоненти та системи. Це означає, що при проведенні таких тестів використовуються реальні дані, середовища та умови, які відтворюють реальне використання програмного забезпечення.

Зазвичай end-to-end тести охоплюють весь життєвий цикл програмного забезпечення, від введення даних або запиту користувача до отримання очікуваного результату або виведення інформації для користувача. Ці тести дозволяють переконатися, що всі компоненти програми взаємодіють правильно і що програма працює як очікується в реальних умовах.

Потрібно запам'ятати, що які б тести розробник не зробив, він ніколи не повинен тестувати свою програму самостійно, — каже відома книга "Мистецтво тестування програм", автор Гленфорд Майерс.

Давайте свою програму тестувати іншим.

Є дуже специфічні випадки, коли ви можете зробити тест самостійно, наприклад, перебрати всі значення певної функції, її бути на 100% впевнені, що все працює. Але краще звертатися за свіжим поглядом до інших.

Відомий принцип тестування каже: Не плануйте тестування, виходячи з припущення, що помилок не буде виявлено.

Але це не означає, що ви маєте робити припущення про те, що програміст не старанно працює.

Завжди вважайте, що розробник свідомо не винен у виникненні помилок, інакше будуть конфлікти, тобто застосуйте презумпцію невинуватості.

Спочатку виконуйте "тестування методом чорної скриньки", потім "тестування методом білої скриньки" (white-box testing).

Тестування "методом білої скриньки", також відоме як "тестування на основі коду", — це метод тестування програмного забезпечення, при якому тестувальник має доступ до внутрішньої структури

та коду програми. Під час цього виду тестування тестувальник використовує знання про внутрішні процеси програми для створення тестових випадків та випробування різних шляхів виконання програми.

Якщо для тестування поля вводу потрібно перебрати тисячі комбінацій, тоді згрупуйте ці комбінації та робіть тести тільки для однієї комбінації з групи (класи еквівалентності). Наприклад, група чисел, літер, мікс, пусте значення, спецсимволи.

Пишіть негативні й позитивні тести. Наприклад, що повинно вводитись, та що не повинно. Бо, може бути таке, що у ваш input можна вводити все, тому всі позитивні тести виконаються.

Коли потрібно писати автотести:

1. Компонент використовується в багатьох місцях.
2. Код потребує частого ретесту, регресійного тестування.
3. Компонент важко тестувати під час розробки. Допоможе TDD.

Якщо під час кожної зміни коду input компонента вам потрібно робити 30 тестів вручну, то краще спочатку написати тести, а потім розробляти компонент.

Коли не потрібно писати автотести:

1. Швидке прототипування, доказ гіпотези.
2. Маленький і простий (одноразовий) проект, лендінг пейдж.

Сприймайте модуль (юніт) як частину коду, яка може використовуватися незалежно. Якщо частина коду (юніт) буде використовуватися незалежно - пишіть для неї юніт тест.

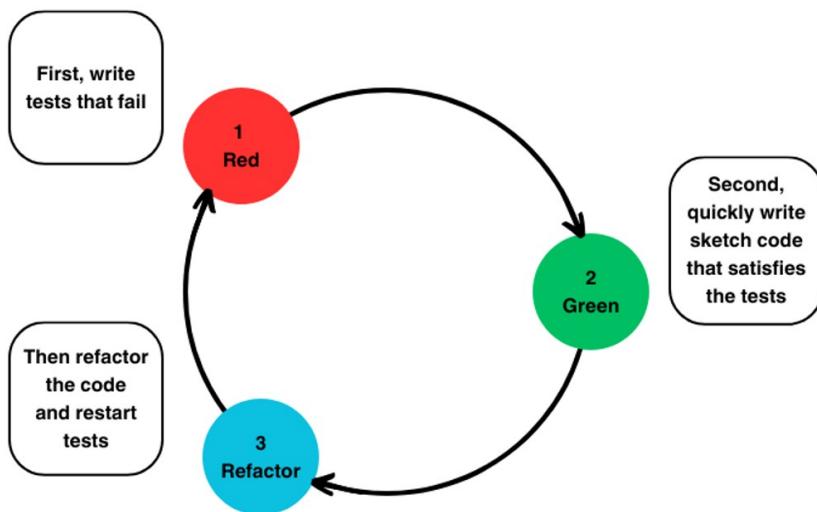
Кент Бек у своїй відомій книзі про TDD (Керована тестами розробка) описує TDD мантуру.

TDD (Test-driven development) мантра

Червоний/зелений/ рефакторинг — мантра TDD.

1. Червоний — Напишіть невеликий тест, який не працює, і, можливо, спочатку навіть не компілюється.
2. Зелений — Змусити тест працювати швидко, не витрачаючи час на доведення коду до ідеалу.
3. Рефакторинг — Усуњте всі дублювання, створені просто для того, щоб змусити тест працювати.

TDD



Три закони TDD

Перший закон: ви не можете писати робочий код, доки не напишете невдалий модульний тест (юніт тест).

Другий закон: ви не можете написати більше модульного тесту, ніж достатньо для невдачі, а відсутність компіляції означає невдачу.

Третій закон: ви не можете писати більше робочого коду, ніж достатньо для проходження поточного невдалого тесту.

TDD не тільки допомагає писати простий код, який постійно перевіряється тестами, а ще й спонукає до ретельної декомпозиції завдань, модулів та інших програмних структур.

TDD (Test-Driven Development) спонукає розробників до ретельної декомпозиції завдань і модулів на більш дрібні частини, що дозволяє створювати більш зрозумілий, гнучкий і підтримуваний код.

Юніт тест тестує поведінку. Якщо ви додали приватний метод в клас, вам в загальному не потрібно його тестувати, бо його включають публічні методи, для яких вже є юніт тести.

Роберт Марктін в книзі “Чистий код” зауважує: "Краще ніяких тестів, ніж брудні тести".

Тести не повинні дублюватися.

Принцип "ви не можете писати більше робочого коду, ніж достатньо для проходження поточного невдалого тесту" означає, що ви пишете саме стільки коду, скільки необхідно, щоб проходити тест, і не більше. Це допомагає уникнути надмірної складності та забезпечити, що ваш код відповідає саме тим вимогам, які ви встановили у ваших тестах.

Якщо ми використовуємо тестування "введення/виведення" разом із тестуванням "чорної скриньки" чи "білої скриньки", ми можемо повністю впевнитися, що функція (чи вхід) працює на 100% правильно, якщо ми маємо "повну індукцію", тобто ми протестували всі можливі комбінації входів.

Однак досягти цього зазвичай неможливо, оскільки в деяких відносно простих сценаріях кількість комбінацій може бути на зразок 10^{14} комбінацій.

Таким чином, вичерпне тестування неможливе у цій ситуації, але все ж можна написати розумні тести, використовуючи класи еквівалентності.

Класи еквівалентності в тестуванні передбачають групування наборів значень входу, які повинні бути оброблені системою аналогічно. Ідея полягає в тестуванні одного представницького значення з кожного класу еквівалентності, щоб забезпечити, що система поводиться однаково для всіх значень в межах цього класу.

Розгляньте систему входу, де користувачі вказують ім'я користувача та пароль. Класи еквівалентності для імені користувача можуть включати дійсні імена користувачів, порожні імена користувачів та імена користувачів зі спеціальними символами.

Приклади тестових випадків:

- Дійсне ім'я користувача та дійсний пароль,
- Порожнє ім'я користувача та дійсний пароль,
- Дійсне ім'я користувача та порожній пароль,
- Порожнє ім'я користувача та порожній пароль,
- Ім'я користувача зі спеціальними символами та дійсний пароль,
- Дійсне ім'я користувача та пароль зі спеціальними символами.

Юніт тест (модульний тест) перевіряє окремий юніт (наприклад, функцію, об'єкт, клас, React компонент). Ви можете писати юніт тести для юніта (модуля), навіть якщо він великий і містить багато інших модулів. Щоб ефективно протестувати великий модуль, ви можете розбити його на менші, більш керовані модулі.

Необхідні ознаки юніт тесту:

1. Виконує перевірку одного юніта.
2. Локалізує проблему в межах юніта.
3. Поведінка юніта, а отже коректність юніт тесту, не залежить від сторонніх сервісів й налаштувань системи (таких як дата, база даних, сторонні сайти тощо).
4. Юніт тест не тестує побічні ефекти за межами юніта, які викликає юніт.

“Модульне тестування (або юніт тестування) — це процес тестування окремих підпрограм, класів або процедур у програмі”

(“Мистецтво тестування програмного забезпечення” Гленфорда Майєрса)

Приклад автоматичного юніт тесту:

```
// Функція для додавання двох чисел
function add(a, b) {
    return a + b;
}

// Юніт-тести для функції add()
describe('add function', () => {
    // Тест на перевірку додавання двох позитивних чисел
    test('adds two positive numbers', () => {
        expect(add(1, 2)).toBe(3);
    });
})
```

```

});
```

// Тест на перевірку додавання від'ємного та позитивного чисел

```

test('adds a negative and a positive number', () => {
  expect(add(-1, 2)).toBe(1);
});
```

// Тест на перевірку додавання двох від'ємних чисел

```

test('adds two negative numbers', () => {
  expect(add(-1, -2)).toBe(-3);
});
```

// Тест на перевірку додавання нуля до числа

```

test('adds zero to a number', () => {
  expect(add(5, 0)).toBe(5);
});
```

Важайте, що ціль автоматичних тестів не довести працездатність коду, а виявити помилки (баги). Це розуміння одразу показує цінність навіть мінімального покриття коду автотестами. Якщо автотест показує помилку ви можете одразу перевірити вручну чи вона дійсно є (чи тест не хибно позитивний) і одразу почати виправляти свій код, або повернати код на допрацювання. Коли всі тести пройшли, ви робите контрольну перевірку вручну і все. Тобто автотест економить час і сили більше ніж витрачено на його написання.

Рефакторинг (refactoring) — процес редагування програмного коду, внутрішньої структури програмного забезпечення для полегшення розуміння коду та внесення подальших правок без зміни зовнішньої поведінки самої системи.

"Щоразу, коли я роблю рефакторинг, перший крок завжди одинаковий. Мені потрібно переконатися, що я маю надійний набір тестів для цієї частини коду. Перш ніж почати рефакторинг, переконайтесь, що у вас є надійний набір тестів. Рефакторинг змінює програми невеликими кроками, тому, якщо ви припуститеся помилки, легко знайти, де помилка" (Мартін Фаулер, "Рефакторінг")

Огляд коду (Code reviews, рев'ю кода) — це методична оцінка коду, призначена для виявлення помилок, підвищення якості коду та допомоги розробникам у вивчені вихідного коду.

Перехресне рев'ю коду — практика перегляду коду, коли рев'ю коду можуть робити учасники команди будь-якого рівня (а не тільки розробники однакового рівня). Молодший спеціаліст може переглядати код старшого спеціаліста. Якщо рев'ю коду робить молодший спеціаліст, тоді назначають декількох рев'юверів коду. Рев'ю коду робиться на основі узгоджених стандартів. Ця практика одночасно покращує якість коду й рівень знань серед учасників команди.
(Якісний код — робочий код, який влаштовує всю команду.)

Debugger вже використовували в 1960-х роках в компанії IBM.

Дебагер (Debugger) — це інструмент, який дозволяє вам виконувати крокування та відстежувати виконання програми.

В JavaScript можна використовувати ключове слово `debugger`, в Node.js слово `inspect`.

Найпоширеніша помилка, яку роблять налагоджувачі-початківці, полягає в спробі вирішити проблему шляхом внесення експериментальних змін до програми.

Не вносіть жодних змін, доки не дізнаєтесь причину помилки. Шукаємо першопричину (root cause analysis), а не лікуємо симптоми.

Налагодження (debugging) — це процес, який складається з двох частин — виявлення помилки та її усунення.

Якщо у вас виникла помилка, спочатку перегляньте програму та поясніть, як вона працює, це допоможе перевірити, чи вона розроблена так, як ви думаете.

Локалізуйте помилку, виключивши частину коду.

Використовуйте відладчик (`debugger`) для проходження виконання програми.

Як тестиувати штучний інтелект (AI) ?

Так само як перевірити знання людини.

Принципи тестиування ШІ:

1. Тестиування повинно бути безперервне (Continuous testing and monitoring).

Зазвичай, випущене програмне забезпечення без ШІ (називемо його класичним) потребує повторного тестиування лише після внесення змін. Однак, тестиування додатків ШІ відрізняється. Необхідно постійно навчати (retrain) існуючу модель ШІ, включаючи запобігання деградації ШІ, перенавчання (overfitting and underfitting), та налаштовувати програмне забезпечення під нові дані та вхідні параметри. Отже й тестиування повинно бути постійне (за розкладом).

2. Відповідь (output) ШІ недетермінована, а ймовірнісна (псевдовипадкова).

Це означає, зокрема, що такі системи можуть поводити себе по-різному при однакових вхідних параметрах. Забезпечення якості в цьому контексті є більшим, ніж просто перевірка відсутності помилок у коді. Тому має бути забезпечення якості даних, перевірка точності прогнозів ШІ. (Уявіть, що вам потрібно тестиувати квантову систему, яка не детермінована, відповідно до класичної моделі).

3. Грунтуючись на AI benchmarks під час тестиування та метриках.

Для тестиування ШІ потрібні інструменти для перевірки даних, моніторингу продуктивності моделі та автоматизованих каналів навчання (retrain). Зазвичай використовуються фреймворки та бібліотеки, такі як TensorFlow Extended (TFX), MLflow.

Еталонні показники, наприклад, SuperGLUE and MLPerf Benchmarks.

4. Поняття баг (дефект) замінюється поняттям "точність" (accuracy of prediction and results).

Класичне тестиування зосереджено на помилках у коді, таких як синтаксичні помилки, логічні помилки та помилки виконання. Класичне тестиування орієнтоване на функціональність, продуктивність, безпеку та інтерфейс користувача.

Тестиування ШІ більше зосереджено на точності моделі, якості даних, на яких вона навчалася, її продуктивності на різних наборах даних та на тому, як вона узагальнює нові дані.

Ось декілька основних прикладів задач з набору SuperGLUE:

- BoolQ (Boolean Questions): Задача полягає в відповіді на питання закритого типу (так/ні), яке базується на фрагменті тексту.
- CB (CommitmentBank): Тест на розуміння тривіальних і зобов'язувальних висловлювань у контексті тексту. Модель має визначити, чи стверджує, відкидає, чи залишається нейтральною до тверджень, поданих у тексті.
- COPA (Choice of Plausible Alternatives): Вибір правдоподібної причини або наслідку для даної події з двох варіантів.

- MultiRC (Multi-Sentence Reading Comprehension): Задача з розумінням багатопропозиційних текстів, де необхідно відповісти на питання і вказати, які речення з тексту підтверджують відповідь.
- ReCoRD (Reading Comprehension with Commonsense Reasoning Dataset): Задача, яка вимагає від моделі заповнити пропуски в тексті на основі контексту, використовуючи відповідні імена або замінники.
- RTE (Recognizing Textual Entailment): Визначення того, чи логічно випливає одне твердження з іншого.
- WiC (Words in Context): Визначення, чи одне і те ж слово в двох різних реченнях має однакове значення.

Що таке якісний код (\approx чистий код)?

Абсолютні показники якості коду:

1. Надійність (код працює й обробляє помилки),
2. Консистентність (стиль, інструменти, назви, типи),
3. Однозначність (мови програмування за замовчуванням однозначні),
4. Надлишковість сутностей (обгортки, які не несуть користь і які можна просто видалити) - принцип "Бритва Оккама".
Принцип Оккама: "Не множте принципи та сутності без необхідності".
5. Безпека в термінах шкоди (Яку шкоду можна зробити цим кодом).

Все інше відносно вимог та конкретної команди.

Наприклад, Ефективність, Читабельність, Масштабованість (Scalability), Розширюваність (Extensibility), Портативність (Portability), Простота розуміння, не є абсолютними показниками якості коду, бо встановлюються відповідно до конкретної задачі, команди (team style guide) й парадигми програмування.

Наприклад, для деяких задач найефективніший алгоритм $O(n)$, а для інших $O(1)$, чи $O(\log n)$.

Як писав Кант, іноді коротший текст забирає більше часу на прочитання (бо потрібно довше думати, щоб його зрозуміти).

Використовуйте чіткі назви змінних, функцій, класів тощо.

Для функцій використовуйте дієслова, для змінних та класів іменники.

Ось стилі написання:

CamelCase: Це стиль, в якому кожне слово у назві розділяється великою літерою, окрім першого слова. Наприклад: myVariableName, calculateInterestRate, getUserInfo.

snake_case: Це стиль, в якому слова у назві розділяються нижніми підкресленнями. Наприклад: my_variable_name, calculate_interest_rate, get_user_info.

Hungarian notation: Це стиль, в якому до назви змінної додається префікс, що вказує на тип даних цієї змінної. Наприклад, "n" може вказувати на числовий тип, "s" - на рядковий, "b" - на булевий.

Наприклад: nCount, sFirstName, bIsActive. Клас — CPerson, Інтерфейс — IHuman.

Які є принципи налагодження програм (debugging)?

Тестувальник може розглядати помилку (дефект) як будь-яке відхилення від очікуваної функціональності або специфікацій продукту, незалежно від того, чи є це логічна помилка, синтаксична помилка чи щось інше. Такий підхід допомагає забезпечити повністю функціональний і коректно працюючий продукт, враховуючи потреби користувачів та очікування.

Логічна помилка — це помилка в програмі, яка призводить до неправильної роботи програми через невірну логіку або алгоритм. Наприклад, якщо в програмі є неправильно написаний умовний оператор, що призводить до невірних вирішень програмою, це може бути прикладом логічної помилки.

Синтаксична помилка, з іншого боку, виникає, коли програма має невірно написаний синтаксис, який не відповідає правилам мови програмування. Наприклад, якщо ви забули поставити крапку з комою або використовуєте невірне ключове слово, це може привести до синтаксичної помилки.

Отже, основна відмінність між ними полягає в тому, що логічна помилка виникає через неправильну логіку, а синтаксична — через невірний синтаксис мови програмування.

Принципи налагодження програм:

1. Не змінюйте код, якщо не виявили суть проблеми.
2. Локалізація, Ізоляція: Відключайте частини програми (модулі), щоб виявити, який модуль спричиняє помилку. Використовуйте конструкцію `try{} catch(error){}` для ізоляції частин коду під час вилову помилки.
3. Метод каченя — це практика, коли програміст намагається розв'язати проблему або зрозуміти складну задачу, розмовляючи про неї з іншою людиною або навіть об'єктом, як от гумове каченя. Метод описаний в книзі “Прагматичний програміст”. Суть методу полягає в тому, щоб викласти проблему або завдання так, ніби ви пояснююте його іншій особі, навіть якщо ця особа фактично не відповідає. Це допомагає розібратися в проблемі, оскільки, говорячи про неї, людина змушенена систематизувати свої думки і з'ясувати, що саме вона не розуміє або де саме виникла проблема. Також цей метод може допомогти виявити рішення або новий підхід до проблеми. Метод часто застосовують математики, задача яких формально описати всі інтуїтивні припущення.
4. Time traveling debugging — це метод налагодження програмного забезпечення, що дозволяє розбирати і виправлювати помилки, шляхом переміщення у часі під час виконання програми. Він дозволяє розглядати стан програми на різних етапах виконання, включаючи минулі моменти, коли помилка ще не виникла.
Основні підходи до time traveling debugging включають запис та відтворення стану програми на різних етапах, використання засобів налагодження, що підтримують перехід у часі, а також використання спеціалізованих інструментів, які автоматизують цей процес.
Цей метод може бути корисним для виправлення складних помилок, які з'являються лише в певних умовах або на певних етапах виконання програми. Він дозволяє розглядати та аналізувати весь контекст, який призводить до помилки, що допомагає розуміти її причини та знайти ефективне вирішення.
Покрокове виконання коду (stepping та breakpoints) означає процес виконання програми по одній інструкції або одному рядку коду за раз. Це дозволяє розробникам спостерігати за станом програми на кожному кроці та аналізувати її поведінку, що часто використовується для налагодження.

З іншого боку, налагодження подорожуючи в часі (Time traveling debugging) — це більш просунута концепція, яка дозволяє розробникам не лише проходити код, але й рухатися назад і вперед у часі під час виконання програми.

5. Використання логування: Будування в програму системи логування, яка записує інформацію про роботу програми, що допомагає виявляти помилки та проблеми.

6. Двійковий пошук: Коли ви зіткнулися зі складною проблемою, подумайте про звуження масштабу проблеми до частин, які можна вирішити. Двійковий пошук по коду — це коли ви ділите код навпіл і систематично звужуєте місце розташування помилки. Такі методи, як коментування коду, використання операторів друку або ізоляція компонентів, можуть допомогти вам швидко виділити проблемну область.

Як встановлювати версії продукту?

Версія продукту — це конкретна ідентифікаційна мітка або номер, який вказує на певний стан або варіант програмного продукту, апаратного забезпечення чи будь-якого іншого продукту. Це може включати в себе числовий код або назву, яка дозволяє визначити конкретну версію продукту.

Версії продуктів випускаються для впровадження нововведень, виправлення помилок, поліпшення функціональності або інших змін. Коли вибираєте або встановлюєте програмне забезпечення, може бути важливою інформацією звертати увагу на версію, оскільки це може вплинути на функціональність, стабільність та безпеку продукту.

API, або Прикладний програмний інтерфейс (Application Programming Interface), це набір правил і протоколів, які дозволяють різним програмам або компонентам програмного забезпечення взаємодіяти один з одним.

API може бути реалізованим у вигляді набору функцій, класів, методів або вебслужб, залежно від конкретного використання. Наприклад, веб-API дозволяють вебсайтам і програмам взаємодіяти через мережу шляхом відправлення та отримання HTTP-запитів.

Правила управління версіями (SemVer Specification)

Для версії MAJOR.MINOR.PATCH, збільшуйте:

MAJOR (головна версія) - коли ви вносите несумісні зміни в API.

MINOR (додаткова версія) - коли додаєте функціональність у сумісний з попередньою версією спосіб.

PATCH (патч-версія) - коли ви виправляєте сумісні з попередньою версією помилки.

Звичайний номер версії МАЄ містити форму X.Y.Z, де X, Y і Z - невід'ємні цілі числа і НЕ МАЮТЬ включати ведучих нулів. X - це головна версія, Y - додаткова версія, і Z - патч-версія. Наприклад: 1.9.0 -> 1.10.0 -> 1.11.0.

Major version zero (0.y.z) призначена для початкового розроблення. Будь-що МОЖЕ змінитися в будь-який момент. Загальний API НЕ ПОВИНЕН вважатися стійким.

Версія 1.0.0 визначає загальний API. Спосіб, яким збільшується номер версії після цього випуску, залежить від цього загального API та того, як воно змінюється.

Патч-версія Z (x.y.Z | x > 0) МУСИТЬ бути збільшена, якщо вводяться лише сумісні зворотні виправлення помилок. Виправлення помилок визначаються як внутрішні зміни, що виправляють некоректну поведінку.

Мінорна версія Y (x.Y.z | x > 0) МУСИТЬ бути збільшена, якщо в загальний API вводиться новий, сумісний зворотний функціонал. Вона МУСИТЬ бути збільшена, якщо будь-який функціонал загального API відрізняється як застарілий. Вона МОЖЕ бути збільшена, якщо в приватному коді

вводиться значний новий функціонал або покращення. Вона МОЖЕ включати зміни рівня патча. Патч-версія МУСИТЬ бути скинута на 0, коли збільшується мінорна версія.

Мажорна версія X (X.y.z | X > 0) МУСИТЬ бути збільшена, якщо в загальний API вводяться будь-які зворотно несумісні зміни. Вона ТАКОЖ МОЖЕ включати зміни рівні мінора і патча. Патч- і мінорні версії МУСИТЬ бути скинуті на 0, коли збільшується мажорна версія.

Коли мажорна, мінорна і патч версії рівні, версія з передвижком має менший пріоритет, ніж звичайна версія:

Приклад: 1.0.0-alpha < 1.0.0.

Також: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

Альфа версія (alpha): Це перша стадія розробки програми. В цьому етапі функції можуть бути лише частково реалізовані, можуть бути помилки та деякі вузькі місця. Зазвичай, цю версію використовують внутрішньо у команді розробників для тестування базового функціоналу.

Бета версія (beta): Це наступний етап після альфа версії. Бета версія — це вже більш стабільна версія програми, але все ще може містити помилки. Її вже можуть випробовувати зовнішні користувачі, що допомагає виявляти більше помилок та збирати фідбек для подальших поліпшень.

Реліз кандидат (rc): Це версія програми, яка вважається готовою для випуску, якщо тестування пройшло успішно. Реліз кандидат зазвичай має зафіксовані усі відомі серйозні помилки.

Стабільна версія (stable): Це фінальний варіант програми, який вважається готовим для загального використання. В цій версії виправлені більшість відомих помилок і вона пройшла всі тести, випробування та перевірки, щоб забезпечити стабільність та надійність.

Ці терміни використовуються для позначення різних етапів розвитку програмного забезпечення та вказують на рівень його готовності до використання користувачами.

Форма Бекуса-Наура для правильних версій SemVer

```
<valid semver> ::= <version core>
    | <version core> "-" <pre-release>
    | <version core> "+" <build>
    | <version core> "-" <pre-release> "+" <build>

<version core> ::= <major> "." <minor> "." <patch>

<major> ::= <numeric identifier>

<minor> ::= <numeric identifier>

<patch> ::= <numeric identifier>

<pre-release> ::= <dot-separated pre-release identifiers>

<dot-separated pre-release identifiers> ::= <pre-release identifier>
    | <pre-release identifier> "." <dot-separated pre-release identifiers>

<build> ::= <dot-separated build identifiers>

<dot-separated build identifiers> ::= <build identifier>
    | <build identifier> "." <dot-separated build identifiers>

<pre-release identifier> ::= <alphanumeric identifier>
    | <numeric identifier>

<build identifier> ::= <alphanumeric identifier>
    | <digits>

<alphanumeric identifier> ::= <non-digit>
    | <non-digit> <identifier characters>
    | <identifier characters> <non-digit>
    | <identifier characters> <non-digit> <identifier characters>

<numeric identifier> ::= "0"
    | <positive digit>
    | <positive digit> <digits>

<identifier characters> ::= <identifier character>
    | <identifier character> <identifier characters>

<identifier character> ::= <digit>
    | <non-digit>

<non-digit> ::= <letter>
    | "-"

<digits> ::= <digit>
    | <digit> <digits>

<digit> ::= "0"
    | <positive digit>
```

<positive digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<letter> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
| "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
| "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"
| "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"
| "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"
| "y" | "z"

Як оцінювати якість продукту?

Програмний продукт = код + тести + документація (+ ліцензія + підтримка).

Наступна формула є апроксимацією, яка припускає, що всі фактори якості продукту можна наблизено перевести в кількість витраченого часу на створення, налагодження та фідбек стосовно продукту.

Якість реалізації фічі (продукту) до етапу тестування обчислюється за формулою:

Якість реалізації фічі = 1 - (Час на виправлення помилок після тестування / Початкова естімація на імплементацію).

При цьому естімація враховує найкращі практики розробки, такі як оптимальність, безпека, зручність підтримки та зручність в користуванні.

Якість реалізації фічі на рівні 0.8 (до етапу QA - quality assurance) вважається нормальним показником, але вказує на потребу в додатковому часі для виправлення помилок (+20% від початкової оцінки на імплементацію).

Наприклад, якщо розробник витратив 5 днів на реалізацію фічі та 1 день на виправлення помилок, то якість реалізації фічі дорівнює 1 - 1/5, тобто 0.8.

Ідеальним варіантом є досягнення якості реалізації фічі, рівної 1, але це може бути важко досяжним на практиці через обмежені ресурси та інші фактори.

Якщо якість реалізації дорівнює 0, це означає, що час на виправлення помилок дорівнює часу на розробку.

Якість реалізації менше за 0 вказує на дуже низьку якість роботи.

Ця формула досить універсальна, щоб використовувати її для оцінки роботи як програміста та й всієї команди (якщо врахувати бета-тестування).

Історичним прикладом поганої естімації часу та вартості розробки може слугувати приклад Аналітичної машини Беббіджса, яку він так і не завершив самостійно. Також декілька мейнфреймів IBM вийшли за рамки бюджету та термінів. Однак слід зауважити, що Чарлз Беббідж був видатною постаттю у світі розробки.

Доцільно вчитися на прикладах минулого, щоб уникнути подібних помилок у майбутньому.

Ось складніша формула для оцінки якості продукту:

Якість = (Функціональність + Продуктивність + Надійність + Безпека + Практичність +
Масштабованість + Обслуговування) / Вартість;

Де:

- Функціональність оцінюється від 0 до 10 і представляє наскільки функціональність відповідає вимогам та очікуванням користувачів.
 - Продуктивність оцінюється від 0 до 10 і представляє швидкість та ефективність функціональності.
 - Надійність оцінюється від 0 до 10 і представляє стійкість до відмов та помилок.
 - Безпека оцінюється від 0 до 10 і представляє захищеність функціональності від потенційних загроз.
 - Практичність оцінюється від 0 до 10 і представляє зручність та легкість використання функціональності.
- Масштабованість: Оцінка (від 0 до 10) можливості функціоналісті масштабуватися для обробки більшої кількості даних або користувачів без втрати продуктивності.
- Обслуговування та стандарти коду: Оцінка (від 0 до 10) того, наскільки функціональність легко піддається обслуговуванню, розширенню та модифікаціям в майбутньому.
- Вартість представляє витрати на розробку та обслуговування функціональності.

Альфа-тестування — це вид приймального тестування, що проводиться в тестовому середовищі розробника особами поза розробницькою організацією.

Бета-тестування — це вид приймального тестування, що проводиться на зовнішньому сайті відносно тестового середовища розробника особами поза розробницькою організацією.

Наприклад, якщо компанія розробляє новий софт, вони можуть провести альфа-тестування, запрошуючи внутрішніх користувачів або тестерів, які не є частиною розробницького відділу, для перевірки програмного забезпечення в їх власному тестовому середовищі.

Натомість бета-тестування може включати випробування програми або продукту на реальних користувачах або в зовнішньому середовищі, яке відрізняється від тестового середовища розробника. Наприклад, компанія може надати доступ до бета-версії свого програмного забезпечення обраним клієнтам або групі тестерів для випробування перед публічним релізом.

Під час створення якогось продукту, пристрою, важливим є:

1. Створювати тестові прототипи.
2. Використовувати комп'ютерне моделювання.
3. Абстрагуватись, тобто виділяти ключові ознаки продукту.
4. Порівнювати свій продукт з продуктами наявними на ринку.
5. Вивчати наукове підґрунтя вашого продукту.

У програмній інженерії, "Don't repeat yourself" (**DRY**, укр. не повторюйся) — принцип розробки програмного забезпечення, що направлений на уникнення дублювання інформації будь-якого вигляду (наприклад, програмний код чи текст інтерфейсу користувача). Принцип був сформульований Енді Хантом та Дейвом Томасом в їх книзі *The Pragmatic Programmer* наступним чином: "Будь-яка інформація повинна мати єдине, однозначне, авторитетне представлення в системі". На думку авторів, принцип має застосовуватися також в "схемах баз даних, тест-планах, в системах збірки, навіть в документації". Правильне використання DRY дозволяє розробникам робити атомарні зміни в системі, коли модифікація одного елементу системи не вимагає модифікації сторонніх елементів. Ті ж елементи, що мають логічне відношення до модифікованого, змінюються прогнозовано та одноманітно.

Прагматичний програміст: від підмайстра до майстра — книга про комп'ютерне програмування та програмне забезпечення, написана Ендрю Гантом та Девідом Томасом та видана в жовтні 1999 року.

В книзі "Прагматичний програміст" автори використали вислів Гіппократа: "Головне, не нашкодь".

У книзі використовуються аналогії та оповідання для представлення методологій розвитку та застережень, наприклад, теорія розбитих вікон, історія кам'яного супу чи жаби в окропі. Деякі поняття були впроваджені або популяризовані в книзі, наприклад, кодові кати, невеликі вправи для відпрацювання навичок програмування та зневадження гумової качки — метод налагодження, назва якого покликається до розповіді у книзі.

Теорія розбитих вікон — теорія, яку сформулювали Джеймс Квінн Вілсон і Джордж Келлінг у 1982 році. Відповідно до цієї теорії, якщо хтось розбив скло в будинку і ніхто не вставив нове, то незабаром жодного цілого вікна в цьому будинку не залишиться, а потім почнеться мародерство. Іншими словами, явні ознаки безладу і недотримання людьми прийнятих норм поведінки провокують оточення теж забути про правила.

Якщо нехтувати чистотою коду, то згодом він стане нестерпним.

Метод каченяті — це метод розв'язання задачі через делегування її уявному помічнику. Метод описаний в книзі "Прагматичний програміст".

Суть методу полягає в тому, що ви ставите (чи уявляєте) на робочому столі іграшкове каченятко, і коли в вас виникає запитання, на яке важко дати відповідь — задаєте його іграшці, так неначе вона справді може відповісти. А правильне формулювання питання, як відомо, містить половину відповіді.

Також використовується при зневадженні. Якщо певна частина програми не працює, програміст пробує пояснити каченяті, що робить кожен рядок програми, і в процесі цього знаходить невідповідність синтаксису й прагматики.

Автори книги "Прагматичний програміст" (1999) пишуть що базовий прототип (скелет застосунку) ніби Трасувальна куля, яка допоможе вам попасті в ціль.

Базовий прототип — це спрощений, мінімальний застосунок або варіант програми, який створюється на початковому етапі розробки для того, щоб визначити основні аспекти проекту і визначити напрямок подальшої роботи.

Аналогія з трасувальною кулею полягає в тому, що такий базовий прототип допомагає розробникам "попасті в ціль", тобто зрозуміти, яким чином проект може бути реалізований і які проблеми можуть виникнути на початковому етапі розробки. Він служить свого роду "трасувальною кулею", яка допомагає спрямувати розробку у правильному напрямку та уникнути серйозних помилок і зайвого витрат часу на пізніших етапах проекту.

Ця книга стала досить популярною серед програмістів і розробників програмного забезпечення і містить багато корисних порад і підходів до роботи над програмними проектами.

В книзі "Чистий код" Роберт Мартін пропонує використовувати правило бойскаута для коду. Правило бойскаута: Залишай місце чистішим, ніж ти його знайшов. В контексті коду: Залишай код чистішим, ніж ти його знайшов.

Коли ми вводимо поняття "**людино-година**", ми робимо припущення: що всі люди (припустимо, інженери ПЗ) мають однакові вміння, а отже можна чітко визначити час на виконання певної задачі. Це припущення вже невірне, коли в команді є старші та молодші спеціалісти, які роблять одне й те саме за різний час. Крім того, поняття "людино-година" говорить про те, що збільшення кількості людей буде пропорційно відображатися на зменшенні терміну створення продукту, але це міф. Цю помилку розбирав Фред Брукс у своїй книзі "Міфічний людино-місяць". Фредерік Брукс пише, що потрібно врахувати витрати на комунікацію між інженерами, які в деяких випадках можуть бути настільки великі, що виконується закон Брукса: "Додавання робочої сили до запізнілого програмного проекту затягує його ще більше". Додавання більшої кількості людей до завдання, яке дуже поділене, наприклад, прибирання номерів у готелі, зменшує загальну тривалість завдання (аж до моменту, коли додаткові працівники заважають один одному). Однак інші завдання, включаючи багато спеціальностей у проектах програмного забезпечення, менш подільні. Щоб вирішити ці недоліки поняття "людино-година", були створені story points в Scrum.

Story point (Очки складності завдання) - це одиниця виміру, яка використовується в методології розробки програмного забезпечення для оцінки складності задачі. Вона використовується для приблизної оцінки зусиль, не прив'язуючись до конкретної часової рамки.

Коли команда розробників оцінює задачу, вони приймають у розгляд три основні фактори: складність, обсяг роботи та ризики. Вони потім призначають задачі певну кількість story points, яка відображає загальну складність. Чим більше story points, тим складніше завдання. Це дозволяє команді спрогнозувати, скільки робочих годин знадобиться на виконання задачі та планувати роботу відповідно.

Між story point та часом не має прямопропорційної залежності.

З допомогою Story points визначається velocity of Scrum team.

Аксіоми Story points (Очки задачі):

1. Story points відносні (поставте якесь опорне завдання).
2. Story points представляють відносні зусилля, складність і ризики для завдання. (Складність стосується структури, а не часу).
3. Story points є лінійними: $2 \text{ SP} = 1 \text{ SP} * 2$. (Одна 3-points story повинна вимагати приблизно стільки ж роботи, скільки інша 3-points story).
4. Не існує найбільшого числа сторіпойнтів (story points). (На практиці кладуть ліміт, наприклад, якщо одна задача більше ніж 13 сторіпойнтів, розділяємо її на дві).

Деталі про Story points в книгах:

"Agile Estimating and Planning" by Mike Cohn,
"Learning Agile" by Andrew Stellman.

Що таке ринкова економіка й відповідність зарплати?

“Не слухай тих, хто обіцяє тобі багатство за одну ніч, мій хлопчуку. Як правило, вони або дурні, або шахрай!”, — порада цвіркуна з книги “Пригоди Піноккіо” Карла Коллоді.

“Економіка — це дослідження людства у звичайних справах життя; вона досліджує ту частину індивідуальної та соціальної діяльності, яка найтісніше пов’язана з досягненням і використанням матеріальних реквізитів доброту. Таким чином, з одного боку, це дослідження багатства; і з іншого, більш важливого боку, частина вивчення людини” (Альфред Маршалл, “Принципи економіки”, 1890)

Однією з найраніших праць з економіки є "Економіка" грецького філософа Ксенофонта, який жив у V-IV століттях до н.е. Це давньогрецький трактат, в якому він розглядає різні аспекти управління домашнім господарством, включаючи сільське господарство, виробництво, управління рабами та фінанси.

Планова економіка — це державна економіка, в якій трудові ресурси виділяються на виконання заздалегідь розробленого господарського плану і виконання темпів виробництва певного товару відповідно до господарського плану. Ціноутворення повністю контролюється державою. У плановій економіці де-факто чи де-юре все виробництво належить державі. Простіше кажучи, держава з плановою економікою методом директив вказує, який продукт і приблизно в якій кількості слід виробляти. Крім того, в плані також чітко прописана кількість грошей або товарів, яку отримає робітник, залежно від того, яку посаду обіймає робітник.

Переваги планової економіки

1. дозволяє ефективно мобілізувати ресурси для реалізації конкретного економічного плану.
2. дозволяє заздалегідь закласти в план найнеобхідніше і тим самим забезпечити країну товарами стратегічного значення, а саме: харчовими продуктами, одягом, житлом, ліками, зброєю.
3. зниження собівартості продукції внаслідок ефекту масштабу. Виробництво налаштовується один раз, а не кілька разів, як у системі конкуренції.

Недоліки планової економіки

1. Передбачається, що економічні цілі нинішньої влади ідентичні цілям всього народу або переважної більшості, що не завжди вірно.
2. Оскільки в плановій економіці попит і пропозиція не враховуються динамічно (у реальному часі), то виявляється, що кількість певного товару на ринку не відповідає бажанням покупців, і навпаки, продукту, на який попит невеликий, може бути більше на ринку. Таким чином, певна продукція буде дефіцитною лише через неправильне складення господарського плану, а не через об’єктивні причини (війна, голод, епідемія). Через те, що держава розподіляє гроші між людьми, тобто визначає купівельні можливості кожного і водночас визначає, скільки й в якій кількості товару буде в магазинах, може виникнути ситуація, коли людина купує певний товар у магазині, а інший, хто хоче його купити, вже не може, оскільки цього товару немає в наявності.
3. Відсутність ринкової конкуренції, яка загрожує тим, що людям продадуть поганий товар, але у них не буде вибору, тобто можливості купити інший.
4. Тривалий процес планування в централізованих системах.
5. Заборона приватного бізнесу.

Ринкова економіка — це державна економіка, в якій держава не визначає, який продукт виробляти та в яких кількостях, а виробник визначає це та ціноутворення сам, залежно від попиту та пропозиції на ринку. Загалом усі учасники ринку є приватними підприємцями, а державні підприємства зведені до мінімуму і в кінцевому підсумку повністю виведені з вільного ринку (при цьому вони можуть працювати на інших законних засадах). Ринкова економіка передбачає рівний доступ до земних

ресурсів і заборону монополізму. Неможливо, щоб держава надала можливість володіти корисними копалинами лише певним людям.

Переваги ринкової економіки

1. Конкуренція між підприємцями загалом сприяє виробництву найкращого продукту та пропонуванню найкращих послуг.
2. Самоорганізація. Попит і пропозиція, тобто кількість послуг і кількість запитів на ці послуги або товари, дають можливість сформувати об'єктивне ціноутворення і відповідне виробництво.

Недоліки ринкової економіки

1. Необхідність досить складного приватного моніторингу ринку для адекватного ціноутворення та товарного виробництва.

2. Необхідність реклами.

3. Неможливість бути повністю незалежною від держави, точніше, держава повинна частково контролювати стратегічно важливі сфери економіки, а саме: забезпечувати продовольство, медичне та військове постачання, не порушуючи при цьому законів вільного ринку.

4. Ринок сам по собі не в змозі усунути або компенсувати шкоду, заподіяну зовнішніми ефектами. На практиці, коли виникають серйозні проблеми, потрібне державне втручання. Держава запроваджує жорсткі стандарти, обмеження, використовує систему штрафів, визначає межі, які учасники господарської діяльності не мають права переступати. У міру зростання суспільного багатства проблема зовнішніх ефектів стає все гострішою. Збільшення кількості автомобілів у використанні супроводжується забрудненням повітря.

Ринкова та планова економіка передбачає контроль за якістю продукції та її безпекою.

Імператор Діоклетіан в третьому столітті нашої ери наказав встановити фіксовані ціни на певні товари, що мало негативні наслідки, призвело до зменшення якості продукції, зменшення виробництва та розквіту чорного ринку.

Елементарна формула для розрахунку ціни послуги: $s = (a * t) + b + c;$

s - ціна послуги.

a - ціна середньої заробітної плати за годину роботи, яка доступна практично кожному і не вимагає особливих навичок (люди-година).

Простіше кажучи, "а" виражає погодинний заробіток, який ви точно могли б отримати, якби не були зайняті іншою роботою.

t – час, витрачений на виконання послуги (тобто на роботу).

b - витрати на матеріал, електроенергію тощо, тобто на витрати, які не залежать від вас. Необхідно прагнути до скорочення числа b шляхом пошуку більш вигідних постачальників.

Формула $((a * t) + b)$ показує собівартість товару. Потрібно зменшувати собівартість товару.

c - додаткова ціна щодо попиту та пропозиції на ринку. c - максимальна ціна, яку можна додати до основної ціни, щоб не порушити попит на товар. Це число (c) встановлено емпірично. Ця компонента, тобто c , повинна відображати динаміку ринку, чим компонента більша, тим більший попит на ваш товар, тобто якщо покупці готові взяти товар з великим c , то є попит на ваш продукт. Попит на продукт може бути обумовлений його унікальністю або високою якістю. Можна сказати, що компонента c відображає суму, яка була додана до початкової суми $((a * t) + b)$ в наслідок аукціону, тобто перехресних пропозицій від покупців.

Умовний Ваня Айвазовський намалював картину. Він витратив на матеріали (фарби, полотно, обортку, оренду студії, комуналку тощо) суму а грошей. Далі, за весь час малювання він міг би заробити суму в грошей. Вважаємо, що Ваня прагне працювати швидко, але в комфортному для себе режимі (без перевтоми). Тоді $a + b$ буде собівартістю його товару (або послуги). З цієї суми він почне аукціон. Хто найбільше запропонує за картину ($+ c$), тому він її продасть. Ця формула дуже універсальна. Припустимо, що Ваня прагне зменшити собівартість картини. Він намагається пришвидшити свої результати, але без шкоди для якості товару й здоров'я. Тоді він буде заробляти більше (точніше менше робити, але отримувати те ж саме). Та чи підходить ця формула для працівників, які беруть замовлення, або є найманими?

Якщо ви берете замовлення, ви можете заздалегідь прикинути термін виконання роботи й сказати клієнту суму за яку вам буде комфортно працювати, враховуючи вищеописану формулу, тобто собівартість та прибуток, який можна встановити аналізуючи ринок, й попит замовника.

Друге діло компанії.

В компаніях вам дають якусь ставку (обговорену суму зарплати) й за неї хочуть побачити певні вміння та якість роботи, що відповідають ринковій якості. Ринкова якість встановлюється умовно Ванею й подібними учасниками ринку, який працює як підприємець, за вищеописаною схемою. Чим менше вимагає компанія й чим більше платить, тим вона вигідніша, якщо не враховувати інші фактори (інтерес, досвід, навчання, спілкування). Тому чим більше вимоги, тим більша повинна бути зарплата.

Проста формула прибутковості пропозиції: Зарплата / вимоги.

Або, Зарплата / складність роботи.

Також:

Зарплата / (вимоги * складність роботи).

"Виробництво — це не купівля за низькими цінами та продаж за високими. Це процес закупівлі матеріалів чесно і, з мінімальними можливими додатковими витратами, перетворення цих матеріалів у товар для споживача. Азарт, спекуляція та гострі угоди лише заважають цьому процесу" (Генрі Форд, "Мое життя і робота")

Вигідність товару й послуги для споживача: Ціна / Якість.

Хто такий технічний лідер в команді?

Перед розглядом посади "технічний лідер", розглянемо структуру команди.

В книзі "Міфічний людино-місяць" Фред Брукс пише, що команди розробників, які спільно працюють над задачею, повинні бути невеликі, приблизно 10 людей.

Джефф Сазерленд і його Scrum методологія також каже, що команда повинна складатися з 4-10 людей.

Якщо проект величезний, то організовуються декілька самодостатніх команд, кожна з яким працює над своєю частиною проекту. Це може навіть відбиватися на архітектурі проекту, зокрема, використовують мікросервісну архітектуру та мікрофронтенд. Між-командну взаємодію виконують лідери команд та технічний директор (СТО).

І Фред Брукс і Scrum-методологія вважають, що команда повинна бути крос-функціональна.

Команда з крос-функціональним складом включає в себе фахівців із різних областей, необхідних для виконання проекту чи завдання. У контексті Scrum це зазвичай означає наявність учасників із різними навичками у сферах дизайн, розробки, тестування та інших необхідних компетенцій. Це різноманіття допомагає команді вирішувати різні аспекти проекту без постійного залучення зовнішніх ресурсів, сприяючи ефективності та співпраці. QA спеціаліст (тестувальник) вважається частиною Scrum команди та слугує, щонайменше для двох цілей:

1. Програміст не повинен виключно сам тестувати свій код (як каже книга "Мистецтво тестування" Гленфорда Майерса).
2. Звіт менеджеру проекту про відсутність виявлених багів (помилок). Помилкою (багом) вважається відхилення від специфікації.

Відповідно до Scrum команди є **багатофункціональними** (cross-functional), тобто учасники мають усі навички, необхідні для створення цінності кожного спринту (ітерації розробки). Хоча багатофункціональні команди повинні мати різноманітний набір навичок, це не означає, що кожен член повинен бути експертом у всіх сферах. Основна увага приділяється володінню навичками, необхідними для поступового постачання продукту. У той час як команди можуть вибирати людей, які мають Т-подібний набір знань, деякі команди можуть вибирати експертів з бізнес-аналізу або експертів з тестування.

(Scrum is defined in the Scrum Guide by Ken Schwaber and Jeff Sutherland).

Т-подібні навички (T-shaped skills) — це комбінація глибоких знань у конкретній області (вертикальна частина "T") і широкого спектра загальних навичок (горизонтальна частина "T"). Це означає, що людина має глибокі експертні знання в одній області, але також може працювати та спілкуватися ефективно в інших галузях.

Технічний лідер та лідер команди повинні мати Т-подібні навички, а не фреймове мислення.

Технічний лідер відповідальний за технічну придатність проекту. Він повинен контролювати, щоб інші, менш досвідчені розробники, не зробили критичні помилки.

Фред Брукс пропонує розглядати технічного лідера в команді, як хірурга в операційній. В операційній є головний хірург і його асистенти, які йому допомагають. Хірург робить головні рішення. В такому випадку маємо чітку ієархію в команді та обов'язки. Звісно, що така команда не може бути великою, до 10 людей. Цей підхід особливо добре працює, якщо в команді є початківці, або спеціалісти середнього рівня, які не мають достатньої експертизи. Цей підхід працює навіть, якщо в команді всі спеціалісти високого рівня. Тех лід пояснює структуру проекту й каже, що кому робити відповідно до його навичок. Тех лід може консультуватися з учасниками команди, впроваджувати перехресне рев'ю

коду. На тех ліді відповідальність за технічну частину проекту, тому він повинен контролювати дії та знання інших розробників, вказувати на те, що потрібно вивчити й змінити.

Важливо зауважити, що члени команди не повинні боятися бути ініціативними й навіть показувати, що можливо є краще рішення, ніж те, що пропонує технічний лідер. Але вони ні в якому разі не повинні наполягати на ньому, бо відповідальний саме технічний лідер за критичні рішення, а інші члени команди відповідальні за свій спектр задач і за те, як вони комунікують з лідером.

Тех лід може виконувати роль тім ліда, але також це можуть бути окремі ролі.

Обов'язки лідера команди (team lead):

1. Спілкування з Product owner (Власник Продукту (Product Owner) — представляє зацікавлені сторони (stakeholders) та є голосом клієнта).
2. Спілкування з менеджером проекту.
3. Спілкування з іншими командами.
4. Контроль комунікації в команді.
5. Розгляд кар'єрного росту (dev plan).
6. Пріоритет задач.
7. Складання roadmap та беклогу (списку задач).
8. Комунікація з технічним лідером.
9. Деякі технічні задачі, код рев'ю.
10. Документація.

Team lead (тім лід) тільки один в команді, технічних лідерів може бути декілька. Наприклад, командний лідер може контролювати всю Scrum крос-функціональну команду, в якій є група бекенд розробників та група фронтенд розробників, кожна з яких має свого технічного лідера.

Обов'язки технічного лідера (tech lead):

1. Архітектура й стек технологій проекту.
2. Перехресне код рев'ю.
3. Технічний менторінг спеціалістів.
4. Введення стандартів коду.
5. Налаштування CI/CD (з DevOps).
6. Технічна документація.

Технічний лідер та Лідер команди — це ролі, а не рівень експертизи. В команді можуть бути всі спеціалісти високого рівня, але тільки один з них виконує роль технічного лідера.

Найм працівників виконує тех лід разом з тім лідом. (Якщо тім лід і тех лід є однією особою, тоді bus factor збільшується, і можливість зірвати терміни релізу, deadline, також).

Лідер команди (тім лідер) повинен добре комунікувати з командою, залучати технічного лідера для обговорення технічних аспектів, налагоджувати розвиток команди.

Задача тім ліда й тех ліда оцінювати рівень розробників. Для цього розробляється спеціальний список питань. Крім того, потрібно оцінювати якість роботи розробника. Менеджер скоріше за все не буде мати технічної компетенції, щоб оцінити швидкість і якість виконання задачі. Менеджер звернеться за оцінкою до лідера команди або технічного лідера. Велика помилка коли тім лід, або менеджер оцінює рівень працівника без відповідної на те компетенції (тех лід в допомогу, або незалежний консалтинг).

Програміст повинен аналізувати вимоги, давати естімацію (оцінку задач), робити декомпозицію задачі.

Професійний програміст повинен розуміти, що за методологією Agile ми не вимагаємо детального опису задачі (прояснення вимог переходить на сторону комунікації).

Також є формула оцінки якості виконання задачі перед етапом QA:

1 - (Час на виправлення багів / час початкової естімації).

Зарплату членів команди краще встановлювати за об'єктивними критеріями відповідно до рівня досвіду, складності посади, ринкової зарплати.

Абсолютна система грейдів (junior-ставка, middle-ставка, senior-ставка) хороша тим, що робить розвиток програмістів незалежним й мотивує їх, якщо вона відповідає ринковим стандартам й є адекватною. При такій системі грейдів, програмістів не має цікавити заробітна плата один одного. Є грейд, є вимоги, є ставка цього грейда. Якщо немає прописаних рівнів зарплати й станеться так, що розробник з меншою зарплатою, дізнається про колегу того ж рівня, але з більшою зарплатою, тоді швидше всього він буде працювати менше за свого колегу (це очевидно).

Якості лідера, які описані в книзі Генка Рейнвотера "Як пасти котів":

1. Системне мислення, тобто розуміння зв'язків.
2. Широке мислення всім інтелектом, баланс між логікою й естетикою.
3. Вміння аналізувати.
4. Самокритика, вміння вчитися на помилках.
5. Вміння делегувати завдання.
6. Вміння слухати.
7. Нормативна й чітка мова, вміння висловлюватися.
8. Розуміння процесів та цілей бізнесу.

Антипатерни лідера команди:

1. Диктатор і генерал.

"Диктатор". Девіз цих діячів — "я правий, бо правий". Диктатор намагається нав'язувати своїм підлеглим конкретні методи вирішення поставлених ними завдань. Насправді всім нам не заважає ретельно стежити за тим, щоб не піти цим шляхом. Лідерство дійсно передбачає демонстрацію співробітникам можливих способів роботи, але в цьому слід бути обережним, оскільки позбавляти підлеглих почуття причетності до створення продукту неприпустимо. Диктатор створює навколо себе атмосферу жаху, яка дозволяє співробітникам повною мірою виявляти свої здібності. Справжній лідер повинен створювати умови, в яких розробники могли б пишатися результатами своїх праць, а щоб їхня гордість втілювалася в хороших результатах, їм слід надавати певну свободу вибору засобів досягнення мети.

"Генерали" виявляють значну схожість із диктаторами, виявляючи у своїй діяльності ще більшу жорсткість. Якщо ви уявили себе генералом, а своїх підлеглих вважаєте солдатами, майте на увазі: битви із противником вам доведеться вести одному. Солдати, швидше за все, будуть сидіти десь у кутку, спокійно попиваючи каву. Протистоять такому підходу стиль співпраці — і, ви знаєте, він має широке вживання просто тому, що він правильний. Під час війни співпраця рідко має місце. У сфері розробки програмного забезпечення, навпаки, важко уявити ситуацію, у якій встановленню відносин з урахуванням співробітництва щось перешкодило б.

2. Всезнайка. Такий діяч широ вірить у те, що йому відомо все про його роботу, роботу компанії, про конкретні завдання програмістів. Він буде нехтувати порадами інших. Буде часто робити помилки. (Як казали багато філософів, чим більше знаєш, тим краще бачиш безодні невідомого).

3. Альфа "ботан".

Людина, яка постійно вказує на чужі помилки, як на невігластво. Думає, що всі мають знати те, що він, і ще більше. Такий працівник просто не розуміє психологію людей і що люди різні.

4. "Розумник". Людина, яка постійно любить повторювати очевидні банальні істини, щоб показати свої знання. "Розумник" дуже гордий і честолюбний. Він, чи вона, завжди публічно вкаже на слабкості інших, щоб підняти свою самооцінку за рахунок інших.

Деякі причини для звільнення члена команди:

1. Саботаж роботи, свідоме затягування процесів.
2. Критика і не повага до проджект менеджера, або лідерів команди.
3. Постійне скидання відповідальності та задач на інших.
4. Байдужість до продукту, відстороненість.

Відомий принцип тестування каже: Не плануйте тестування, виходячи з припущення, що помилок не буде виявлено.

Але це не означає, що ви маєте робити припущення про те, що програміст не старанно працює. Завжди вважайте, що розробник свідомо не винен у виникненні помилок, інакше будуть конфлікти, тобто застосовуйте презумпцію невинуватості.

Командний й технічний лідер повинен контролювати, щоб розробники середнього рівня:

1. Розвивались, вивчали необхідні технології.
2. Не робили критичних помилок.
3. Не вводили експериментальні, не надійні технології.
4. Виконували свою роботу.

Якщо спеціаліст справляється зі своїми обов'язками, йому потрібно хвалити за хорошу роботу, щоб він бачив, що ви ним задоволені. Не варто думати, що хвалять тільки за екстраординарні здобутки. Такий зворотний зв'язок допомагає молодшому спеціалісту розуміти, що він на правильному шляху й усуває його тривогу.

Якщо технічний лідер повинен доробляти роботу програміста середнього рівня, тоді це дуже поганий знак, щось йде не так. Задача технічного лідера не робити все за всіх, а делегувати обов'язки, контролювати процес й підказувати, якщо необхідно.

Менеджерам проекту важливо розуміти, що навіть якщо технічний лідер робить тільки якусь спеціалізовану частину проекту, а все інше роблять інші програмісти, яких він контролює, все одно технічному лідеру потрібна ширша експертіза, обсяг знань, що повинно враховуватися в заробітній платі.

Комуникація дуже важлива в команді.

Структура організації та її комунікаційні канали впливають на структуру розроблюваного продукту. Наприклад, якщо команда розробників розділена на кілька підгруп, кожна з яких має обмежені комунікаційні засоби з іншими, то архітектура програмного забезпечення, ймовірно, відобразить ці розділення.

Організації, які створюють системи (що означає проєктування або розробку програмного забезпечення), обмежені у своїх проєктах тими комунікаційними структурами, які вони використовують.

Закон Конвея - "Будь-яка організація що проєктує системи, з великою ймовірністю створить дизайн зі структурою, яка буде копією структури комунікації в компанії".

У своїй колекції нарисів про невдалі програмні проєкти Роберт Гласс (Robert Glass) склав список найбільш поширених "програмних катастроф", зокрема:

1. Неадекватний опис завдань проєкту.
2. Незадовільне планування та оцінка.
3. Застосування нової для цієї компанії технології.
4. Непридатна/відсутня методологія керівництва проєктом.
5. Нестача провідних спеціалістів групи.
6. Зрив домовленостей виробниками апаратного/програмного забезпечення.

Bus factor (фактор автобуса) проєкту — це кількість ключових учасників команди, які у випадку втрати своєї дієздатності, призведуть до неможливості рухати спільній проєкт далі. Bus factor є

мірою зосередження інформації поміж окремими учасниками проекту. Високий bus factor означає, що проект зможе продовжувати розвиватись навіть за несприятливих умов. Вищий bus factor означає більшу стійкість проекту до втрати ключових фахівців. Збалансована розподіленість відповідальності та знань допомагає знизити цей ризик, забезпечуючи більшу стійкість команди до втрати будь-якого окремого члена.

Для зменшення "bus factor" у Scrum, рекомендується:

1. Робити code reviews: Забезпечте, щоб код переглядали інші члени команди, це допомагає із взаєморозумінням коду.
2. Документація: Створюйте докладну технічну документацію, щоб інші розробники могли легко розуміти та працювати з вашим кодом.
3. Навчання команди: Проводьте тренінги та ділітесь знаннями в команді, щоб інші члени могли взяти на себе роль у випадку необхідності.
4. Ротація завдань: Дозволяйте різним членам команди працювати з різними частинами проекту, щоб було більше людей, які розуміють всю систему.

Ці підходи допоможуть розподілити знання і знизити ризик через "bus factor".

Робота вашого менеджера полягає в тому, щоб робити найкраще для компанії та команди, а не робити все можливе, щоб зробити вас щасливими.

Ваш керівник очікує, що ви принесете рішення, а не проблеми.

Артефакти в менеджменті — це конкретні об'єкти, документи або діаграми, що виникають у процесі управління проектами чи бізнес-процесами. Це може включати плани, звіти, графіки Ганта та інші матеріали, які використовуються для організації та контролю проектів. Артефакти грають важливу роль у забезпеченні чіткості та ефективності управління.

Маніфест гнучкої розробки (Agile):

1. Люди та співпраця важливіші за процеси та інструменти.
2. Працюючий продукт важливіший за вичерпну документацію.
3. Співпраця із замовником важливіша за обговорення умов контракту.
4. Готовність до змін важливіша за дотримання плану.

Мікросервісна архітектура є підходом до розробки програмного забезпечення, в якому програма розбивається на невеликі, автономні модулі, відомі як мікросервіси. Кожен мікросервіс виконує конкретну функцію і може комунікувати з іншими мікросервісами через мережу. Цей підхід дозволяє полегшити розгортання, масштабування та розвиток програмного забезпечення. Кожен мікросервіс може бути розроблений та підтримуватися окремою командою розробників.

Перехресне рев'ю коду — практика перегляду коду, коли рев'ю коду можуть робити учасники команди будь-якого рівня (а не тільки розробники однакового рівня).

"Перехід на особистості" — риторичний прийом, коли аргумент направлений не проти самої думки чи позиції, а проти особи, яка її висловлює. Замість того, щоб вести обговорення на рівні ідей чи доказів, людина, що використовує аргумент "Перехід на особистості", намагається підірвати позицію противника, звертаючись до його особистих характеристик, недоліків або вад. Це може бути використано для відволікання від фактів або справжньої суті обговорення.

Приклади:

1. "Я не вірю тобі щодо цієї проблеми, ти ж завжди робиш помилки",
2. "Цей вчений не вартий уваги, він не має навіть докторського ступеня",
3. "Я не підтримую твою позицію, ти ж ще молодий і не маєш досвіду".

Книги для лідерів команди:

1. "Herding Cats: A Primer for Programmers Who Lead Programmers" by Hank Rainwater,
2. "The Mythical Man-Month" by Fred Brooks,
3. "The Manager's Path: A Guide for Tech Leaders" by Camille Fournier,
4. "Clean Agile: Back to Basics" by Robert Martin,
5. "Scrum: The Art of Doing Twice the Work in Half the Time" by Jeff Sutherland,
6. "Learning Agile: Understanding Scrum, XP, Lean, and Kanban" by Andrew Stellman,
7. "Extreme Programming Explained" by Kent Beck.

Що таке Scrum?

Scrum (Скрам) та Extreme Programming (XP) ґрунтуються на Agile маніфесті.

Перша редакція Agile маніфесту була написана з 11 по 13 лютого 2001, на гірськолижному курорті в горах Юти.

Серед авторів маніфесту були: Кент Бек, Роберт Мартін, Мартін Фаулер, Ендрю Хант, Дейв Томас, Кен Швабер, Джейф Сазерленд.

Маніфест для Agile розробки програмного забезпечення:

1. Люди та співпраця важливіші за процеси та інструменти.
2. Працюючий продукт важливіший за вичерпну документацію.
3. Співпраця із замовником важливіша за обговорення умов контракту.
4. Готовність до змін важливіша за дотримання плану.

Гнучкі методики розробки (agile software development, Agile розробки) — узагальнюючий термін для цілого ряду підходів і практик, що ґрунтуються на цінностях Маніфесту гнучкої розробки програмного забезпечення та 12 принципах, що лежать у його основі.

Scrum — це легкий фреймворк, який допомагає людям, командам та організаціям створювати цінність шляхом адаптивних рішень для складних проблем.

Кен Швабер та Джейф Сазерленд вперше спільно представили Scrum на конференції OOPSLA у 1995 році.

П'ять цінностей Скраму: **Сміливість** — приймати виклики, **Зосередженість** — концентруватися на важливому, **Відданість (commitment)** — відповідальність за досягнення цілей, **Відкритість** — прозорість у спілкуванні, **Повага** — цінування внеску кожного члена команди.

Фреймворк — це каркас, або система принципів, яка встановлює певні рамки для роботи.

Ролі в Scrum:

- Власник продукту (Product Owner): Представляє клієнта та визначає, що потрібно будувати.
- Скрам-майстер (Scrum master): Налагоджує процес Scrum і допомагає команді подолати перешкоди.
- Команда розробників: Крос-функціональна команда, відповідальна за доставку продукту.

Найефективнішим і дієвим методом передачі інформації команді розробників і всередині неї є бесіда віч-на-віч.

Команди Scrum є крос-функціональними, що означає, що учасники мають всі необхідні навички для створення цінності кожного спринту.

Колаборативний характер крос-функціональної команди Scrum сприяє відчуттю спільноЙ відповідальності.

Замість того, щоб покладатися на одну особу для вирішення певного завдання чи області, команда спільно бере на себе власність і співпрацює, щоб вирішити будь-які перешкоди чи проблеми, що виникають.

Хоча крос-функціональні команди призначені мати різноманітний набір навичок, це не означає, що кожен учасник повинен бути експертом у всіх областях.

Крос-функціональні команди призначені для концентрації на одному проекті або продукті одночасно. Ефективна координація та комунікація все ще є ключовими в крос-функціональних командах.

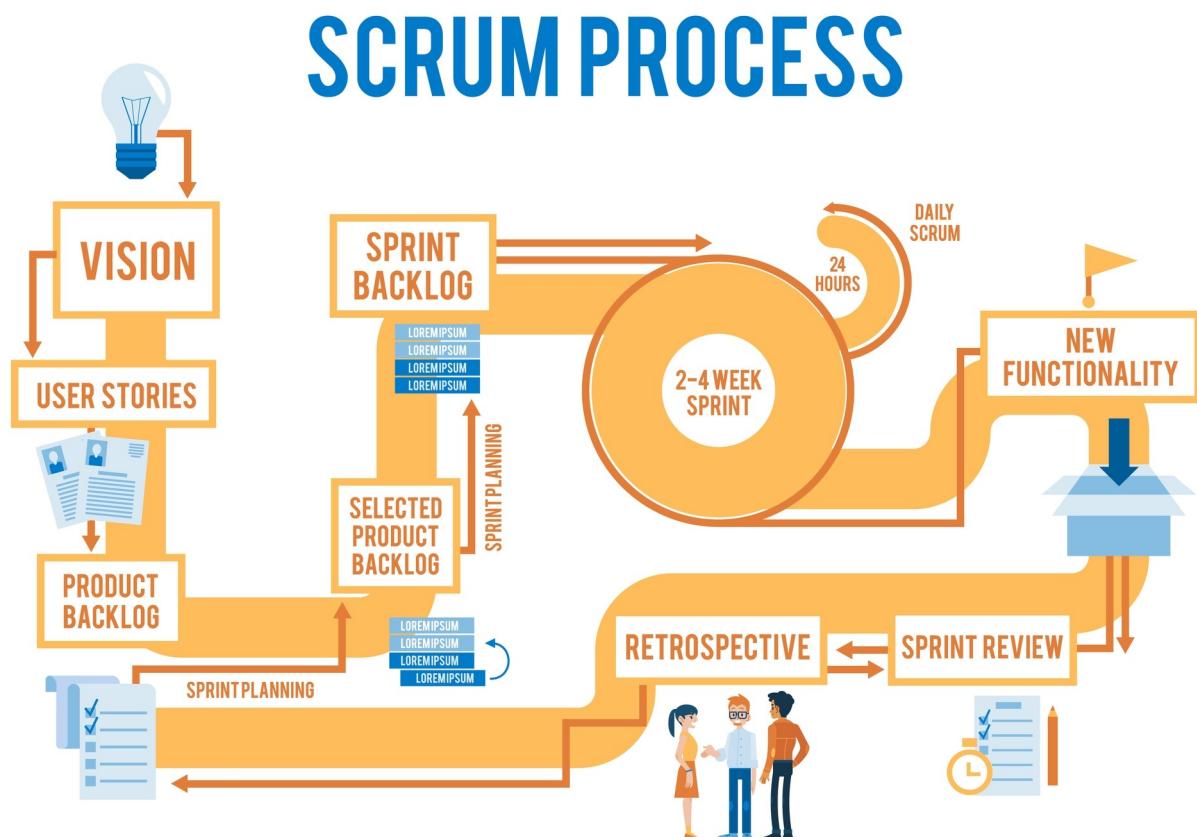
Команда Scrum складається з одного Scrum Master, одного Product Owner та розробників. У команді Scrum немає підкоманд чи ієрархій. Це згуртований підрозділ професіоналів, які зосереджені на одній меті за раз, цілі продукту.

Хоча формальна ієрархія відсутня, технічний лідер може виконувати роль старшого фахівця, який підтримує команду своїми знаннями та досвідом. Це допомагає забезпечити високий технічний рівень рішень.

Технічний лідер, який часто називається "Tech Lead", може існувати в Scrum команді для керування технічним боком проекту. Він відповідає за архітектуру та технічне бачення, допомагає команді у розв'язанні складних технічних проблем та забезпечує якість коду.

Scrum вимагає від Скрам-майстра створення середовища, де:

1. Власник продукту розміщує роботу для складної проблеми у Backlog продукту.
2. Команда Scrum перетворює відібрану роботу на приріст вартості (Інкремент цінності) під час спринту.
3. Команда Scrum та її зацікавлені сторони (stakeholders) оглядають результати та вносять корективи на наступний спринт.
4. Процес повторюється.



Під час Scrum спринту ідеї перетворюються на цінність.

Спринти — це фіксовані за часом події тривалістю один місяць або менше (2 тижні).

Новий спринт починається негайно після завершення попереднього.

Весь необхідний обсяг роботи для досягнення мети продукту, включаючи планування спринту, щоденні Scrum-зустрічі, огляд спринту та ретроспективу, відбуваються протягом спринтів.

Під час спринту:

- Не вносяться зміни, які б могли загрожувати меті спринту;
- Якість не знижується;
- Беклог (backlog) продукту уточнюється за потреби; і,
- Об'єм (scope) може бути уточнений та переглянутий з Власником продукту, коли стане відомо більше.

Спринт можна скасувати, якщо ціль спринту застаріє. Лише Власник продукту має право скасувати спринт.

Зазвичай проект у методології Scrum дотримується цих правил:

Кожен спринт починається з планування, проведеного Скрам-майстром, Власником продукту та рештою команди, і складається з наради (meeting), розділеної на дві частини, кожна з яких обмежена в часі чотирма годинами. Домашнє завдання Власника продукту перед спринт-плануванням — розробити пріоритетний беклог продукту, що складається з набору елементів (items), які користувачі та зацікавлені сторони погодили. У першій частині наради (зустрічі) Власник продукту співпрацює з командою, щоб вибрати елементи, які будуть доставлені до кінця спринту на основі їхньої цінності та оцінки роботи, яку вони вимагають. Команда згодна продемонструвати робоче програмне забезпечення, що включає ці елементи, до кінця спринту. Ця перша частина обмежена в часі (для спринту тривалістю 30 днів вона обмежена в часі чотирма годинами; для коротших спринтів вона пропорційно коротша), так що в кінці команда бере все, що вони вже зробили, і використовує це як беклог спринту. У другій частині наради члени команди (з допомогою Власника продукту) визначають індивідуальні завдання, які вони використовуватимуть для реальної реалізації цих елементів. Знову ж таки, ця частина обмежена в часі в залежності від тривалості спринту. На кінець планування спринту вибрані елементи стають беклогом спринту.

Команда проводить щоденну Scrum нараду (Daily Scrum meeting) щодня. Всі члени команди (включаючи Scrum-майстра та Власника Продукту) повинні брати участь, а зацікавлені сторони (stakeholders) також можуть брати участь (але повинні залишатися тихими спостерігачами). Нарада обмежена в часі 15 хвилин, тому всі члени команди повинні приходити вчасно. Кожен член команди відповідає на три питання: Що я зробив з моменту останньої щоденної наради? Що я збираюся зробити між цією та наступною щоденною нарадою? Які перешкоди та завади стоять на моєму шляху? Кожен член команди має бути лаконічним; якщо відповідь потребує обговорення, відповідні члени команди планують наступну нараду відразу після наради.

Кожний спринт обмежений в часі певною тривалістю, визначеною під час планування спринту: багато команд використовують 30 календарних днів, але ця тривалість може відрізнятися — деякі команди обирають двотижневі спринти. Протягом спринту команда перетворює елементи беклогу спринту в робоче програмне забезпечення. Вони можуть отримувати допомогу від людей, які не є членами команди, але такі люди не можуть диктувати команді, як виконувати їхню роботу, і повинні довіряти команді доставку. Якщо хтось з команди виявляє посередині спринту, що вони зобов'язалися занадто багато або що вони можуть ще додати елементи, вони повинні переконатися, що Власник продукту дізнається, як тільки вони розуміють, що спринт у небезпеці. Власник продукту — член команди, який може співпрацювати з користувачами та зацікавленими сторонами, щоб переглянути їх очікування, і використовувати цю інформацію для коригування беклогу спринту, щоб відповісти фактичній потужності команди. І якщо вони виявляють, що у них закінчиться робота до кінця спринту, вони можуть додати більше елементів до беклогу спринту. Команда повинна тримати беклогу спринту актуальним та видимим для всіх.

В дуже аномальних випадках та в екстремальних обставинах Власник продукту може припинити спринт раніше і почати нове планування спринту, якщо команда виявляє, що вони не можуть доставити робоче програмне забезпечення (наприклад, виникає серйозна технічна, організаційна або кадрова проблема). Але всі повинні знати, що припинення спринту є рідкісним явищем і має дуже негативні наслідки з точки зору їхньої здатності виробляти та доставляти програмне забезпечення.

Після завершення спринту команда проводить зустріч огляду спринту (sprint review meeting), де вони демонструють робоче програмне забезпечення користувачам та зацікавленим сторонам. Демонстрація може включати лише ті елементи, які фактично завершенні та протестовані (що в цьому випадку означає, що команда завершила всю роботу над ним і протестувала її, і що це було прийнято Власником продукту як завершене). Команда може представляти лише функціональне, робоче програмне забезпечення, а не проміжні елементи, такі як схеми архітектури, схеми баз даних, функціональні специфікації й т.д. Зацікавлені сторони можуть ставити питання, на які команда може відповісти. На кінець демонстрації зацікавлені сторони запитуються про свої думки та відгуки, і є можливість поділитися своїми думками, почуттями, ідеями та думками. Якщо потрібні зміни, це враховується при плануванні наступного спринту. Власник продукту може додати зміни до беклогу продукту, і якщо вони потрібні негайно, вони потраплять до беклогу наступного спринту.

Після спринту команда проводить зустріч ретроспективи спринту (sprint retrospective meeting), щоб знайти конкретні шляхи поліпшення своєї роботи. Участь беруть команда та Scrum-майстер (і, за бажанням, Власник продукту). Кожна людина відповідає на два питання: Що пішло добре під час спринту? Що може покращитися у майбутньому? Scrum-майстер фіксує будь-які покращення, і конкретні елементи (такі як налаштування нового сервера збирання, впровадження нової практики програмування чи зміна офісного розташування) додаються до беклогу продукту.

У методології Scrum є поширений інструмент, який називається "Scrum board" або "Kanban board". Це зазвичай цифрова дошка (або фізична дошка), розділена на колонки, що представляють різні етапи роботи, такі як "To Do", "In Progress" і "Done". Дошка допомагає візуалізувати прогрес завдань або користувацьких історій протягом спринту або проекту, що полегшує команді відстеження їхньої роботи та виявлення будь-яких заторів або проблем.

Ємність (Capacity) — це максимальна кількість роботи, яку команда може взяти на себе в наступному спринті. Команда може швидко встановити ємність, виходячи з середньої швидкості, а потім коригувати її враховуючи доступність кожного під час наступного спринту.

Навантаження (Load) — це кількість роботи, обрана командою для поточного спринту. Це кількість роботи, яку команда планує завершити протягом спринту. Воно повинно бути менше або рівним ємності.

Швидкість (Velocity) — це кількість роботи, завершена в попередніх спринтах. Це міра минулої продуктивності команди. Зазвичай дивляться на останні три-п'ять спринтів і беруть середнє значення їхньої швидкості.

Ось формули:

швидкість \geq ємність \geq навантаження,

буфер = ємність - навантаження

Різниця між ємністю та навантаженням — це ваш планувальний буфер.

- Беклог продукту: Пріоритетний список функцій та покращень.
 - Беклог спринту: Функції, обрані з Беклогу продукту для спринту.
 - Інкремент: Сума всіх завершених завдань в кінці спринту.
 - Спринт: Часовий блок (зазвичай 2-4 тижні) під час якого створюється потенційно готовий до відвантаження інкремент продукту.
 - Планування спринту: Зустріч для планування роботи на майбутній спрінт.
 - Щоденний Scrum (Daily Scrum): Коротка щоденна зустріч для синхронізації та планування роботи на день.
 - Огляд спринту (sprint review): Демонстрація завершеної роботи в кінці спринту.
 - Ретроспектива спринту: Аналіз минулого спринту для покращення процесів.
- Scrum підкреслює ітеративний та інкрементальний розвиток, сприяючи співпраці, гнучкості та постійному вдосконаленню.

Щоденний Scrum — це подія тривалістю 15 хвилин для розробників команди Scrum.

Огляд спринту — передостання подія спринту та обмежується до чотирьох годин для місячного спринту. Для коротших спрінктів подія, зазвичай, коротша.

Мета ретроспективи спринту — спланувати способи підвищення якості та ефективності.

Беклог продукту — це впорядкований список того, що потрібно для поліпшення продукту. Це єдине джерело роботи, яке виконується командою Scrum.

Скрам в основному зосереджений на організаційних процесах розробки програмного забезпечення, але Екстремальне програмування (Extreme Programming, XP) навпаки, акцентується на процесах кодування та тестування. XP було розроблено Кентом Беком. Воно підкреслює керовану тестами розробку (Test-Driven Development, TDD), парне програмування, спільну власність, постійну інтеграцію/постійне розгортання (Continuous Integration/Continuous Deployment, CI/CD) та невеликі ітерації.

Story Point (Сторі пойнт) та міфічна людино-година.

Коли ми вводимо поняття "людино-година", ми робимо припущення: що всі люди (припустимо, інженери ПЗ) мають рівні вміння, а отже можна чітко визначити час на виконання певної задачі. Це припущення вже невірне, коли в команді є старші та молодші спеціалісти, які роблять одне й те саме за різний час. Крім того, поняття "людино-година" говорить про те, що збільшення кількості людей буде пропорційно відображатися на зменшенні терміну створення продукту, але це міф. Цю помилку розбирав Фред Брукс у своїй книзі "Міфічний людино-місяць". Фредерік Брукс пише, що потрібно врахувати витрати на комунікацію між інженерами, які в деяких випадках можуть бути настільки великі, що виконується закон Брукса: "Додавання робочої сили до запізнілого програмного проекту затягує його ще більше". Додавання більшої кількості людей до завдання, яке дуже поділене, наприклад, прибирання номерів у готелі, зменшує загальну тривалість завдання (аж до моменту, коли додаткові працівники заважають один одному). Однак інші завдання, включаючи багато спеціальностей у проектах програмного забезпечення, менш подільні. Щоб вирішити ці недоліки поняття "людино-година", були створені story points в Scrum.

Story point (Очки складності завдання) - це одиниця виміру, яка використовується в методології розробки програмного забезпечення для оцінки складності задачі. Вона використовується для приблизної оцінки часу і зусиль, не прив'язуючись до конкретної часової рамки.

Коли команда розробників оцінює задачу, вони приймають у розгляд три основні фактори: складність, обсяг роботи та ризики. Вони потім призначають задачі певну кількість story points, яка відображає загальну складність. Чим більше story points, тим складніше завдання. Це дозволяє команді спрогнозувати, скільки робочих годин знадобиться на виконання задачі і планувати роботу відповідно.

Між story point та часом не має прямопропорційної залежності.

Nexus - фреймворк для мульти-командної розробки в рамках Scrum.

Коли продукт дуже великий його ділять на модулі, мікросервіси, мікрофронтенд. Кожен модуль та мікросервіс розробляє окрема команда. Але потрібно це все інтегрувати та керувати цим процесом.

Методологія Nexus є однією з підходів до масштабування розробки Agile, спрямованою на спільну роботу декількох Scrum-команд над великими проектами.

Основні правила методології Nexus включають:

-Роль Nexus Integration Team (NIT): Спеціалізована команда, яка відповідає за координацію роботи декількох Scrum-команд у межах Nexus.

-Nexus Sprint Planning: На цьому етапі *представники* всіх Scrum-команд розглядають інкременти, що плануються, і спільно визначають, які завдання потрібно виконати для досягнення цілей.

-Nexus Daily Scrum: Щоденні зустрічі для спілкування між членами команд та вирішення проблем, що можуть виникнути при інтеграції робочих інкрементів.

-Nexus Sprint Review: Під час цього заходу *представники* всіх команд демонструють свої робочі інкременти та обговорюють, які кроки слід вжити для подальшого вдосконалення.

-Nexus Sprint Retrospective: Регулярні огляди роботи всієї Nexus-структурі з метою виявлення та виправлення проблем та покращення робочих процесів.

-Nexus Goal: Кожен спринт має свою мету, яка визначається спільно всіма командами та допомагає забезпечити спільний фокус на досягнення цілей.

-Nexus Daily Scrum of Scrums (DoS): Це щоденна зустріч *представників* кожної Scrum-команди з метою обговорення інтеграційних питань та вирішення конфліктів.

-Nexus Sprint Backlog: Спільний перелік завдань для всіх Scrum-команд, який допомагає уникнути дублювання роботи та забезпечити спільне розуміння пріоритетів.

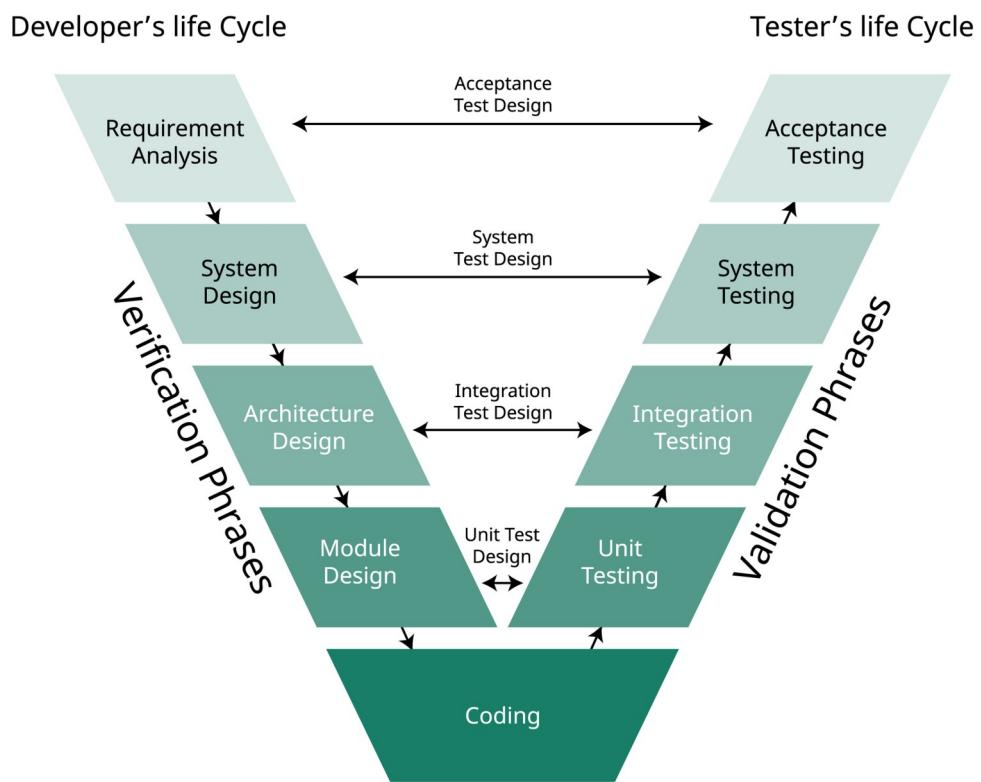
Ці правила допомагають забезпечити ефективну спільну роботу декількох команд у межах Nexus, спрощуючи співпрацю та забезпечуючи координацію.

Методологія Водоспад та V-модель

Методологія Водоспад (Waterfall) — це традиційний підхід до управління проектами, який використовується в розробці програмного забезпечення та інших галузях. Вона характеризується лінійним та послідовним процесом, де прогрес поступово рухається вниз (як водоспад) через попередньо визначені фази. Ось розбір типових етапів Водоспаду: Збір вимог, Проектування системи, Реалізація, Тестування, Розгортання, Обслуговування. Одна з ключових характеристик методології Водоспад — кожна фаза повинна бути завершена перед переходом до наступної. Це означає, що мало місця для гнучкості або змін після завершення фази, що може бути обмеженням в проектах, де вимоги ймовірно зміняться чи розвиватимуться з часом. Крім того, зворотний зв'язок від користувачів або зацікавлених сторін зазвичай приходить пізно в процесі, що може привести до дорогоцінного перероблення, якщо виникають непорозуміння або зміни в вимогах. Незважаючи на ці недоліки, методологія Водоспад може бути ефективною для проектів з чітко визначеними та стабільними вимогами, де важливі передбачуваність та контроль.

В моделі Waterfall тестування зазвичай відбувається після завершення всіх етапів розробки. V-модель розширяє модель Waterfall, додаючи тестування на кожному етапі розробки. Кожен етап розробки має відповідний етап тестування, утворюючи форму літери "V".

V-модель є концептуальною моделлю, що використовується в рамках методології Waterfall для візуалізації зв'язку між різними етапами проекту та відповідними тестами. На початку V-діаграми розташовані етапи, такі як збір вимог, аналіз, проектування системи та програмування, де визначаються вимоги та планується архітектура системи. Після цього настає фаза реалізації, де виконується написання коду. Після завершення фази реалізації, робота переходить до фази тестування, де виконуються тести для перевірки відповідності програмного забезпечення вимогам та специфікаціям. Всі ці етапи, починаючи з аналізу та завершуючи тестуванням, утворюють "прямий шлях" V-діаграми. У V-моделі тестування починається відразу після виконання кожного етапу розробки.



V-модель не відповідає Agile розробці.

Екстремальне програмування

Екстремальне програмування (Extreme Programming, XP) було розроблене в 1990-х роках Кентом Беком разом з іншими програмістами. XP є одним із методів гнучкої розробки програмного забезпечення, який акцентує на швидких ітераціях розробки, невеликих командних розмірах, тестуванні перед написанням коду та підтримці стандартів якості. (Book "Extreme Programming Explained" by Kent Beck, 1999)

Принципи Екстремального програмування (Extreme Programming, XP):

Планування:

- Користувачькі історії (user stories) написані.
- Планування випуску (Release planning) створює розклад випуску.
- Робіть часті та невеликі випуски.
- Проект розділений на ітерації.
- Планування ітерацій (Iteration planning) починається з кожної ітерації.

Управління:

- Зустріч “стендап” починається кожен день.
- Вимірюється швидкість проекту (Project Velocity).
- Наднормові вважаються поганою практикою.

Проектування:

- Простота (принцип KISS).
- Виберіть системну метафору, для загального опису системи.
- Створіть пробні рішення (spikes) для зменшення ризику.
- Жодна функціональність не додається заздалегідь (принцип YAGNI).
- Робіть рефакторинг, коли це можливо, де завгодно (але перед цим пишіть тести).

Кодування:

- Клієнт завжди доступний.
- Код повинен бути написаний відповідно до погоджених стандартів.
- Спочатку напишіть модульний тест (TDD).
- Усі виробничі коди програмуються в парі (pair programming).
- Тільки одна пара інтегрує код одночасно.
- Інтегруйте часто.
- Налаштуйте спеціальний інтеграційний комп'ютер (сервер). Автоматизація є основним принципом досягнення успіху DevOps, а CI/CD є критично важливим компонентом.
- Використовуйте колективну власність (collective ownership). За код відповідає вся команда.

Тестування:

- Весь код повинен мати модульні тести.
- Весь код повинен пройти всі модульні тести, перш ніж його можна буде випустити.
- Якщо знайдено помилку, створюються тести.
- Приймальні тести (Acceptance tests) часто запускаються, і результати публікуються.

Простота: Ми будемо робити те, що потрібно та вимагається, але не більше. Це максимізує створену вартість від вкладених до цього дня інвестицій.

Комуникація: Кожен є частиною команди, і ми щоденно спілкуємося. Ми працюватимемо разом над усім, починаючи від вимог до коду.

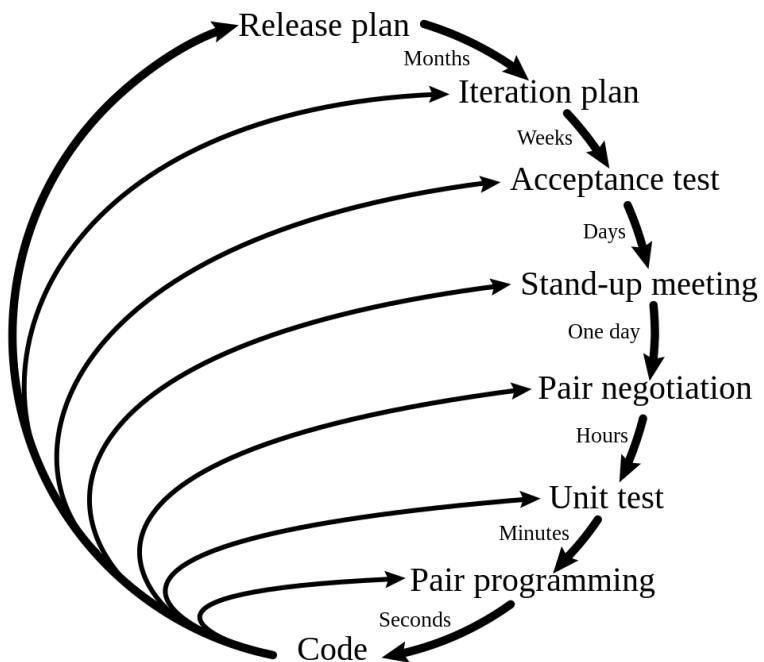
Зворотний зв'язок: Ми серйозно ставимось до кожної ітераційної зобов'язаності, поставляючи робоче програмне забезпечення. Ми демонструємо наше програмне забезпечення часто та рано, потім уважно слухаємо та вносимо будь-які зміни, які необхідні.

Повага: Кожен отримує та відчуває повагу, яку він заслуговує як цінний член команди. Кожен вносить свою вартість, навіть якщо це просто ентузіазм. Розробники поважають експертизу клієнтів, і навпаки. Керівництво поважає наше право приймати відповідальність та отримувати повноваження над своєю роботою.

Мужність: Ми говоритимемо правду про прогрес та оцінки. Ми не документуємо виправдання для невдачі, оскільки ми плануємо успіх.

Метафора системи — це простий і послідовний спосіб опису вашої програмної системи за допомогою знайомої області або аналогії. Це допомагає вам і вашій команді спілкуватися та розуміти дизайн, архітектуру та функціональність вашої системи. Наприклад, якщо ви створюєте веб-бібліотечну систему, ви можете використати метафору фізичної бібліотеки з книгами, полицями, каталогами, позиками та поверненнями. Ця метафора може допомогти вам узгоджено називати ваші класи, методи, змінні та інтерфейси.

Planning/feedback loops



Спайк (spike) - це інновація в екстремальному програмуванні (XP) в гнучкій (agile) розробці програмного забезпечення. Це невелика історія (user story) або робота, яка призначена для збору інформації, а не для створення інкременту в продукті.

Слово "спайк" походить від скелелазних занять. Під час сходження ми можемо зупинитися, щоб вбити спайк (цвях) в скелі, що не є фактичним сходженням, але цим ми забезпечуємо, що майбутнє сходження буде легким і простим.

Так само під час розробки команда проводить невеликий експеримент або дослідження та створює доказ концепту (POC, Proof of Concept), який не є фактичною розробкою чи виробничими завданнями, але спрощує майбутній розвиток.

Історію (user story) не можна оцінити, поки розробницька команда не проведе дослідження обмежене по часу (не більше часу однієї ітерації). Тому спайкам не призначаються бали історії (story points), оскільки вони не пов'язані з інкрементом продукту і, отже, не сприяють швидкості команди.

Спайки обмежені в часі.

Визначення завершеності (Definition of Done) та контроль якості (QA) в Екстремальному програмуванні (XP).

Приймальні тести (Acceptance tests) створюються під час аналізу вимог і перед кодуванням. В Екстремальному програмуванні (XP) приймальні тести зазвичай також розробляються відповідно до методології розробки через тести (TDD).

Історія користувача (User story) не вважається завершеною, поки вона не пройде свої приймальні тести (Acceptance tests). Це означає, що нові приймальні тести повинні бути створені кожної ітерації, або команда розробників буде повідомляти про нульовий прогрес.

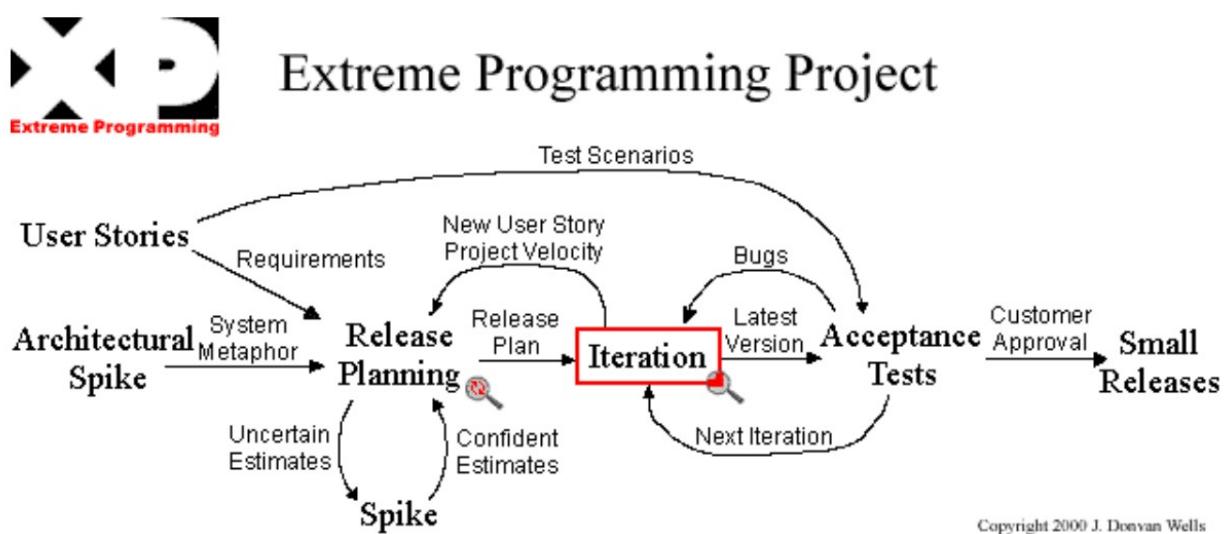
Контроль якості (QA) є важливою частиною процесу XP (Extreme programming).

Екстремальне програмування вимагає взаємодії розробників із QA спеціалістами.

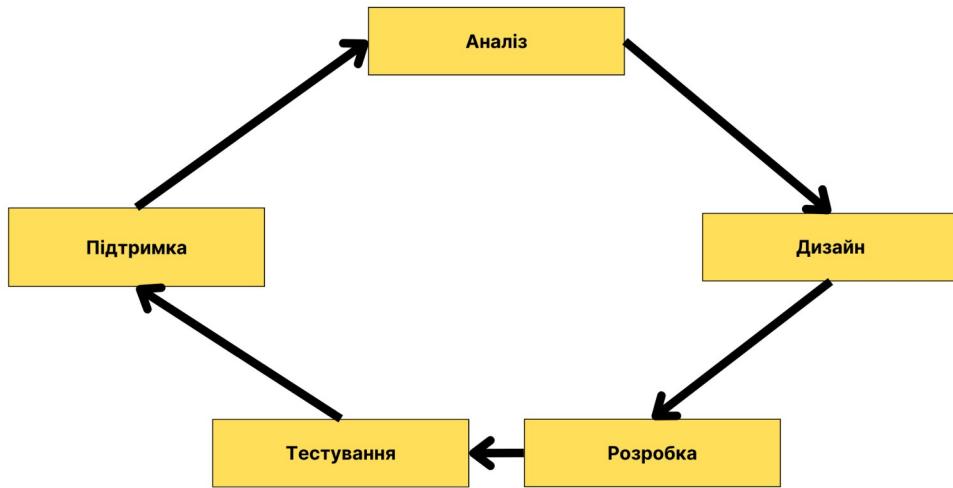
Приймальні тести повинні бути автоматизовані, щоб їх можна було часто запускати (CI/CD).

Результати приймальних тестів публікуються команді. Це відповідальність команди запланувати час кожної ітерації для виправлення будь-яких неуспішних тестів.

Екстремальне програмування базується на TDD, автоматичних тестах.



Цикл розробки програмного забезпечення



Цикл розробки програмного забезпечення (ПЗ) включає такі етапи:

1. Збір та аналіз вимог,
2. Проектування системи (дизайн),
3. Розробка (програмування),
4. Тестування,
5. Розгортання (deploy),
6. Підтримка та покращення системи (програми).

Кожен етап важливий для успішного завершення проєкту.

Зокрема, він описаний у SWEBOK від IEEE.

Тестування — процес застосування чітких (кількісних) методів для отримання інформації про об'єкт (систему), зокрема, щоб виявити його придатність до використання.

Тобто в тестуванні є чіткий метод, ціль, висновок. Застосовують дерево прийняття рішень, повну індукцію, дедукцію. Адекватність методу тестування встановлюється на основі прагматизму, коректності його результатів.

Що таке Неперервна інтеграція (CI/CD)?

Неперервна інтеграція (Continuous Integration, CI) - це практика в розробці програмного забезпечення, коли код розробляється інкрементально та регулярно інтегрується в спільний репозиторій. Це допомагає виявляти конфлікти та помилки якомога раніше в розробці, забезпечуючи стабільніше та надійніше програмне забезпечення.

Неперервна інтеграція — це практика, а не інструмент. Її чітко описав й використовував Кент Бек в 1999 році.

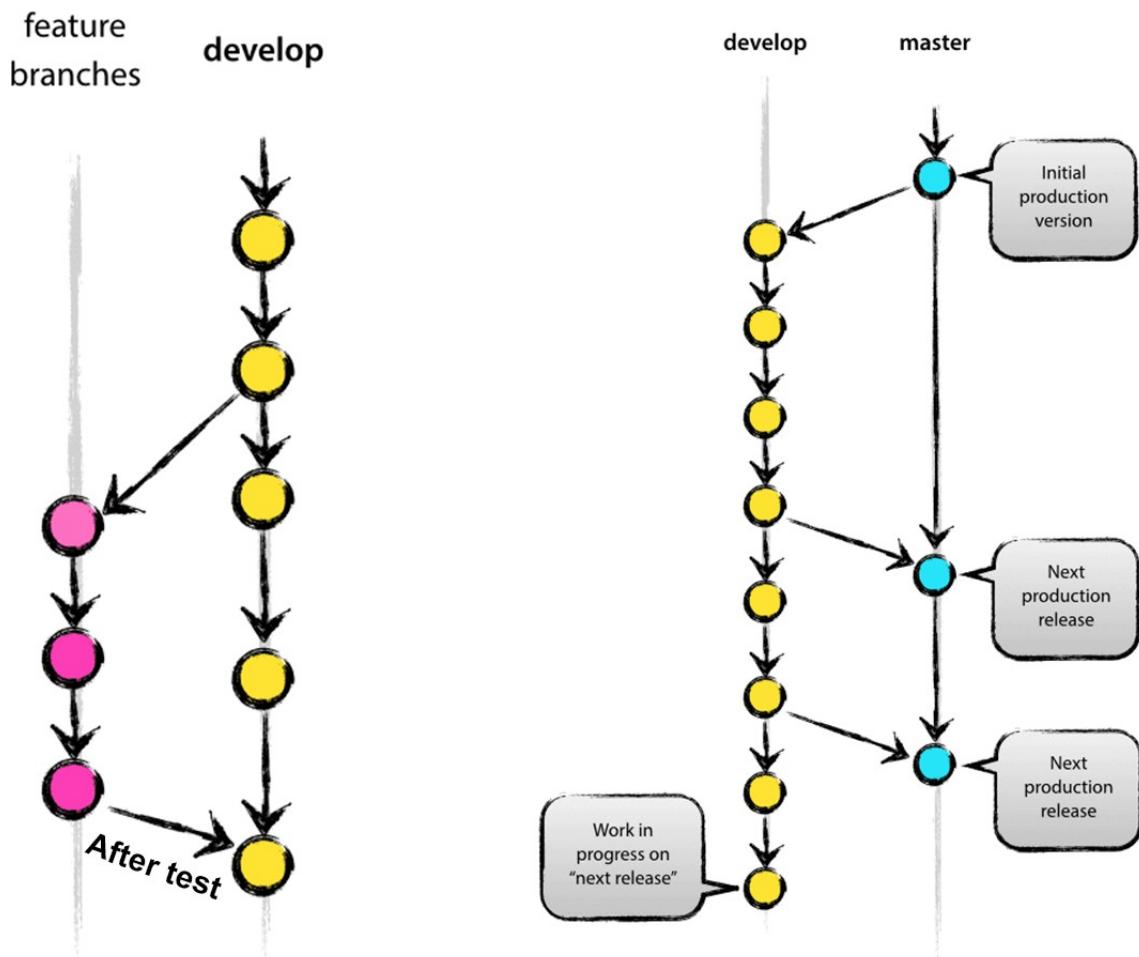
Уявимо, що існує єдина головна база (сховище) початкового коду проєкту (головна гілка репозиторію коду).

Ця база коду призначена для кінцевого користувача, тобто саме вона буде в фінальному продукті. В команді працюють декілька розробників. Кожен з них копіює собі цю базу коду (або її частину) та вносить в неї зміни відповідно до свого завдання. Припустимо, злиття всіх баз коду в основну (main, master) відбувається під час нового випуску продукту (product release). З таким підходом одразу виникають декілька проблем. Наприклад, синхронізація гілок між серверною логікою (backend) та інтерфейсом (frontend) під час розробки та тестування.

Основна проблема в тому, що програміст перед релізом (випуском продукту) може зробити декілька задач, і задачі розробників на практиці часто перетинаються, тобто два різних розробники можуть змінити одну й ту ж частину коду під час розробки нової версії. Проблема в тому, що ці конфлікти виникнуть аж перед випуском релізу. Розробники не могли їх синхронізувати, бо працюють в різних гілках. Відповідно, потрібно робити Continuous integration, тобто неперервну інтеграцію змін в одну кодову базу, яка потім буде об'єднуватися з основною гілкою (сховищем коду).

Ми створюємо копію основного кодової бази під назвою develop. Будемо називати її гілкою develop. Ця гілка містить копію основної гілки (master, main).

Коли програміст починає розробляти щось, він створює нову гілку від гілки develop, а після завершення задачі інтегрує гілку задачі в гілку develop. Коли він починає нову задачу, він знову робить гілку від develop та повторює процес. В кінці гілка develop буде "залита" в основну гілку master. Конфлікти в коді будуть розв'язуватися після завершення задачі, а не перед випуском продукту. Тестувальник (QA/QC) тестує саме гілку develop, тобто інтегровані зміни.



Author: Vincent Driessen. Original blog post: <http://nvie.com/posts/a-successful-git-branching-model>
 License: Creative Commons BY-SA.

Гілка в системах контролю версій, таких як Git, являє собою копію основної лінії розвитку проекту. Кожна гілка може мати свою власну історію змін, що дозволяє розробникам працювати паралельно над різними функціями чи виправленнями помилок, не впливаючи на основний код. Гілки дозволяють вам експериментувати з новими функціями, розгалужувати роботу на різних напрямках, а потім об'єднувати зміни з різних гілок у основну лінію розвитку за допомогою операції злиття (merge). Наприклад, якщо у вас є основна гілка (зазвичай називається "main" чи "master"), ви можете створити нову гілку для розробки певної функції. Після того, як розробка закінчиться і функція буде готова, ви можете злити цю гілку з основною, щоб включити зміни у основний код. Гілки сприяють організації роботи над проектом, спрощують співпрацю між розробниками та дозволяють краще контролювати розвиток програмного забезпечення.

Continuous Delivery (CD) - це методологія розробки програмного забезпечення, яка забезпечує автоматизовану, повторювану доставку програмного забезпечення у виробниче середовище. Головна мета - максимально швидко та безпечно впроваджувати зміни у коді та відразу ж перевіряти їх працездатність.

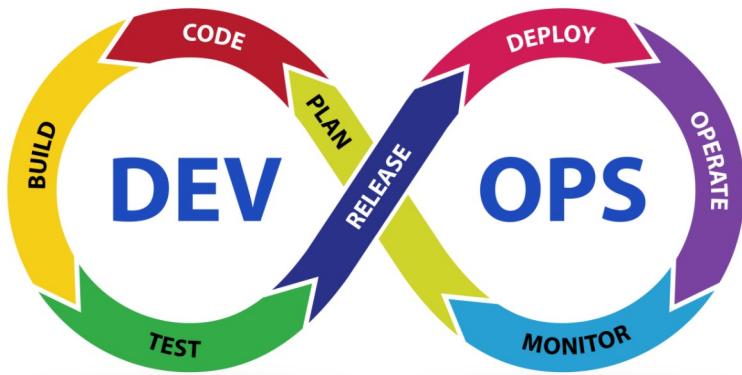
Це збільшує співпрацю між розробниками, тестувальниками та замовниками завдяки постійному циклу зворотного зв'язку.

Скільки часу знадобиться вашій організації, щоб розгорнути зміни (для користувачів), які включають лише один рядок коду?

Конвеєр розгортання (deployment pipeline): **Додайте код > автоматичне модульне (unit) тестування > автоматичне приймальне тестування (acceptance testing) > ручне тестування змін та ручне дослідницьке (exploratory) тестування > випуск (release).**

Один з прикладів Continuous Delivery може бути розробка веб-додатку, де кожен коміт (внесок) коду автоматично пройде через серію тестів (функціональні, модульні, інтеграційні тощо) і після успішного проходження цих тестів буде автоматично розгорнуто на тестовому сервері для перевірки. Якщо тестування пройде успішно, програмний код буде автоматично розгорнуто в продакшн. Таким чином, будь-які зміни в коді швидко та надійно доставляються до кінцевих користувачів.

Однак, перш ніж випустити його в продакшн (виробництво), зазвичай розробники використовують тестові сервери для перевірки працездатності та відповідності функціональним вимогам. Тому після успішного проходження тестів код може спочатку розгорнатися на тестовому сервері для фінальних перевірок перед випуском у продакшн.



З безперервною доставкою:

1. Кожен член команди не повинен безперервно завантажувати всі гілки коду та програми собі.
2. Власник продукту та користувачі можуть безперервно оцінювати програму.
3. QA-фахівець тестує інтегровану та розгорнуту задачу на сервері, а не піклується про версії коду чи про його майбутні інтеграції.

Приклад простого процесу розгортання:

1. Напишіть код і додайте для нього покриття тестами понад 80% (модульні, інтеграційні, системні (і приймальні) тести). Автоматизовані приймальні тести може писати QA спеціаліст, а не сам розробник.
2. Об'єднайте його до гілки “develop” сховища коду (репозиторія коду).
3. Потім об'єднайте гілку “develop” із гілкою “test”. (Зазвичай, один раз на день). Спеціаліст із забезпечення якості (QA) перевіряє функцію на гілці “test”, яка автоматично розгортається на сервері.
4. Повторіть процес.
5. Якщо вам потрібна продакшн версія, ви створюєте нову гілку “release x.y.z” з гілки “test” і нарешті випускаєте її.
6. Повторіть процес.

Цей процес оснований на автоматичному тестуванні, яке позбавляє постійного регресивного тестування та робить процес злиття гілок достатньо безпечним. Якщо у вас немає покриття тестами, тоді складність та час мануального (ручного) тестування буде значно більший і складніший.

Якщо код не покритий автоматичними тестами, процес залишається таким самим, але розробники вручну перевіряють чи їхні зміни не зламають гілку develop. Тестувальник натомість тестує гілку test в тестовому середовищі.

Деталі:

<https://nvie.com/posts/a-successful-git-branching-model/> (GitFlow),
<http://www.extremeprogramming.org/rules/integrateoften.html>,

- Jez Humble, David Farley, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" (Addison-Wesley, 2010),
- Paul M. Duvall, Steve Matyas, Andrew Glover, "Continuous Integration: Improving Software Quality and Reducing Risk" (Addison-Wesley, 2010).

Які є структури даних?

Існує багато різних структур даних в програмуванні, і кожна з них має свої особливості та застосування. Ось декілька основних структур даних:

1. **Масив (Array):** Масив (кортеж) — це упорядкована колекція елементів, які зберігаються під послідовними індексами. Вони дозволяють ефективно звертатися до елементів за їхніми індексами.

Масив в мові JavaScript:

```
const numbers = [1,2,3,4, 5];  
console.log(numbers[0]); //1  
console.log(numbers[3]); //4
```

2. **Стек (Stack):** Стек — це структура даних, яка працює за принципом "останній ввійшов, перший вийшов" (Last-In, First-Out, LIFO). Видаляючи елемент із стеку, ви отримуєте останній доданий елемент.

Стек схожий на стопку книг, де ви кладете нову книгу на верхню частину та берете зверху ту, яку востаннє поклали.

Дві головні дії зі стеком — додати (кладемо щось наверх) та взяти (беремо верхню книгу). Стек використовується для відстеження дій або стану, де останнє діяло перше скасовується.

3. **Черга (Queue):** Черга — це структура даних, яка працює за принципом "перший ввійшов, перший вийшов" (First-In, First-Out, FIFO). Видаляючи елемент із черги, ви отримуєте перший доданий елемент.

Черга схожа на чергу в магазині — перший, хто вступив, першим і обслуговується. Черга використовується в ситуаціях, коли обробка елементів повинна бути у визначеному порядку, як участь в черзі.

4. **Граф (Graph):** Граф — це структура даних, яка складається з вершин та ребер, які з'єднують ці вершини. Графи використовуються для моделювання складних взаємозв'язків між об'єктами.

5. **Дерево (Tree):** Дерево — це ієрархічна структура даних, певний вид графу, де кожен елемент має багато дітей. Дерева використовуються, наприклад, для організації даних в базах даних та для реалізації алгоритмів, таких як дерева пошуку. Як стовбур дерева має гілки, і кожна гілка може розгалужуватися на інші гілки.

6. **Хеш-таблиця (Hash Table):** Хеш-таблиця — це структура даних, яка використовує хеш-функцію для збереження ключів та відповідних значень. Вона надає швидкий доступ до даних за ключем.

7. Множина (Set): Множина — це структура даних, яка містить унікальні елементи. Вона використовується для виконання операцій об'єднання, перетину та різниці між множинами. Множина $\{1, 2, 1\} = \{1, 1, 2\} = \{1, 2\}$.

Хеш-таблиці

Хеш-таблиці — це тип структури даних, у якій значення адреси/індексу елемента даних генерується за допомогою хеш-функції. Це забезпечує дуже швидкий доступ до даних, оскільки значення індексу поводиться як ключ для значення даних.

Іншими словами, хеш-таблиці зберігають пари ключ-значення, але ключ генерується за допомогою функції хешування. Таким чином, функція пошуку та вставки елемента даних стає набагато швидшою, оскільки ключові значення самі стають індексом масиву, який зберігає дані. Під час пошуку ключ хешується, а отриманий хеш вказує, де зберігається відповідне значення.

```
var arr = [];
var x = 5;
arr[hash(x)] = x;
arr[hash(x)] + 1 = 6 // true
```

Ідея хешування полягає в тому, щоб розподілити записи (пари ключ/значення) між масивом комірок.

За наявності ключа алгоритм обчислює індекс, який вказує, де можна знайти запис:

індекс = $f(\text{ключ}, \text{розмір_масиву})$

Часто це робиться у два етапи:

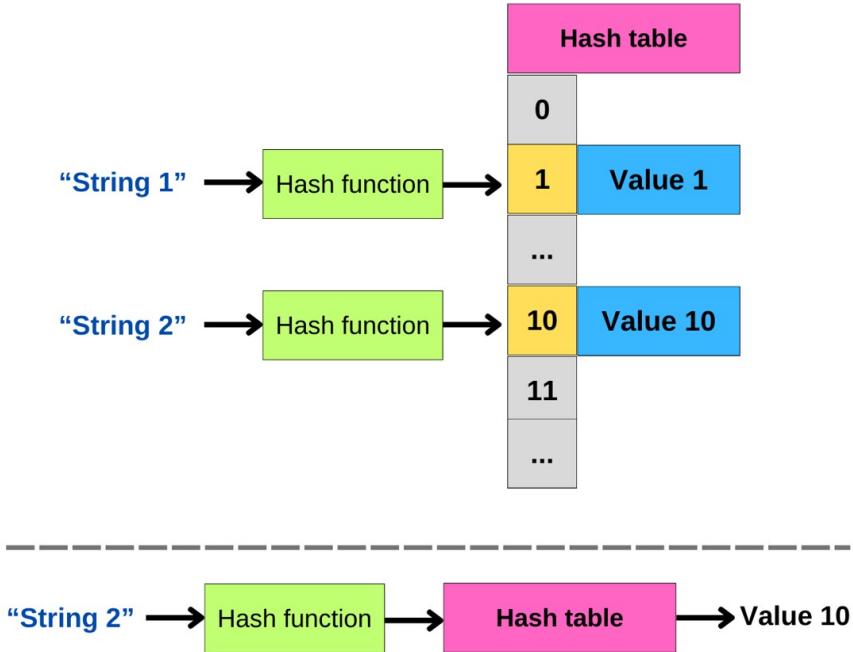
хеш = hashfunc(ключ)

індекс = хеш % array_size.

У добре продуманій хеш-таблиці середня вартість (кількість інструкцій) для кожного пошуку не залежить від кількості елементів, що зберігаються в таблиці. Складність алгоритму отримання елементу з хеш-таблиці $O(1)$ – константа. Багато дизайнів хеш-таблиць також дозволяють довільні вставки та видалення пар ключ-значення за постійної середньої вартості операції.

У багатьох ситуаціях хеш-таблиці виявляються більш ефективними, ніж дерева пошуку чи будь-яка інша структура пошуку таблиць. З цієї причини вони широко використовуються в багатьох видах комп’ютерного програмного забезпечення, зокрема для асоціативних масивів, індексування баз даних, кешу і наборів.

SHA-2 — це сімейство криптографічних функцій, які перетворюють вхідні дані у коротке, унікальне значення, яке використовується для перевірки цілісності даних та безпеки в інтернеті. Ці функції забезпечують надійний захист інформації, дозволяючи перевірити, чи не змінювалися дані під час передачі або зберігання.



Хеш-функція djb2 мовою TypeScript:

```
function djb2Hash(str: string) {
    let hash = 5381;
    for (let i = 0; i < str.length; i++) {
        hash = ((hash << 5) + hash) + str.charCodeAt(i);
    }
    return hash >>> 0; // Ensure the hash is a 32-bit unsigned integer
}
```

Функція djb2Hash приймає рядок str як вхідні дані.

Вона ініціалізує змінну hash значенням 5381, простим числом, обраним як початкове хеш-значення.

Потім вона проходить по кожному символу у вхідному рядку за допомогою циклу for.

У циклі оновлюється хеш-значення за допомогою формули DJB2: $hash = ((hash \ll 5) + hash) + str.charCodeAt(i)$.

На кінці переконуємося, що хеш-значення є 32-бітним беззнаковим цілим числом за допомогою беззнакового зсуву вправо ($>>> 0$).

Функція DJB2 не призначена для криптографічних цілей. Вона може бути достатньо міцною для застосувань, де потрібен простий хеш для швидкого доступу або для невеликих об'ємів даних. Проте, якщо безпека є важливим аспектом, краще використовувати криптографічно стійкі хеш-функції, такі як SHA-256.

Які існують алгоритми сортування масивів?

Алгоритми сортування — це способи організації даних у впорядковану послідовність. Кожен з них має свої унікальні особливості та ефективність в різних сценаріях.

Bubble Sort (сортування бульбашкою): Це простий алгоритм, де порівнюються сусідні елементи та обмінюються, якщо вони розташовані у неправильному порядку. Він проходить через список кілька разів, переміщуючи найбільший (або найменший) елемент на його правильне місце.

Insertion Sort (сортування включенням): Цей метод розглядає невідсортовану частину списку та поступово додає кожен елемент у відсортовану частину, вставляючи його на відповідне місце.

Selection Sort (сортування вибором): Він розділяє список на відсортовану та несортovanу частини. Він шукає найменший елемент у несортованій частині та обмінює його з першим елементом у несортованій частині.

Quicksort (швидке сортування): Цей алгоритм базується на стратегії "розділяй і володарюй". Він вибирає елемент, який називається опорним, та розбиває масив на під списки навколо цього елемента. Після цього він рекурсивно сортує кожен під список.

Ці алгоритми мають різну ефективність в залежності від обсягу даних та вже впорядкованості списку. Наприклад, Bubble Sort, хоча простий для реалізації, неефективний для великих списків, тоді як Quicksort може бути швидким, але вимагає більше ресурсів та пам'яті.

Ефективність алгоритмів сортування визначається їх часом виконання, який може змінюватися в залежності від обсягу даних, типу даних та вже впорядкованості списку.

Bubble Sort має часову складність в середньому $O(n^2)$, де n - кількість елементів у списку. Цей алгоритм може бути неефективним, особливо для великих списків, оскільки вимагає багато порівнянь та обмінів.

Insertion Sort також має часову складність $O(n^2)$ у середньому, але він може працювати краще для маленьких списків або вже впорядкованих даних.

Selection Sort також має часову складність $O(n^2)$ у середньому. Він схожий на Bubble Sort за ефективністю, але може бути трохи ефективнішим через меншу кількість обмінів.

Quick Sort має середню часову складність $O(n * \log(n))$, що робить його швидким для великих списків. Однак, у найгіршому випадку, коли погано выбраний опорний елемент, часова складність може бути $O(n^2)$, що робить його менш ефективним у цьому випадку.

Загальний висновок: для великих списків Quick Sort зазвичай виявляється найефективнішим за часом виконання, тоді як для менших списків Insertion Sort або Selection Sort можуть бути прийнятними через їх простоту. Bubble Sort, у більшості випадків, не є оптимальним вибором через свою високу складність в порівнянні з іншими алгоритмами.

Бульбашкове сортування

Бульбашкове сортування використовується, коли потрібно відсортувати масив даних. Хоча часова складність алгоритму бульбашкового сортування досить велика, а саме, він працює за квадратний час, тобто n^2 , де n — кількість даних в масиві, проте, він дуже простий для опису і не використовує багато пам'яті комп'ютера.

Суть алгоритму така:

Припустимо, надано масив з N натуральних чисел, які потрібно відсортувати в натуральному порядку, тобто в порядку зростання.

Щоб відсортувати цей масив, потрібно виконати такі операції:

1. Візьмемо перший елемент масиву і другий.
2. Порівняйте перший і другий елемент, якщо перший елемент більший за другий, то його потрібно поставити на місце другого, а другий на місце першого.
3. Зробіть те ж саме, тобто крок 1, для другого і третього елементу, потім третього і четвертого, і продовжуйте робити це, поки масив не буде впорядковано.

Наприклад

$N = [5, 1, 2, 3, 6, 4, 8, 9, 7]$;

$5 > 1$ тоді $N = [1, 5, 2, 3, 6, 4, 8, 9, 7]$;

$5 > 2$ тоді $N = [1, 2, 5, 3, 6, 4, 8, 9, 7]$;

$5 > 3$ тоді $N = [1, 2, 3, 5, 6, 4, 8, 9, 7]$;

$5 < 6$ тоді $N = [1, 2, 3, 5, 6, 4, 8, 9, 7]$;

$6 > 4$ тоді $N = [1, 2, 3, 5, 4, 6, 8, 9, 7]$;

$6 < 8$ тоді $N = [1, 2, 3, 5, 4, 6, 8, 9, 7]$;

$8 < 9$ тоді $N = [1, 2, 3, 5, 4, 6, 8, 9, 7]$;

$9 > 7$ тоді $N = [1, 2, 3, 5, 4, 6, 8, 7, 9]$;

$1 < 2$ тоді $N = [1, 2, 3, 5, 4, 6, 8, 7, 9]$;

$2 < 3$ тоді $N = [1, 2, 3, 5, 4, 6, 8, 7, 9]$;

$3 < 5$ тоді $N = [1, 2, 3, 5, 4, 6, 8, 7, 9]$;

$5 > 4$ тоді $N = [1, 2, 3, 4, 5, 6, 8, 7, 9]$;

$5 < 6$ тоді $N = [1, 2, 3, 4, 5, 6, 8, 7, 9]$;

$6 < 8$ тоді $N = [1, 2, 3, 4, 5, 6, 8, 7, 9]$;

$8 > 7$ тоді $N = [1, 2, 3, 4, 5, 6, \mathbf{7}, \mathbf{8}, 9]$;

$8 < 9$ тоді $N = [1, 2, 3, 4, 5, 6, 7, \mathbf{8}, \mathbf{9}]$;

Перевірте масив, якщо він впорядкований, тоді сортування зупиняється. Це означає, що коли ще один набір операцій не змінив масив, то його впорядковано і ми можемо зупинитися. Американський програміст Дональд Кнут описує “Бульбашкове сортування” в книзі "Мистецтво програмування" (1973).

JavaScript код Бульбашкового сортування (Bubble sort):

```
function bubbleSort(arr) {  
    const n = arr.length;  
  
    for (let i = 0; i < n - 1; i++) {  
        // Останні i елементи вже відсортовані, тому нам не потрібно їх перевіряти  
        for (let j = 0; j < n - i - 1; j++) {  
            // Поміняти місцями, якщо знайдений елемент більший за наступний  
            if (arr[j] > arr[j + 1]) {  
                // Поміняти місцями arr[j] і arr[j+1]  
                const temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
  
    return arr;  
}  
  
// Приклад використання:  
const unsortedArray = [64, 34, 25, 12, 22, 11, 90];  
const sortedArray = bubbleSort(unsortedArray);  
  
console.log("Sorted array:", sortedArray);
```

Сортувати вибором

Сортування вибором — це алгоритм сортування.

Дії алгоритму:

1. знайти номер мінімального значення в поточному списку;
2. ми міняємо це значення на значення першої невідсортованої позиції (обмін не потрібен, якщо мінімальний елемент вже знаходиться на цій позиції);
3. тепер сортуємо хвіст списку, виключаючи з розгляду вже відсортовані елементи;

Початковий масив	Найменший елемент	Відсортований масив
11, 25, 12, 22, 64	11	11

25, 12, 22, 64	12	11, 12
25, 22, 64	22	11, 12, 22
25, 64	25	11, 12, 22, 25
64	64	11, 12, 22, 25, 64

Сортування включенням

Сортування включенням — це алгоритм сортування.

Скажімо, у нас є масив чисел, які ми хочемо відсортувати за зростанням (природним) порядком. Щоб зробити це за допомогою методу сортування вставкою, потрібно виконати такі дії:

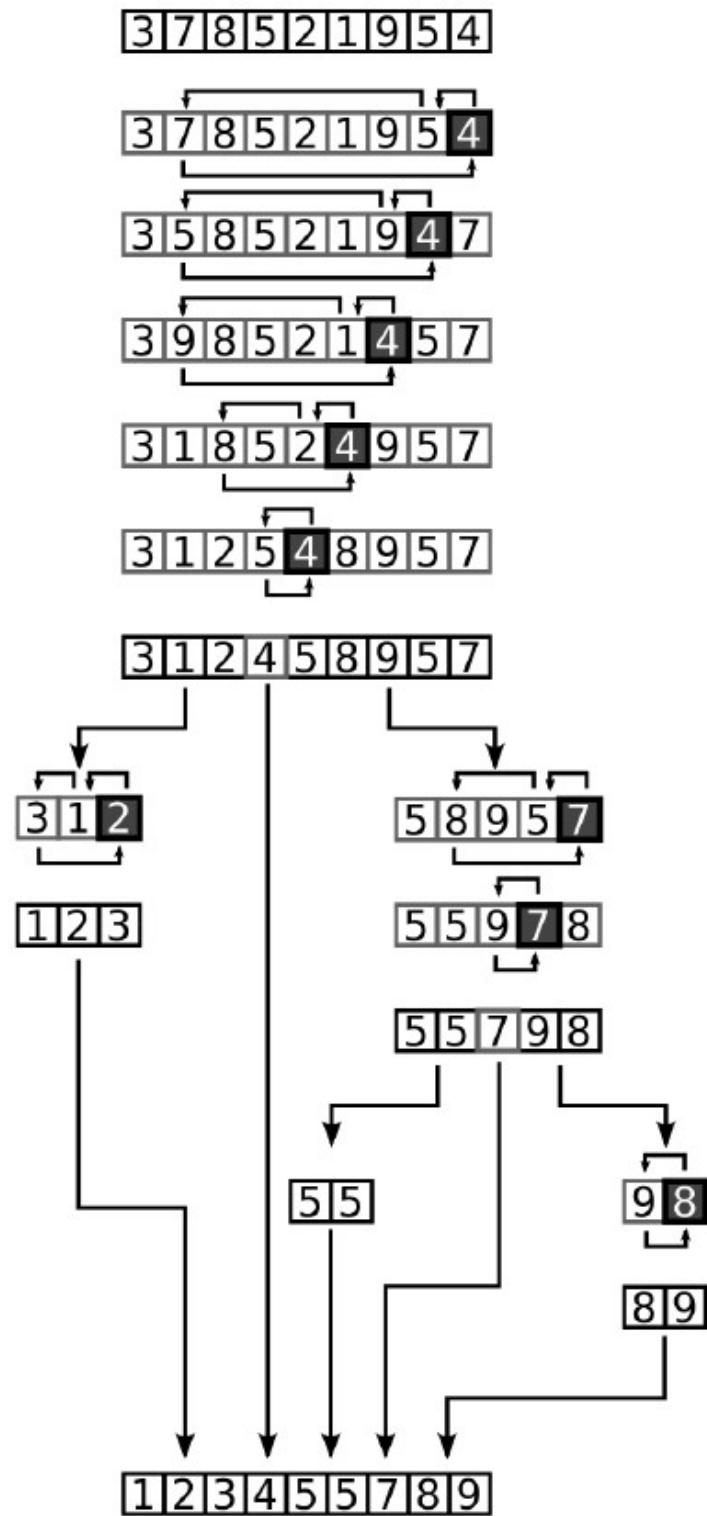
1. Беремо другий елемент масиву і порівнюємо його з першим, якщо другий менше першого, то міняємо місцями.
2. Беремо третій елемент масиву і порівнюємо його з першим і другим, якщо він менше другого, але більше першого, міняємо місцями з другим, якщо не перший і другий, ставимо це на місці першого.
3. Візьміть четвертий елемент масиву і порівняйте його з першим, другим і третім елементом. Якщо менше третього, але більше другого, міняємо місцями з третьим, якщо більше третього і другого, але менше першого, то ставимо його на місце другого. Якщо він менше всіх попередніх елементів, ставимо його на місце першого.
4. Аналогічно виконуємо крок 3 для всіх елементів масиву.

Швидке сортування

Quicksort — це алгоритм сортування, який базується на принципі “розділяй і володарюй” (Розділіть велике завдання на підзадачі).

Алгоритм складається з трьох кроків:

1. Виберіть елемент із масиву. Назовемо це опорою.
2. Перерозподіл елементів у масиві таким чином, що елементи, менші за опорну, розміщуються перед нею, а більші або рівні після.
3. Рекурсивно застосуйте перші два кроки до двох підмасивів ліворуч і праворуч від опори. Рекурсія не застосовується до масиву лише з одним елементом або відсутніми елементами.



Сортування підрахунком

Сортування підрахунком (Counting sort) — алгоритм впорядкування, що застосовується для масивів з числами, в яких найбільше число не сильно більше ніж розмір самого масиву. Якщо у вас масив [1, 2, 100] алгоритм не буде ефективним, бо найбільше число 100 значно більше ніж розмір масиву 3. Але якщо масив має 10000 чисел, і жодне число не більше, наприклад, 1000, тоді він буде дуже ефективним. Цей алгоритм вираховує індекс числа в відсортованому масиві на основі його значення, без порівняння самих чисел між собою.

```
function countingSort(arr) {  
    const max = Math.max(...arr);  
    const min = Math.min(...arr);  
  
    const countArray = new Array(max - min + 1).fill(0);  
  
    for (let i = 0; i < arr.length; i++) {  
        countArray[arr[i] - min]++;  
    }  
  
    let outputIndex = 0;  
    for (let i = 0; i < countArray.length; i++) {  
        while (countArray[i] > 0) {  
            arr[outputIndex++] = i + min;  
            countArray[i]--;  
        }  
    }  
  
    return arr;  
}  
  
// Example usage:  
const unsortedArray = [4, 2, 3, 1, 0, 4, 6, 5];
```

```
const sortedArray = countingSort(unsortedArray.slice()); // create a copy to keep the original array  
unmodified  
  
console.log(sortedArray); // Output: [0, 1, 2, 3, 4, 4, 5, 6]
```

Пройдемося по коду крок за кроком:

Знаходження Діапазону:

`const max = Math.max(...arr);`: Цей рядок знаходить максимальне значення в вхідному масиві, що допомагає визначити діапазон значень у масиві.

`const min = Math.min(...arr);`: Цей рядок знаходить мінімальне значення в вхідному масиві. Воно використовується пізніше для обчислення індексу в масиві підрахунку.

Масив Підрахунку:

`const countArray = new Array(max - min + 1).fill(0);`: Цей рядок створює масив підрахунку для зберігання кількості кожного елемента. Розмір масиву підрахунку визначається діапазоном значень у вхідному масиві.

Підрахунок Входжень:

`countArray[arr[i] - min]++;`: Цей цикл перебирає вхідний масив і збільшує лічильник для кожного елемента в масиві підрахунку. Вираз `arr[i] - min` використовується для обчислення індексу в масиві підрахунку.

Оновлення Початкового Масиву:

Наступний набір циклів використовується для оновлення початкового масиву в відсортованому порядку на основі масиву підрахунку.

`for (let i = 0; i < countArray.length; i++) {}`: Цей цикл перебирає масив підрахунку.

`while (countArray[i] > 0) {}`: Цей вкладений цикл виконується, доки є входження поточного елемента в масиві підрахунку.

`arr[outputIndex++] = i + min;`: Це оновлює початковий масив відсортованими значеннями. Вираз `i + min` розраховує фактичне значення на основі індексу в масиві підрахунку.

Повернення Відсортованого Масиву:

`return arr;`: Функція повертає початковий масив, який тепер відсортований.

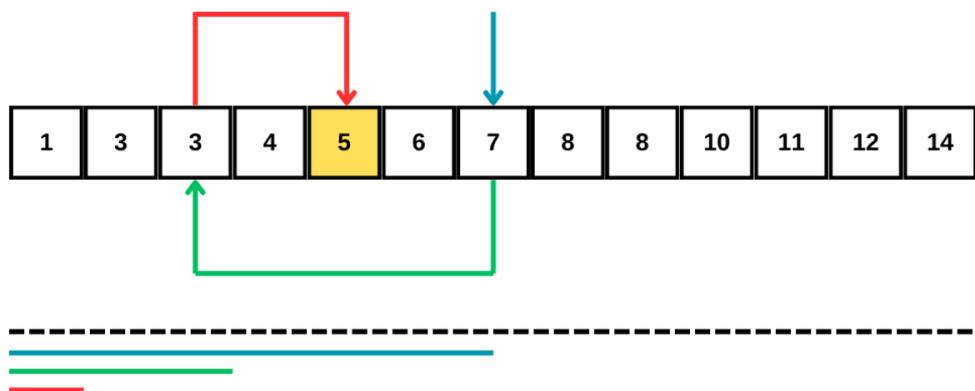
Які є алгоритми пошуку?

Бінарний пошук

Якщо ви хочете знайти статтю в енциклопедії, в якій всі статті впорядковані за алфавітом, тоді вам не потрібно переглядати всі статті, а достатньо відкрити енциклопедію по середині та подивитись на яку літеру починаються статті, якщо ваша літера (буква) передує їй тоді ваша стаття в першій половині енциклопедії, інакше в другій, або ж одразу по центру.

Бінарний (двійковий) пошук — алгоритм знаходження заданого значення у впорядкованому масиві (множина, кортеж), який полягає у порівнянні серединного елемента масиву з шуканим значенням, і повторенням алгоритму для тієї або іншої половини, залежно від результату порівняння. Бінарний пошук працює на відсортованих масивах. Бінарний пошук починається з порівняння елемента в середині масиву з цільовим значенням. Якщо цільове значення відповідає елементу, повертається його позиція в масиві. Якщо цільове значення менше елемента, пошук продовжується в нижній половині масиву. Якщо цільове значення більше за елемент, пошук продовжується у верхній половині масиву. Роблячи це, алгоритм усуває половину, в якій не може лежати цільове значення на кожній ітерації.

Візуалізація бінарного пошуку по масиву. Шукане число - 5.



Алгоритм Кнута для пошуку рядків

Алгоритм Кнута-Морріса-Прата (КМП) є ефективним алгоритмом для пошуку всіх входжень підрядка у тексті. Основна ідея полягає в тому, щоб використовувати інформацію про попередній збіг символів між підрядком та текстом для уникнення зайвих порівнянь.

Основні кроки алгоритму КМП:

1. Побудова таблиці зсувів: Спочатку алгоритм будує таблицю зсувів. Ця таблиця вказує, як далеко можна зсунути підрядок вправо при невідповідності символів. Таблиця заснована на префіксах підрядка.

2. Пошук підрядка в тексті: Після побудови таблиці алгоритм починає порівнювати символи підрядка та тексту. Якщо виявляється невідповідність, таблиця зсувів вказує, як далеко зсунути підрядок вправо, уникаючи порівнянь, які вже були враховані.

3. Знаходження всіх входжень: Алгоритм продовжує здійснювати порівняння та зсувати підрядок вправо до тих пір, поки не знайде всі можливі входження або не завершить пошук.

Переваги алгоритму КМП включають його лінійну складність ($O(n + m)$) = $O(n)$, де n - довжина тексту, а m - довжина підрядка, що шукається. Також, алгоритм ефективний для великих текстів, оскільки він уникає повторних порівнянь.

При будуванні таблиці зсувів важливо враховувати інформацію про префікси та суфікси підрядка. Цей підхід допомагає уникнути зайвого порівняння символів, які вже були порівняні.

Основна ідея полягає в тому, щоб для кожної позиції в підрядку обчислити максимальну довжину коректного префікса, який є також суфіксом. Цю інформацію потім використовують для ефективного зсуву при виявленні неспівпадінь (розвідностей).

Ось простий опис кроків для будування таблиці зсувів:

Ініціалізація: Почнемо зі створення таблиці зсувів розміром рівним довжині підрядка. Ініціалізуємо її значеннями 0.

Заповнення: Для кожної позиції у підрядку обчислюємо довжину найбільшого коректного префікса, який є також суфіксом (позначимо цю величину як "зсув"). Процес заповнення полягає у порівнянні префікса та суфікса, збільшенні зсуву на 1 для кожної одиниці співпадіння (збігу) та оновленні таблиці зсувів.

Префіксом рядка називають будь-яку його частину, яка починається з першого символу. Наприклад, у рядку "ABABC" префіксами є "A", "AB", "ABA", "ABAB".

Суфіксом рядка називають будь-яку його частину, яка закінчується останнім символом. Наприклад, у рядку "ABABC" суфіксами є "C", "BC", "ABC", "BABC".

У виразі "ABAB" таблиця зсувів алгоритму Кнута-Морріса має наступний вигляд: [0, 0, 1, 2], що вказує на те, що найбільший префікс "AB" є також суфіксом на позиції 3. Ця інформація допомагає зменшити кількість порівнянь під час пошуку підрядка, забезпечуючи більш ефективний алгоритм.

A - 0,

AB - 0,

ABA - 1,

ABAB - 2.

Ось простий приклад реалізації алгоритму Кнута-Морріса-Прата (КМП) на JavaScript:

```
function buildKMPTable(pattern) {
    const table = [0];
    let prefixLength = 0;

    for (let i = 1; i < pattern.length; i++) {
        while (prefixLength > 0 && pattern[i] !== pattern[prefixLength]) {
            prefixLength = table[prefixLength - 1];
        }

        if (pattern[i] === pattern[prefixLength]) {
            prefixLength++;
        } else {
            prefixLength = 0;
        }

        table[i] = prefixLength;
    }

    return table;
}

function kmpSearch(text, pattern) {
    const table = buildKMPTable(pattern);
    const matches = [];

    let j = 0; // Index for pattern
    for (let i = 0; i < text.length; i++) {
        while (j > 0 && text[i] !== pattern[j]) {
            j = table[j - 1];
        }

        if (text[i] === pattern[j]) {
            j++;
        }

        if (j === pattern.length) {
            matches.push(i - j + 1);
            j = table[j - 1];
        }
    }

    return matches;
}

// Приклад використання:
const text = "ABABABABCABAABABAB";
const pattern = "ABABC";

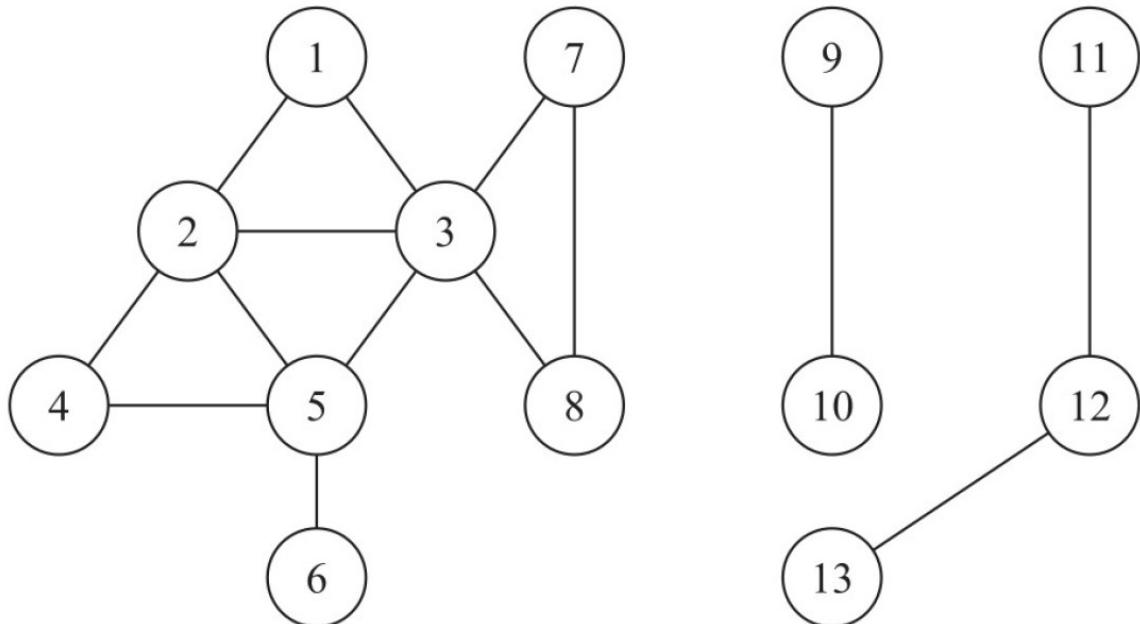
const result = kmpSearch(text, pattern);
console.log("Знайдені збіги:", result);
```

Пошук в ширину та глибину

Граф $G(V, E)$, де V – вершини, E – ребра.

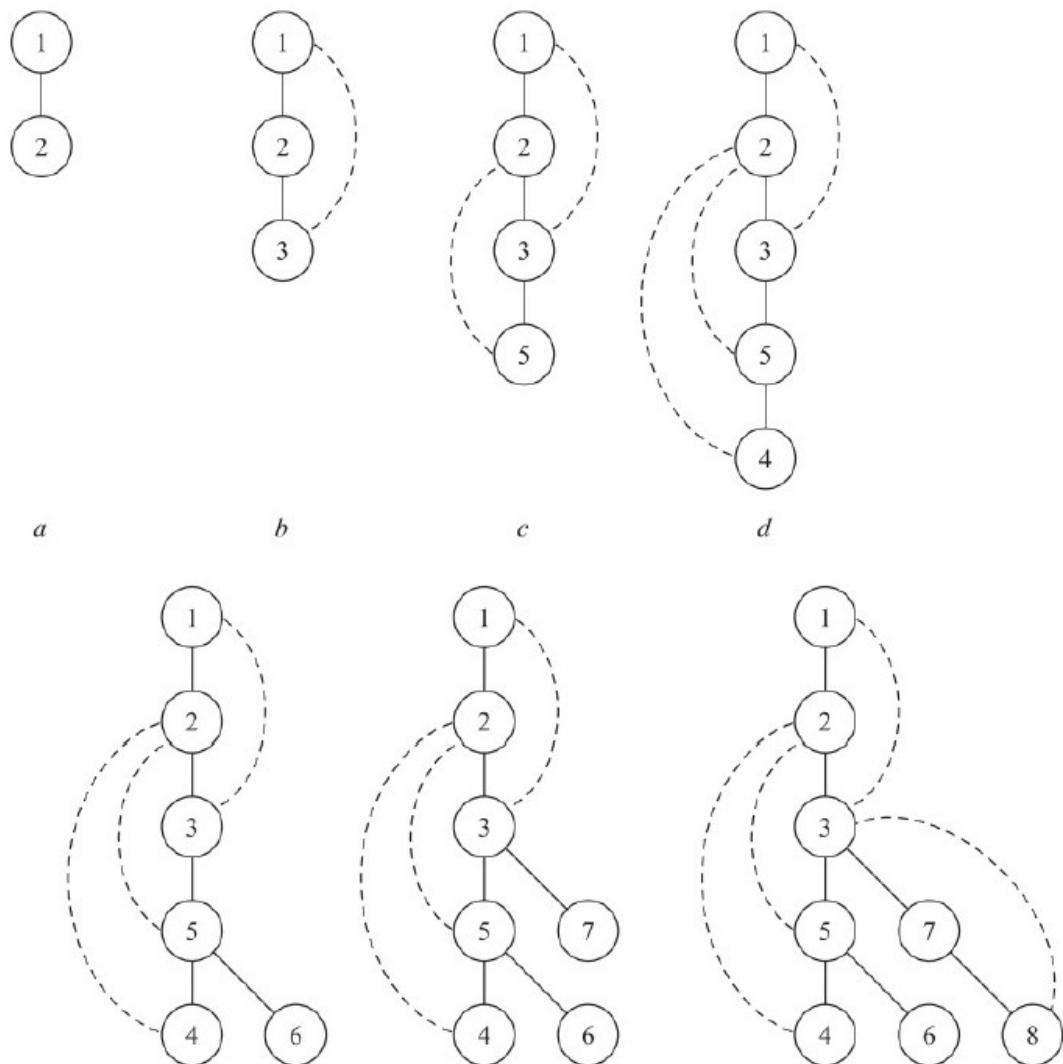
Алгоритм пошуку в ширину. Припустимо, що ми маємо граф $G(V, E)$ та два конкретні вузли s та t . Нам потрібно знайти ефективний алгоритм для відповіді на наступне питання: чи існує в графі шлях від s до t ? Ми будемо називати це проблемою перевірки з'єднаності між s та t . Якщо уявити собі G як лабіринт, кімнати якого відповідають вершинам графа, а коридори — ребрам, що з'єднують вершини (кімнати), то задача полягає в тому, щоб почати з кімнати s та дістатися до іншої заданої кімнати t . Ймовірно, найпростіший алгоритм для перевірки з'єднаності між s та t — це алгоритм пошуку в ширину (Breadth First Search, BFS), який проходить від s у всі можливі напрямки, додаючи один "рівень" за раз. Таким чином, алгоритм розпочинається з s та включає в пошук всі вершини, з'єднані ребром з s — саме так формується перший рівень пошуку. Потім включаються всі вершини (вузли), з'єднані ребром з будь-якою вершиною (вузлом) з первого рівня — другий рівень. Пошук продовжується, поки наступна спроба не знаходить жодної нової вершини.

Є шляхи в цьому графі від вузла 1 до вузлів 2–8, але не до вузлів 9–13.



Якщо, у прикладі на малюнку, ми починаємо з вершини 1, то перший рівень буде складатися з вершин 2 і 3, другий — з вершин 4, 5, 7 і 8, а третій лише з вершини 6. На цьому етапі пошук зупиняється, оскільки нові вершини вже не залишились (зверніть увагу, що вершини 9-13 залишаються недосяжними). Як цей приклад чітко показує, алгоритм має природну фізичну інтерпретацію. В основному ми починаємо з вершини s і послідовно "затоплюємо" граф розширюючи хвилю, яка прагне покрити всі вершини (вузли), які вона може досягти. Рівень вершини представляє момент часу, коли ця вершина буде досягнута пошуком.

Алгоритм пошуку в глибину. Ще один природний метод для знаходження вузлів, досяжних з s , застосовується в ситуації, коли граф G дійсно є лабіринтом взаємопов'язаних кімнат. Ви розпочинаєте з s та перевіряєте перше ребро, що виходить з нього — скажімо, до v . Потім ви слідуєте за першим ребром, що виходить з v , і продовжуєте слідувати за цим шаблоном, поки не потрапите в "тупик". У цьому випадку ви повертаєтесь до вузла, у якого є неверифікований сусід, і продовжуєте з нього. Цей алгоритм називається Пошук в глибину (DFS), оскільки він просувається в глибину G наскільки можливо і відступає лише за необхідності.



Цей приклад дає уявлення про зовнішні відмінності між деревами DFS та деревами BFS. Такі дерева зазвичай є вузькими та глибокими, тоді як останні характеризуються мінімально короткими шляхами від кореня до листя.

Щоб вибратися з класичного лабіринту, достатньо виконати наступний алгоритм:

1. Ідіть вперед і позначте свій шлях стрілками (доріжку можна позначити крейдою на стіні або подряпинами). При поверненні по пройденій дорозі назад стрілки малювати не потрібно.
2. Якщо ви потрапили в глухий кут, то поверніться (за стрілками) до найближчого перехрестя й заблокуйте шлях у глухий кут (закресліть).
3. Якщо ви йшли новим шляхом і зустріли стрілку, тобто шлях, по якому ви вже йшли, то перегородіть цю дорогу з двох кінців, один кінець, що йде до дороги, на яку ви вийшли, а інший кінець по дорозі назад на найближчому до вас перехресті. Тобто поверніться до найближчого до вас перехрестя і перегородіть шлях по цій дорозі (з обох сторін).
4. Продовжуйте виконувати кроки 1-3, доки не знайдете шлях, що веде до виходу.

Цей алгоритм базується на алгоритмі під назвою “пошук у глибину”.

Часова складність цього алгоритму лінійна й залежить від кількості вершин та ребер графа, тобто $O(|V| + |E|)$, де $|V|$ - кількість вершин, $|E|$ - кількість ребер.

Що таке скаляр, вектор та матриця?

Скаляр, вектор та матриця — це поняття, які використовуються в лінійній алгебрі та математиці для опису різних видів даних та об'єктів. Розглянемо кожен з цих термінів окремо:

1. Скаляр: Скаляр — це простий математичний об'єкт, який характеризується лише одним числовим значенням. Скаляри не мають напрямку або розміру. Прикладами скалярів можуть бути числа, такі як 1, 2.5, -3, 14, або будь-які інші числа, які не мають векторного або матричного характеру.
2. Вектор: Вектор — це математичний об'єкт, який містить набір скалярів, які мають відповідні напрямки та розміри. Вектор можна представити як упорядкований набір чисел. Наприклад, вектор може представляти координати точки в просторі, напрямок руху або будь-яку іншу величину, яка має якісь характеристики, що мають значення та напрямок. Вектори зазвичай позначаються літерами зі стрілкою над ними, або жирними літерами, як v .
3. Матриця: Матриця — це таблиця чисел, організованих у визначеній структурі. Матриці складаються з рядків та стовпців, і кожен елемент матриці є скаляром. Матриці використовуються для розгляду різних видів даних та операцій над ними. Вони можуть використовуватися для опису лінійних перетворень, систем лінійних рівнянь, табличних даних та інших структур.

Ось приклади представлення цих понять:

1. Скаляр: Наприклад, число 5.
2. Вектор: Наприклад, вектор позначенням " v " з трьох компонентів, що представляють координати в тривимірному просторі: $v = [2, -3, 1]$.
3. Матриця: Наприклад, матриця A з двома рядками та трьома стовпцями:

$$A = \begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix}$$

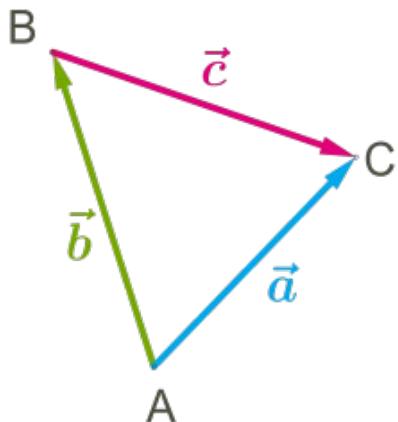
Математика лінійної алгебри використовує ці концепції для аналізу та обробки різних видів даних та вирішення математичних проблем.

Інтуїтивно ми можемо визначити вектор як спрямований відрізок, який має початок і кінець. Довжина вектора зазвичай вказує величину сили, а напрямок вектора вказує напрямок дії сили, який є лінійним.

Кеплер і Ньютона не використовували векторну символіку, її розробив Вільям Гамільтон.

Вільям Ровен Гамільтон (1805—1865) у своїй книзі “Елементи кватерніонів” писав: “Пряма АВ, яка має не тільки довжину, але й напрямок, називається вектором. Його початкова точка А називається його початком; і його кінцева точка В називається його кінцем. Довжина вектора АВ вважається різницею двох його крайніх точок; або, більш повно, є результатом віднімання його власного початку від його власного кінця; Вектор АА або А – А, у такому разі, називається нульовим. Кажуть, що два вектори є рівними, тобто рівняння $AB=CD$ або $BA=DC$ справедливі лише у тому випадку, коли можна здійснити перенесення (або трансляцію) одного вектора так, щоб його початок і кінець збігалися з відповідними точками іншого вектора, не здійснюючи обертання”.

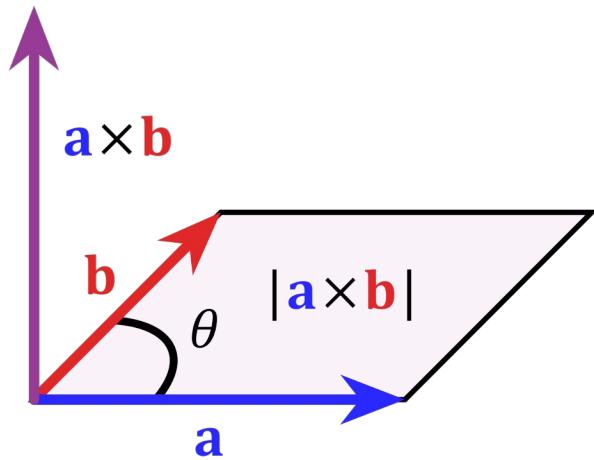
Складання векторів.



$$\vec{c} = \vec{a} - \vec{b}$$

$$\vec{a} = \vec{b} + \vec{c}$$

Векторний добуток.

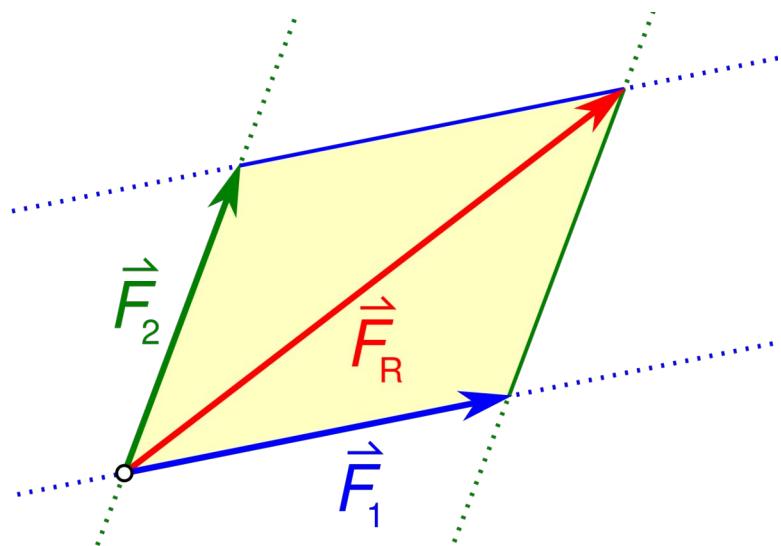


Площа паралелограма дорівнює модулю векторного добутку.

Паралелограм сил — геометрична конструкція, що виражає закон додавання сил. Правило паралелограма сил полягає в тому, що вектор результоючої сили є діагоналлю паралелограма, побудованого на векторах двох доданків сил, як на сторонах. Це робиться тому, що вектор результоючої сили є сумою векторів доданих сил, а сума двох векторів є діагоналлю паралелограма, побудованого на цих векторах. Точне визначення паралелограма сил дав П'єр Варіньон у 1687 році. П'єр Варіньон (1654 - 1722) - французький математик і механік. Навчався в езуїтському коледжі та університеті в Кані, де став майстром у 1682 р. Варіньон був другом Ньютона, Лейбніца та Бернуллі. За винятком Гійома де Лопіталя, Варіньон був першим популяризатором диференціального обчислення у Франції. У 1687 р. у своїй праці "Проект нової механіки..." Варіньон дав точне формулювання закону паралелограма сил, розвинув поняття моменту сил і вивів теорему, яка отримала називу теорема Варіньона.

У своїй роботі "Проект нової механіки..." (1725), проект якої був наданий в 1687 році, Варіньон систематично виклав вчення про додавання і розкладання сил, про моменти сил і про правила для оперування ними.

В праці "Діоптрика" Рене Декарт писав, що швидкість кинutoї кульки можна розкласти на дві складові, а саме, на силу, яка штовхає кульку вбік і силу, яка штовхає кульку вниз. Сучасною мовою це означає, що вектор швидкості можна розкласти на два інші вектори, які є проекціями вектора швидкості на осі координат.



Вектор — це спрямований відрізок, тобто відрізок у двовимірному чи тривимірному просторі, який має чітко позначені початок і кінець (просто кажучи, вектор — це стрілка). Вектори рівні, якщо їх довжини рівні, вони паралельні й односпряжені. Отже, припустимо, що ми маємо одиничний вектор (вектор довжиною в одну одиницю) А і вектор В довжиною 5 одиниць. Для того, щоб перевести вектор А в В, спочатку потрібно встановити йому однакову довжину, тобто помножити його на певне число (скаляр), у нашому випадку на 5. Далі потрібно повернути вектор А так, щоб він став паралельним до вектора В і можна було б провести перпендикуляр від кінця вектора А до кінця вектора В, також з початком векторів. У тривимірному просторі для цього потрібні два кути. Таким чином, маючи значення двох кутів, скаляра (множника довжини) і одиничного вектора, ми можемо перевести одиничний вектор у будь-який інший вектор у тривимірному просторі. Але ви можете зробити це простіше, відповідно до теореми Ейлера про обертання. Щоб перевести один вектор А в інший, достатньо вектор А помножити на певний вектор, а потім повернути його на певний кут (навколо його кінця). Таким чином, щоб перевести вектор А в будь-який інший, достатньо вектор А помножити на певний вектор і на число, яке позначає обертання вектора навколо його кінця.

Кватерніон — це така математична сутність, яка складається з вектора і числа. Тобто, помноживши вектор на певний кватерніон, можна перевести цей вектор в будь-який інший, що випливає з теореми Ейлера про обертання. “кватерніон, розглядається як фактор, який змінює один певний вектор на інший” (Пітер Гатрі Тейт, “Елементарний трактат про кватерніони”, 1890)

Німецький математик Леонард Ейлер в 1776 році довів теорему про обертання.

Теорема про обертання Ейлера стверджує, що будь-який рух твердого тіла в тривимірному просторі, що має нерухому точку, є обертанням тіла навколо певної осі. Таким чином, обертання можна описати трьома координатами: двома координатами осі обертання (наприклад, широтою і довготою) і кутом повороту.

Теорема Ейлера про обертання: тверде тіло, яке має одну нерухому точку, можна перевести з одного положення в будь-яке одним обертом на деякий кут навколо нерухомої осі, що проходить через нерухому точку. З цієї теореми Ейлера випливає теорема Шаля: "Будь-який рух площини, що змінює орієнтацію, є осьовою (обертальною) або ковзною симетрією".

Скажімо, у нас є точка (x_1, y_1) . Точка також визначає вектор, тобто $\langle 0, 0, x_1, y_1 \rangle$.

Вектор $\langle 0, 0, x_1, y_1 \rangle$ має довжину L, тобто $\sqrt{(x_1^2 + y_1^2)}$.

Ми повертаемо цей вектор проти годинникової стрілки навколо початку координат на β градусів.

Повернутий вектор має координати (x_2, y_2) . Повернутий вектор також повинен мати довжину L.

Теорема:

$$x_2 = \cos(\beta) * x_1 - \sin(\beta) * y_1;$$

$$y_2 = \sin(\beta) * x_1 + \cos(\beta) * y_1;$$

Назвемо кут між (x_1, y_1) і віссю х: α .

Далі:

$$x_1 = L * \cos(\alpha);$$
$$y_1 = L * \sin(\alpha);$$

Обертаємо (x_1, y_1) на кут β , щоб отримати (x_2, y_2) .

$$x_2 = L * \cos(\alpha + \beta);$$
$$y_2 = L * \sin(\alpha + \beta);$$

Правило суми кутів дає нам:

$$\cos(\alpha + \beta) = \cos(\alpha) * \cos(\beta) - \sin(\alpha) * \sin(\beta);$$
$$\sin(\alpha + \beta) = \sin(\alpha) * \cos(\beta) + \cos(\alpha) * \sin(\beta);$$

Отже:

$$L * \cos(\alpha + \beta) = L * \cos(\alpha) * \cos(\beta) - L * \sin(\alpha) * \sin(\beta) \Rightarrow x_2 = x_1 * \cos(\beta) - y_1 * \sin(\beta);$$

Матриця — це множина, яка містить одну або декілька множин з однаковою кількістю елементів, наприклад, $\{\{1,2,3\}, \{4,5,6\}\}$. Матриця зазвичай містить числа, які можна змінювати з допомогою матричних операцій: додавання, віднімання, множення. Ділення для матриць не визначене.

Матриця — математичний об'єкт, записаний у вигляді прямокутної таблиці чисел (чи елементів кільця), він допускає операції (додавання, віднімання, множення та множення на скаляр).

Зазвичай, матриці представляються двовимірними (прямокутними) таблицями.

Матриці є корисними для запису даних, що залежать від двох категорій, наприклад: для коефіцієнтів систем лінійних рівнянь та лінійних перетворень.

Розмір матриці визначає кількість рядків і стовпців, які вона містить.

Матрицю із m рядками та n стовпцями називають матрицею $m \times n$ або m -на- n матрицею, а самі m і n називають розмірами матриці.

Множення двох матриць має сенс лише тоді, коли число стовпчиків першої матриці дорівнює числу рядків другої матриці.

Поняття “матриці”, яке вже не було похідним від поняття “візничник” з'явилось тільки в 1858 році в праці англійського математика Артура Кейлі. Термін “матриця” першим став вживати Джеймс Джозеф Сильвестр, який розглядав матрицю, як об'єкт, що породжує сімейство мінорів (візничників менших матриць, утворених викреслюванням рядків та стовпців з початкової матриці). Матриці корисні в економіці для розрахунку різних тенденцій як векторів. Матриці також корисні в комп'ютерній графіці для представлення систем координат і об'єктів.

$$\begin{array}{ccc}
 & \vec{b}_1 & \vec{b}_2 \\
 & \downarrow & \downarrow \\
 \vec{a}_1 \rightarrow & \left[\begin{array}{cc} 1 & 7 \\ 2 & 4 \end{array} \right] \cdot & \left[\begin{array}{cc} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 \end{array} \right] \\
 \vec{a}_2 \rightarrow & A & B & C
 \end{array}$$

Такий метод множення (коли множиться рядок на стовпець) дозволяє зберігати початковий розмір матриці.

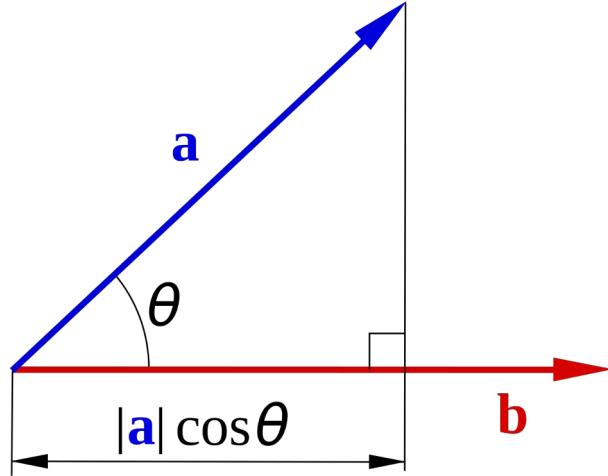
$a = \langle x, y \rangle$;
 $b = \langle v, w \rangle$;
 $a * b = |a| * |b| * \cos(\theta) = xv + yw$; (скалярний добуток).

Якщо вектор тривимірний, тоді скалярний добуток буде:

$$a = \langle x, y, z \rangle;$$

$$b = \langle u, v, w \rangle;$$

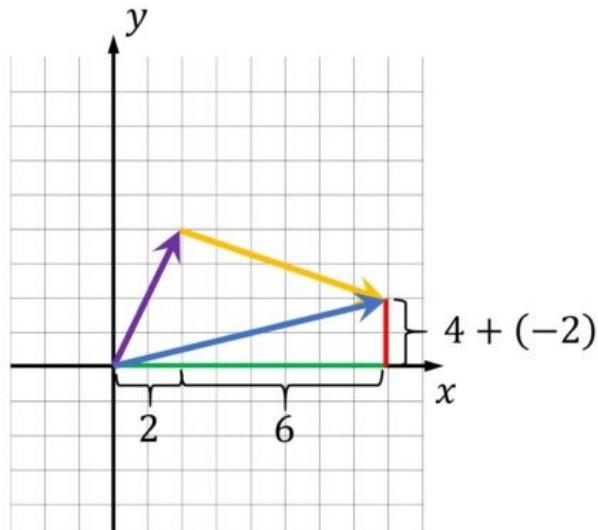
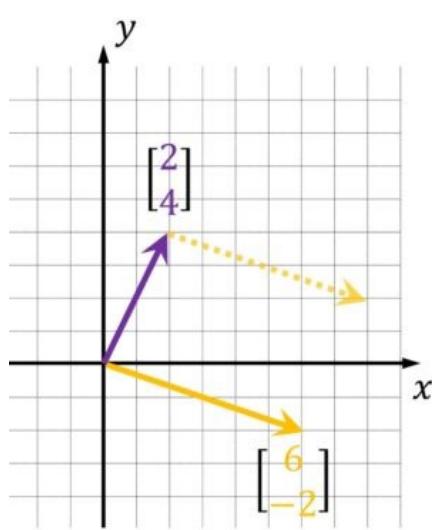
$$a * b = xu + yv + zw;$$



Скалярний добуток векторів a, b дорівнює $|a| * |b| * \cos(\theta)$;

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 4 \end{bmatrix} + \begin{bmatrix} 6 \\ -2 \end{bmatrix} = \begin{bmatrix} 2+6 \\ 4+(-2) \end{bmatrix} = \begin{bmatrix} 8 \\ 2 \end{bmatrix}$$



Додавання матриць — це операція додавання двох матриць, що розраховується за допомогою додавання відповідних елементів. Для додавання дві матриці повинні мати відповідну кількість рядків та стовпчиків. Також можна відняти одну матрицю від іншої, якщо вони мають одинаковий розмір. Віднімання матриць (різниця матриць) $A - B$ — це операція обчислення матриці C , всі елементи якої рівні попарній різниці всіх відповідних елементів матриць A та B .

Основні властивості операцій додавання матриць:

$A + B = B + A$ (комутативність).

$A + (B + C) = (A + B) + C$ (асоціативність).

$A + 0 = A$, при будь-якій матриці.

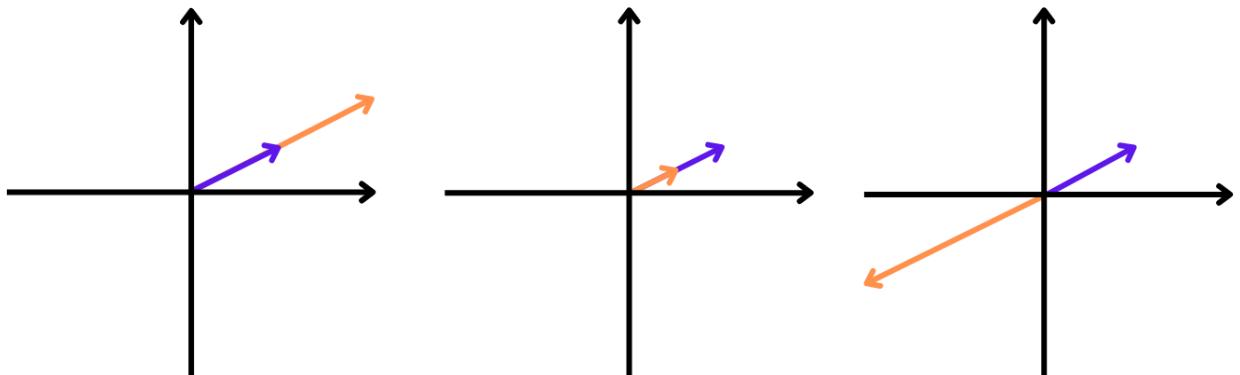
Для будь-якої матриці A існує протилежна матриця $(-A)$, така, що $A + (-A) = 0$.

Добутком матриці A на число k називається матриця B = k * A того ж розміру, отримана з початкової матриці множенням на задане число всіх її елементів.

$$2 * [4, 2] = [8, 4]$$

$$0.5 * [4, 2] = [2, 1]$$

$$-2 * [4, 2] = [-8, -4]$$



Ось функція (мовою JavaScript) для повороту вектора на заданий кут:

```
function rotateVector(vector, angle) {
    // Перетворюємо кут з градусів в радіани
    const angleInRadians = (angle * Math.PI) / 180;

    // Розраховуємо координати обернутого вектора
    const rotatedX = vector.x * Math.cos(angleInRadians) - vector.y * Math.sin(angleInRadians);
    const rotatedY = vector.x * Math.sin(angleInRadians) + vector.y * Math.cos(angleInRadians);

    // Повертаємо новий вектор
    return { x: rotatedX, y: rotatedY };
}

// Приклад використання
const inputVector = { x: 1, y: 1 };
const rotationAngle = 90; // Кут обертання в градусах

const rotatedVector = rotateVector(inputVector, rotationAngle);

console.log('Початковий вектор: (${inputVector.x}, ${inputVector.y})');
console.log('Обернутий вектор: (${rotatedVector.x.toFixed(2)}, ${rotatedVector.y.toFixed(2)})');
```

Які найпопулярніші алгоритми й теореми в арифметиці?

Основні закони арифметики:

1. Асоціативний закон:

Це властивість операції, за якою порядок виконання операцій не впливає на результат. Наприклад, у математиці для додавання це виглядає так: $(a + b) + c = a + (b + c)$.

2. Комутативний закон:

Це властивість операції, за якою порядок елементів не впливає на результат. Наприклад, для додавання: $a + b = b + a$.

3. Дистрибутивний закон:

Це закон, який об'єднує додавання та множення. Наприклад, дистрибутивний закон для множення відносно додавання виглядає так: $a * (b + c) = a * b + a * c$.

4. Розподільний закон множення:

$$(a + b) * (c + d) = ac + ad + bc + bd;$$

5. Формули скороченого множення:

квадрат суми двох виразів дорівнює квадрат першого виразу додати подвоєний добуток цих виразів додати квадрат другого виразу.

$$(a + b)^2 = a^2 + 2ab + b^2;$$

квадрат різниці двох виразів дорівнює квадрат першого виразу відняти подвоєний добуток цих виразів додати квадрат другого виразу.

$$(a - b)^2 = a^2 - 2ab + b^2;$$

У математиці та арифметиці використовується певний порядок виконання операцій, відомий як "PEMDAS":

- Дужки (Parentheses): Операції всередині дужок виконуються першими.
- Піднесення до степеня (Exponents): Потім виконується піднесення чисел до степеня.
- Множення (Multiplication) та Ділення (Division): Ці операції виконуються зліва направо.
- Додавання (Addition) та Віднімання (Subtraction): Також виконуються зліва направо.

Розглянемо вираз: $4 * (5 - 2)^2 + 7$

Виконуємо операції всередині дужок: $5 - 2 = 3$.

Підносимо отримане значення до квадрата: $3^2 = 9$.

Помножимо отримане число на 4: $4 * 9 = 36$.

Додаємо 7 до отриманого результату: $36 + 7 = 43$.

Таким чином, результат виразу $4 * (5 - 2)^2 + 7$ дорівнює 43.

Від зміни доданків сума не змінюється (додавання та множення): $a + b = b + a$; $a * b = b * a$;

Мінус на мінус дає плюс: $-1 * -1 = 1$;

Мінус на плюс дає мінус: $1 * -1 = -1 * 1 = -1$;

Якщо $a \neq b$, тоді $a - b \neq b - a$.

Коли у вас є дріб у показнику степеня, ви можете інтерпретувати це як отримання кореня з основи.

Наприклад, $a^{n/m}$ представляє корінь n -го ступеня з a у степені m .

$0,125 = 1/8$, отже, $5^{0,125}$ еквівалентно 8-му кореню з 5.

Квадратні числа

В 1 столітті нашої ери народився математик Нікомах Гераський. Гераса, в якій народився Нікомах,— це сучасний Джераш на півночі Йорданії. Збереглись два твори Нікомаха, а саме: "Вступ до арифметики" і "Керівництво з гармоніки". У своїх творах Нікомах використовував грецькі букви для запису чисел.

На початку 5 століття нашої ери єпископ Аврелій Августин писав: "Не варто нехтувати цифрами". Нікомах у своїй праці про арифметику розповідає про "досконалі числа".

Досконале число — це натуральне число, що дорівнює сумі своїх додатних дільників, не враховуючи самого числа. Наприклад, 6 має дільники 1, 2, 3 (не враховуючи його самого) і $6 = 1 + 2 + 3$, тому 6 — досконале число.

Нікомах наводить 4 приклади досконалих чисел: 6, 28, 496, 8128.

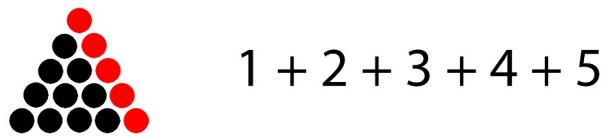
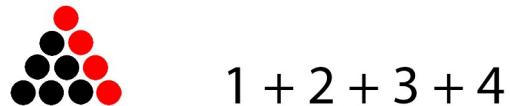
Він також дає загальне правило для знаходження таких чисел, доказ якого міститься в "Елементах" Евкліда, книга 9, пропозиція 36.

Якщо сума $1 + 2 + 2^2 + \dots + 2^n = p$, а p — просте число, то $2^n * p$ — досконале число.

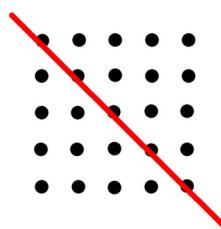
Наприклад: $1 + 2 + 4 = 7$, а $7 * 4 = 28$, отже 28 є досконалим числом, тому що $1 + 2 + 4 + 7 + 14 = 28$.

Нікомах в книзі "Вступ до арифметики" описує трикутні та квадратні числа.

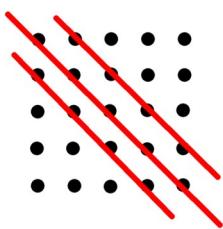
Трикутне число.



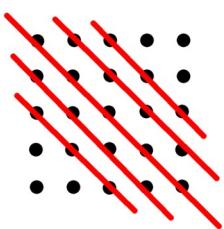
Квадратне число.



5



$4 + 5 + 4$



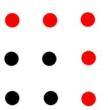
$3 + 4 + 5 + 4 + 3$



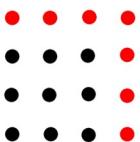
1



$1 + 3$



$1 + 3 + 5$



$1 + 3 + 5 + 7$

Квадратне число можна отримати з допомогою формули:

n^2 , або $n + 2*((n-1) + (n-2) + \dots + (n-n))$.

Трикутне число можна отримати за формулою: $n + (n-1) + (n-2) + \dots + (n-n)$.

Нікомах у своїй праці про арифметику описує табличку множення Піфагора.

Табличка множення у праці Нікомаха.

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Для запису чисел Нікомах використовував грецькі літери. Відоме число шістсот шістдесят шість записано в Новому Заповіті грецькими літерами (буквами).

Існує формула для знаходження піфагорових трійок, тобто чисел, що задовольняють теорему Піфагора, наприклад, 3,4,5,бо $3^2 + 4^2 = 5^2$.

Якщо (a, b, c) – піфагорова трійка, то і (ka, kb, kc) – піфагорова трійка для будь-якого натурального числа k . Наприклад, $(3, 4, 5)$ та $(6, 8, 10)$. Примітивна піфагорова трійка — це трійка, в якій a, b і c взаємно прості (тобто вони не мають загального дільника більше ніж 1). Існує 16 примітивних піфагорійських трійок чисел до 100: $(3, 4, 5), (5, 12, 13), (8, 15, 17), (7, 24, 25), \dots$

Піфагорові трійки вмів знаходити Евклід.

Числа Фібоначчі

У "Кнізі Абака" описані основи підрахунку з допомогою арабських чисел, а також відомі числа Фібоначчі.

У "Кнізі абака" Фібоначчі описав числовий ряд, числа якого тепер називають числами Фібоначчі. Ряд Фібоначчі складається з чисел у такому порядку:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Кожне наступне число є сумою двох попередніх.

Розвиток цієї числової послідовності був заснований на задачі про кроликів.

Ми опишемо задачу в наступному вигляді:

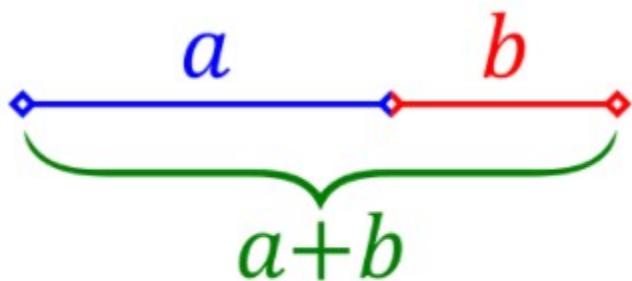
У нас є пара кролів, які щомісяця народжують двох кроленят, які через місяць виростають і теж народжують пару кроленят, скільки буде кролів через 12 місяців?

Відповідь дасть ряд Фібоначчі: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, а саме, буде 233 пари кролів.
Золотий перетин — відношення двох величин a і b , в якому більша величина відноситься до меншої як сума значень до більшої тобто:

$$a / b = (a+b) / a = \varphi.$$

φ — число, яке позначає золотий перетин, і воно ірраціональне тобто записане у вигляді нескінченного неперіодичного дробу.

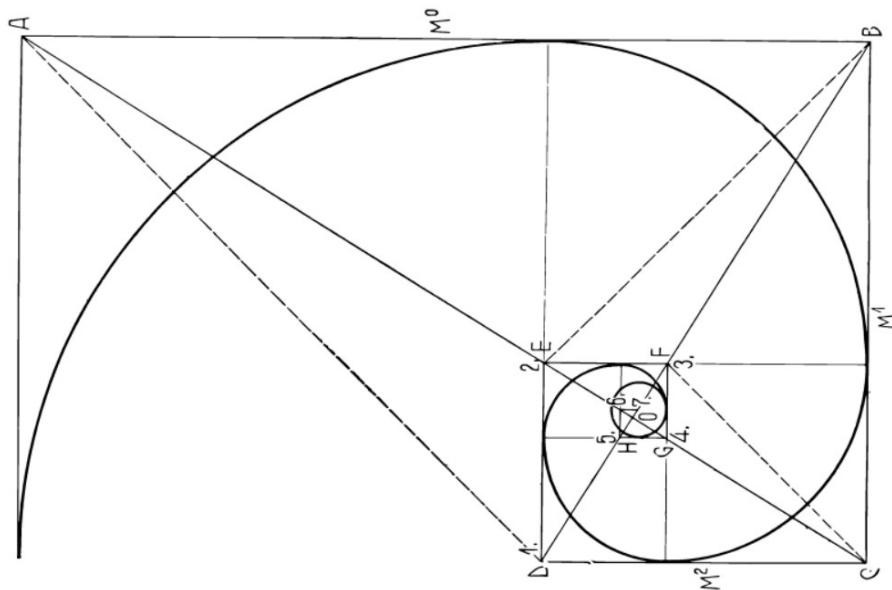
$$\varphi = 1,6180339887498948482\dots$$



$$\varphi = (a+b) : a = a : b$$

Межа відношення чисел Фіbonacci дорівнює золотому перетину тобто чим більшими будуть два числа, які ви візьмете з ряду Фіbonacci підставивши ці числа у формулу $a / b = (a + b) / a$, тим біжче буде число до золотого перетину (1,6180 ...). Золотий перетин був відомий Евкліду, він згадує його в шостій книзі своєї праці “Елементи”, а саме: “Кажуть, що пряма була розрізана в крайньому і середньому відношенні, коли як ціле (вся пряма) до більшого відрізка, так і більший (відрізок) до меншого”.

Золота спіраль — логарифмічна спіраль, швидкість зростання якої дорівнює ϕ — золотій пропорції.



Золота спіраль була відома як мінімум з початку 20 століття. У працях Фібоначчі вона не зображена. Золота спіраль зображена в книзі "Життєві криві" (1914) британського художника Теодора Кука.

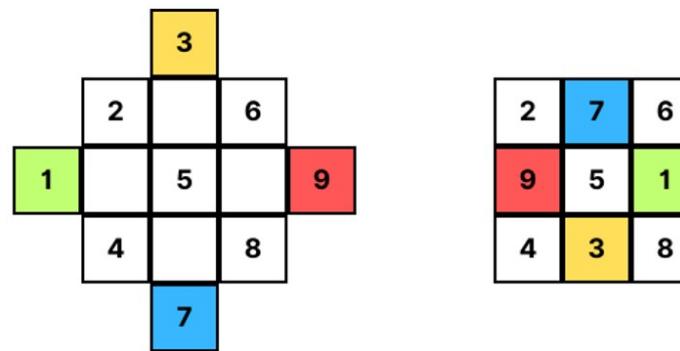
Магічні квадрати

Квадратний масив чисел $n \times n$, як правило, додатних цілих чисел, називається магічним квадратом, якщо суми чисел у кожному рядку, кожному стовпці та обох головних діагоналях однакові.

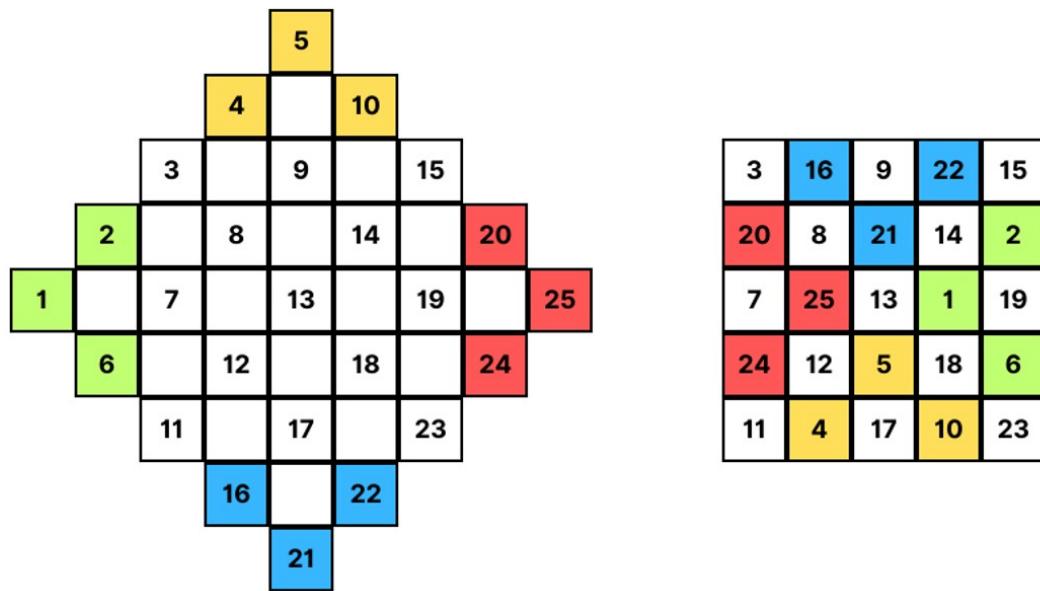
Магічні квадрати зазвичай класифікуються відповідно до порядку n : непарні, якщо n непарне, парні, якщо n парне число. Ця класифікація базується на різних техніках, необхідних для побудови непарних, парних і непарних квадратів.

Непарні магічні квадрати досить легко будуються за допомогою методу де Мезіріака. Французький математик Клод Баше де Мезіріак опублікував цей метод в праці "Збірник цікавих завдань" (1612).

3x3



5x5



Арифметична прогресія

Цікава задача з “Арифметики” Магніцького: Якийсь чоловік продавав коня за 156 рублів, покупець каже продавцю: “Я не можу купити вашого коня, бо він не вартий цієї ціни”. Продавець запропонував йому іншу ціну, сказавши: “Якщо вам здається, що ця ціна завищена для такого коня, то купіть лише цвяхи в його підкову, а коня візьміть безплатно. У кожній підкові шість цвяхів, і за перший цвях даси мені чверть копійки, за другий дві чверті, за третій одну копійку, і так ти купиш усі цвяхи”. Покупець подумав, що це дуже вигідна ціна, і погодився, сподіваючись заплатити за всі цвяхи не більше 10 рублів. Насправді покупець уклав невдалу угоду і заплатив набагато більше. 1 рубль = 100 копійок.

Всього у коня 24 цвяхи.

Цвях 1 — 0,25 копійки.

Цвях 2 — 0,5.

Цвях 3 — 1 копійка.

Цвях 4 — 2.

Цвях 5 — 4.

Цвях 6 — 8.

....

Ми бачимо, що задача має геометричну прогресію.

І це: $0,25 + 0,5 + 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 256 + 512 + 1024 + 2048 + 4096 + 8192 + 16384 + 32768 + 65536 + 131072 + 262144 + 524288 + 1048576 + 2097152 = 4194303,75$ копійки, тобто 41943 рублі, якщо округлити.

Геометрична прогресія — послідовність чисел b_1, b_2, b_3 (називається членами прогресії), в якій кожне наступне число, починаючи з другого, одержується з попереднього шляхом множення його на певне число q (називається знаменником від прогресування), де $b_1 \neq 0$ і $q \neq 0$, наприклад, $b_1, b_2 = b_1q, b_3 = b_2q, \dots, b_n = b_{n-1}q$. Ряд степенів двійки є геометричною прогресією, тобто 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, ...

Арифметична прогресія — числовая послідовність виду $a_1, a_1 + d, a_1 + 2d, \dots, a_1 + (n-1)d, \dots$ тобто послідовність чисел (членів прогресії), в якій кожне число, починаючи з другого, виходить з попереднього шляхом додавання до нього постійного числа d (крок, або різниця прогресії): $a_n = a_{n-1} + d$. Будь-який (n -ий) член прогресії може бути обчисленний з допомогою загальної формули терміну: $a_n = a_1 + (n-1)d$. Арифметична прогресія — це монотонна послідовність.

Якщо ми розглядаємо натуральні числа як послідовність, починаючи з 1, то ця послідовність дійсно може бути розглянута як арифметична прогресія з різницею 1. Тобто послідовність натуральних чисел 1, 2, 3, 4, 5, ... можна трактувати як арифметичну прогресію з початковим членом 1 і різницею 1.

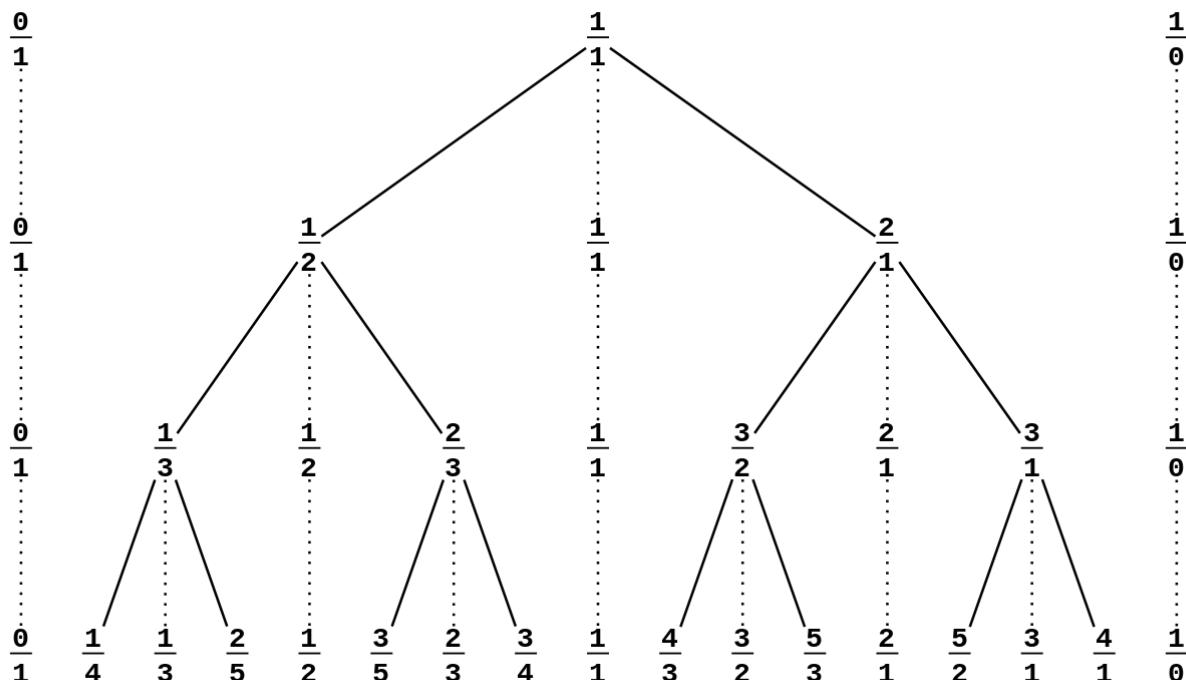
У книзі Аль-Каші "Ключ арифметики" є така задача: "Люди увійшли в сад і один зірвав один гранат, другий два гранати, третій три й так далі. В кінці все, що було зірвано, поділили порівну. Кожен отримав по шість гранатів. Скільки людей увійшло в сад?" Цю відповідь можна знайти аналітично. Кількість людей завжди буде більше середнього арифметичного всіх фруктів, у нашому випадку людей буде більше ніж 6. Відсутність плодів у перших п'яти має бути заповнений п'ятьма людьми, що слідують за шостим, тобто відповідь 11. У саду було 11 людей. Формула для знаходження кількості людей: $a + (a-1)$, де ' a ' — середнє арифметичне загальної кількості зірваних плодів.

Шахи були винайдені приблизно в 7 столітті нашої ери арабами чи індійцями. За однією з легенд, мудрець Сісса ібн Дахір, якому приписують винахід шахів, показав свій винахід правителю країни, якому так сподобалася гра, що він надав винахіднику право вибирати нагороду самому. Мудрець попросив царя заплатити йому одне зерно пшениці за перше поле шахової дошки, два за друге,

четири за третє і так далі, подвоюючи кількість зерен у кожному наступному полі. Правитель, не розбираючись у математиці, швидко погодився, навіть дещо ображений такою низькою оцінкою винаходу, і наказав скарбнику підрахувати й дати винахідникові необхідну кількість зерна. Однак, коли через тиждень скарбник ще не міг підрахувати, скільки потрібно зерна, намісник запитав, що стало причиною такої затримки. Скарбник показав йому розрахунки й сказав, що неможливо розрахуватися, хіба що осушити моря й океани та засіяти пшеницею всю область. Кількість зерна становить приблизно у 1800 разів більше світового врожаю пшениці за рік. Правитель не виконав умови Сісса ібн Дахіра і стратив його. Про шахи писав Аль-Адлі, Аль-Сулі (880 – 948), Аль-Біруні (973 - 1048).

Дерево Штерна-Броко

Дерево Штерна-Броко – це зручний спосіб побудувати множину всіх невід’ємних дробів. Суть цього методу полягає в тому, щоб почати з двох дробів ($0/1$ і $1/0$), а потім повторити наступну операцію необхідну кількість разів: вставити дріб $(m + m') / (n + n')$ між двома сусідніми дробами m / n і m' / n' . Дріб $(m + m') / (n + n')$ називають медіантою дробів m / n і m' / n' . Наприклад, перший крок дає один новий запис між $0/1$ і $1/0$, а саме: $0/1, 1/1, 1/0$.



У теорії чисел дерево Штерна–Броко є нескінченим повним бінарним (двійковим) деревом, вершини якого відповідають один до одного позитивним раціональним числам, значення яких впорядковані зліва направо, як у дереві пошуку. Дерево Штерна-Броко — це спосіб упорядкування всіх невід’ємних незводних дробів у вершинах впорядкованого нескінченного бінарного дерева. Дерево Штерна-Броко було відкрито незалежно Моріц Штерном (1858) і Ахілле Броко (1861). Штерн був німецьким теоретиком чисел, а Броко був французьким годинниковим майстром, який використовував дерево Штерна-Броко для розробки систем передач із передавальним відношенням, близьким до певного бажаного значення, знаходячи співвідношення чисел поблизу цього значення.

Алгоритм Евкліда для найбільшого спільногодільника

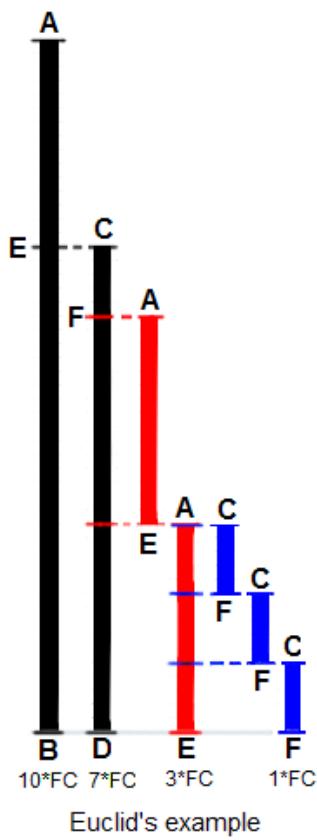
Алгоритм Евкліда є ефективним методом для знаходження найбільшого спільногодільника (НСД) двох чисел, тобто найбільшого числа, яке ділить їх без остачі.

Найбільший спільний дільник не може бути більшим ніж найменше з двох чисел, які він ділить.

Суть алгоритму Евкліда:

1. З двох чисел беремо найменше і дивимось чи ділить воно без остачі більше число, якщо так, тоді воно найбільший спільний дільник.

2. Якщо менше з двох чисел не ділить більше число без остачі, тоді беремо різницю двох чисел, а потім шукаємо спільний дільник цієї різниці та меншого з двох початкових чисел, тобто виконуємо знов першу дію між цими двома числами. Так продовжуємо повторювати доки не знайдемо спільний дільник.



Euclid's example

Метод Евкліда для знаходження найбільшого спільногодільника двох початкових довжин BA і DC. Оскільки довжина DC менша, вона використовується для вимірювання BA, але лише один раз, оскільки залишок EA менший за DC. Тепер EA (двічі) вимірює меншу довжину DC, а залишок FC коротший, ніж EA. Тоді FC вимірює (тричі) довжину EA. Оскільки немає залишку, процес закінчується тим, що FC є найбільшим спільногодільником.

Найменше спільне кратне (НСК) двох цілих чисел — найменше натуральне число, яке є кратним обох цих чисел. Найменше спільне кратне (НСК) можна теж обчислити за допомогою рівності $\text{НСК}(a, b) = |ab|/\text{НСД}(a, b)$, використавши для обчислення найбільшого спільногодільника (НСД) ефективний алгоритм Евкліда.

Скажімо, у вас є два поля для вирощування рослин, одне розміром 36 квадратних метрів, а інше - 48 квадратних метрів. Ви хочете поділити ці поля на однакові за розміром ділянки для висадки рослин. Який найбільший розмір кожної ділянки, яку ви можете вирізати з обох полів, так щоб не залишалося непокритих частин?

Для розв'язання цієї задачі потрібно знайти найбільший спільний дільник (НСД) розмірів обох полів (36 та 48). В цьому випадку, НСД(36, 48) = 12. Таким чином, ви можете вирізати квадратні ділянки розміром 12 квадратних метрів з обох полів, і вони будуть мати одинаковий розмір та покривати всю площину кожного поля.

Два числа x і y є взаємно простими, якщо їхній найбільший спільний дільник (НСД) дорівнює 1. Отже, якщо n ділиться і на x , і на y , вони також ділиться на xy .

Коли x і y не є взаємно простими, вони мають деякі спільні множники. У цьому випадку найменше число, яке ділиться як на x , так і на y , є їхнім найменшим спільним кратним (НСК), а не їх добутком.

Теорема Ламе: якщо алгоритм Евкліда вимагає k кроків для обчислення НСД деякої пари, то менше число в парі має бути більшим або дорівнювати k -му числу Фібоначчі.

Теорему довів французький математик Габріель Ламе в 1845 році.

Просте число — це натуральне число, що ділиться без остачі тільки саме на себе й одиницею, при цьому одиниця не є простим числом, бо не задоволяє основну теорему арифметики про розкладання будь-якого числа на прості множники. Прості числа: 2, 3, 5, 7, 11, 13, 17, ...

Евклід у двадцятій пропозиції дев'ятої книги його збірки "Елементи" доводить методом математичної індукції та доказом від супротивного, що немає найбільшого простого числа.

Теорема Евкліда про відсутність найбільшого простого числа:

Припустимо, що в нас є найбільше просте число. Візьмемо найбільше просте число та всі інші прості числа і перемножимо їх, відтак, отримаємо якесь число P . До P додамо одиницю, тобто отримаємо число $Q = P + 1$. Всі числа прості або складені, отже число Q просте або складене. Якщо Q просте, тоді ми знайшли більше просте число ніж те, що в нас було. Якщо ж число Q не є простим, тоді воно складене, але не може ділитись на відомі нам прості числа, через специфіку свого конструювання, отже існує просте число не відоме нам і воно більше ніж всі прості числа, що були на початку. Цей розсуд можна проводити для будь-якого набору простих чисел, тому гіпотеза про те, що існує найбільше просте число є хибною.

Просте число — це натуральне число, більше одиниці, яке не можна утворити множенням двох менших натуральних чисел, тобто просте число ділиться тільки саме на себе і на одиницю. Всі прості числа крім двійки є також непарними числами, адже не ділиться на два.

Сито Ератосфена

Сито Ератосфена — це алгоритм для знаходження простих чисел методом відсіювання всіх не простих чисел з певної послідовності. Метод знаходження простих чисел за Ератосфеном полягає ось в чому: Візьміть послідовність чисел від 2 до n , тобто $2, 3, 4, 5, \dots, n$. Візьміть перше відоме вам просте число, наприклад, 2. Відсійте зі списку всі числа кратні двом, тобто складені з певної кількості двійок, але саму двійку залишіть у списку. Після цього візьміть число зі списку, що наступне після двійки та не кратне їй, тобто найближче до двійки. Логіка каже, що це число буде простим, а саме, числом 3. Відсійте всі числа зі списку, що кратні трійці і перейдіть до наступного числа. Продовжуйте цей процес доки в списку не залишатиметься одні прості числа.

Детальніше метод Ератосфена можна описати так:

1. Створіть список послідовних цілих чисел від 2 до n : $(2, 3, 4, \dots, n)$.
2. Спочатку нехай p дорівнює 2, найменшому простому числу.
3. Перерахуйте кратні p , підраховуючи кроки p від $2p$ до n , і позначте їх у списку (це будуть $2p, 3p, 4p, \dots$; саме p не слід позначати).
4. Знайдіть у списку перше число, більше за p , яке не позначено. Якщо такого числа не було, зупиніться.

В іншому випадку нехай p тепер дорівнює цьому новому числу (яке є наступним простим) і повторіть з кроку 3.

5. Коли алгоритм завершується, числа, що залишилися не позначені в списку, є всіма простими числами нижче n .

Наприклад:

Крок:

$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$.

Крок: $p = 2$.

Крок: $2, 3, (4), 5, (6), 7, (8), 9, (10), 11, (12), 13, (14), 15$.

Крок: $p = 3$.

Крок: $2, 3, (4), 5, (6), 7, (8), (9), (10), 11, (12), 13, (14), (15)$.

Крок: $2, 3, 5, 7, 11, 13$ — прості числа.

Доказ існування іrrаціональних чисел

Квадратний корінь з двійки не є раціональним числом.

Доказ (від супротивного): Ми знаємо, що квадрат непарних чисел дає непарні числа, а квадрат парних чисел — парні числа. Ми припустимо, що квадратний корінь з двійки — це деяке раціональне число, тобто $\sqrt{2} = a/b$, і що дріб a/b не можна скоротити (змінити на якийсь рівний дріб x/y , де $x < a$ і $y < b$). Тоді $a^2/b^2 = 2$, що означає $a^2/b^2 = 2p/p$ (тобто $a^2 = 2p$ і $b = p$). Якщо $a^2 = 2p$, тоді $a/2 > 0$ і $a/2$ є натуральним числом, простими словами, a є парним числом. Якщо a та a^2 є парними числами, очевидно, p і b також повинні бути парними числами. Отже, ми отримуємо, що $a/b = 2v/2w$, і це означає, що цей дріб можна скоротити до v/w , але це суперечить нашому припущення, що квадратний корінь з двійки не можна скоротити. Таким чином, ми можемо зробити висновок, що наше перше припущення про те, що $\sqrt{2}$ є деяким раціональним числом, тобто $\sqrt{2} = a/b$, є хибним, неправильним, тому що в іншому випадку цей дріб буде завжди скорочуваним, що не може бути істинним відповідно до аксіом та визначень натуральних чисел.

Метод Герона для знаходження кореня степені n

Для знаходження кореня квадратного, кубічного, та коренів інших степенів, можна використовувати наступний алгоритм.

Алгоритм знаходження кореня степені n.

$$A = a^n;$$

Крок 1. Зробіть початкове припущення:

$$x_0 \approx \text{root}(A, n);$$

Якщо $A > 1$, тоді $n > 1$, якщо $A < 1$, тоді $n < 1$.

Крок 2. Встановіть:

$$x_{k+1} = (1/n) * ((n-1)x_k + (A/x_k^{n-1}));$$

Повторюйте крок 2, доки не буде досягнута необхідна точність, тобто:

$$\text{root}(A, n) = \lim(k \rightarrow \infty) x_k.$$

Цей алгоритм (для квадратного кореня) також відомий як метод Герона, на честь грецького математика першого століття Герона Александрійського, який дав перший явний опис методу у своїй роботі “Метрика” 60 року нашої ери.

Нижче наведений приклад реалізації алгоритму мовою JavaScript для знаходження кореня заданого степеня n з числа x:

```
// Функція для знаходження кореня заданого степеня числа за алгоритмом Герона
function nthRootHeronsMethod(x, n, epsilon) {
    if (x < 0 && n % 2 === 0) {
        return NaN; // Для парних степенів корінь від'ємного числа не існує в області дійсних чисел
    }

    // Початкове наближення
    let guess = x / 2;

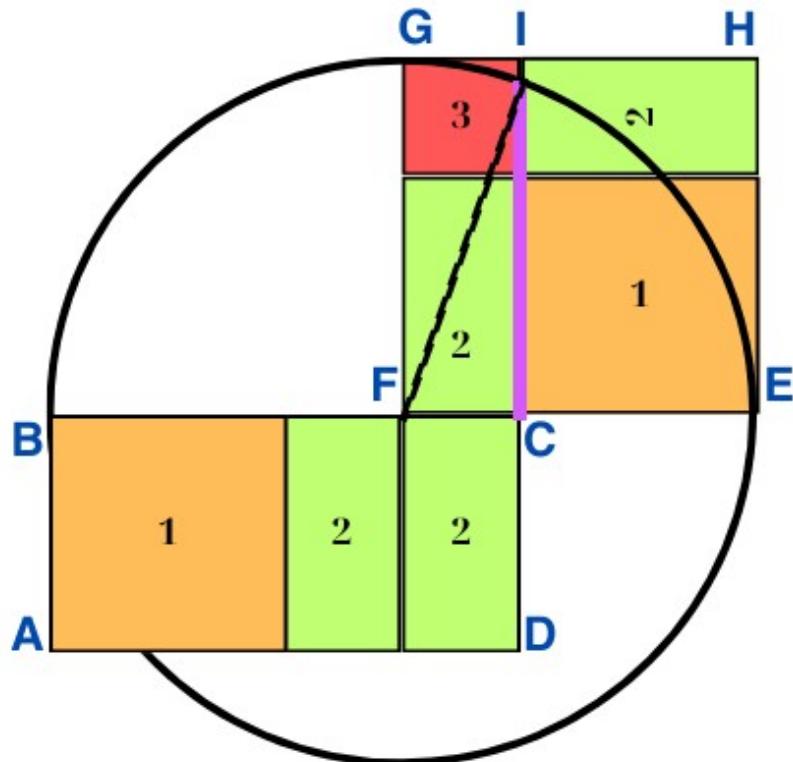
    // Ітерація до досягнення необхідної точності
    while (Math.abs(Math.pow(guess, n) - x) > epsilon) {
        // Оновлення значення за формулою Герона для кореня заданого степеня
        guess = ((n - 1) * guess + x / Math.pow(guess, n - 1)) / n;
    }

    return guess;
}

// Приклад використання
const numberToFindRoot = 8;
const degree = 3;
const epsilonValue = 0.0001;

const result = nthRootHeronsMethod(numberToFindRoot, degree, epsilonValue);
console.log(`Корінь степеня ${degree} з числа ${numberToFindRoot}: ${result}`);
```

Приклад із “Елементів” Евкліда, як можна побудувати квадрат, рівний за площею даному прямокутнику ABCD. На малюнку лінія CI є стороною потрібного квадрата. Метод побудови описаний в 14 пропозиції другої книги “Елементів” Евкліда.



Необхідно побудувати квадрат площею, що дорівнює прямокутнику ABCD. Для цього доповнююмо лінію BC лінією CE, яка дорівнює лінії CD. Намалюйте півколо на лінії BE, таким чином, BE є діаметром кола. Визначте центр кола F. Проведіть пряму CI так, щоб точка I опинилася на колі, а пряма CI була перпендикулярна до BE. $BF = FI$, через те, що обидва вони є радіусами кола. Площа прямокутника ABCD дорівнює $(BF \cdot BA) + (FC \cdot BA) = FG^2 - FC^2$. Але $FG^2 = FI^2$, а $FI^2 = FC^2 + CI^2$, отже $(FG^2 - FC^2) = CI^2$. Площа прямокутника ABCD дорівнює CI^2 . Також $\sqrt{(ABCD)} = CI$.

Декарт у своїй “Геометрії” (1637) використовує цю теорему Евкліда для знаходження квадратних коренів геометричним методом. У наведеному вище випадку ми знайшли квадратний корінь з числа $\sqrt{(ABCD)} = CI$. Тут ABCD позначає сам прямокутник і його площу, тобто $ABCD = AB * BC$, де AB = висота CD, AD = ширина прямокутника BC. Якщо $AB = 1$, то $\sqrt{(ABCD)} = \sqrt{BC} = CI$.

Теорема. $a^2/b^2 = (a/b)^2$; (Також $a^n/b^n = (a/b)^n$);

Доказ.

$a/b^2 = (a/b)/b$, тому що $a/b^2 = m$;

$b^2 * m = a$;

$(a/b)/b = n$;

$b * n = (a/b)$;

$b^2 * n = a$;

$m = n$;

$a/b^2 = m = (a/b)/b = n$;

Також,

$a^2/b = (a/b)*a$, тому що $a^2 = a + a + \dots + a$; $a^2/b = a / b + a / b + \dots + a / b$;

Таким чином,

$a^2/b^2 = ((a/b)*a)/b$;

$((a/b)*a)/b = ((a/b)/b)*a = (a/b) * (a/b)$;

Отримуємо, що

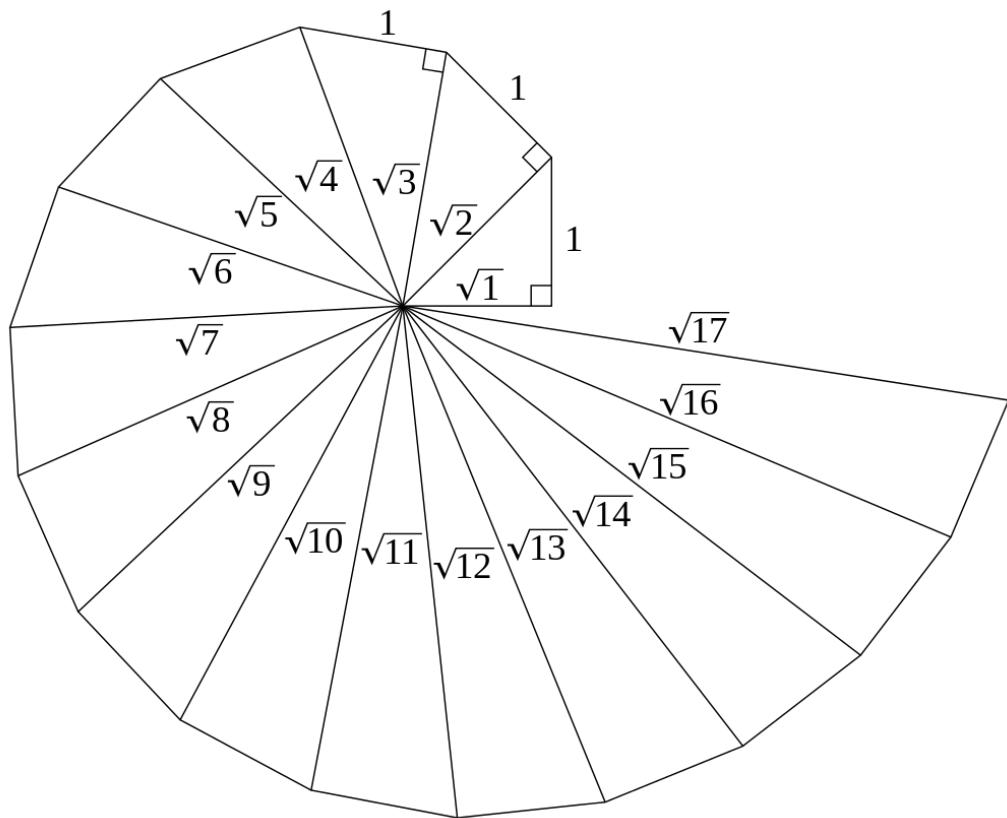
$a^2/b^2 = (a/b)^2$, або $\sqrt{(a^2/b^2)} = (a/b)$;

По індукції доводимо, що

$a^n/b^n = (a/b)^n$;

Ця формула присутня в астрономічних законах Кеплера.

Спіраль Теодора

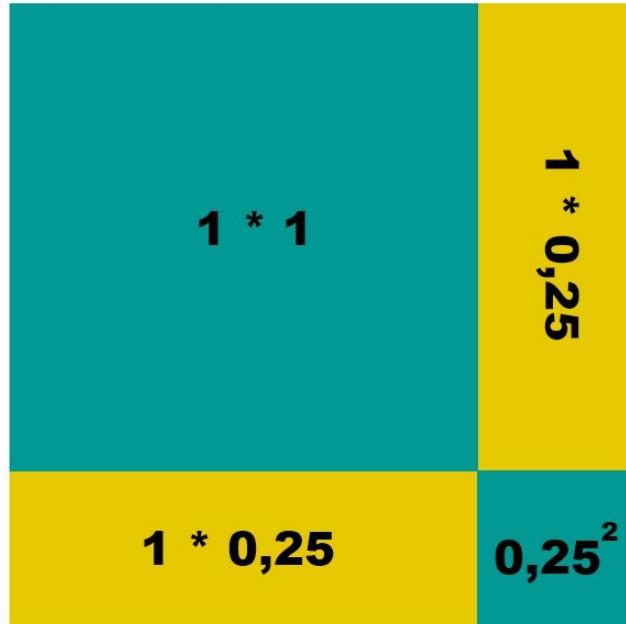


(Феодор) Теодор Кіренський довів іrrаціональність коренів натуральних чисел до 17 (виключаючи, звісно, точні квадрати - 1, 4, 9 і 16).

Якщо квадратний корінь з числа є цілим числом, це число називається точним квадратом, або квадратним. Якщо ціле число не є точним квадратом, тоді його корінь квадратний є іrrаціональним числом.

Будь-яке натуральне число (крім нуля) помножене саме на себе дає квадратне число. Припустимо у нас є раціональне число m,n , тобто ціле число з дрібною частиною n (наприклад, 2,5). Воно не може помножене саме на себе дати квадратне число, бо $(m,n)^2 = m^2 + 2(m * 0,n) + (0,n)^2$;

$$1,25 * 1,25 =$$



Квадрат нецілого раціонального числа дасть неціле раціональне число. Тому корінь квадратний з цілого числа дасть ціле число, або ірраціональне.

Теорема. Квадрат нецілого раціонального числа дасть неціле раціональне число.

Скористаємось визначенням: $\text{pow}(x:y, z) = (\text{pow}(x, z):\text{pow}(y, z))$;

$$(x + n/m)^2 = x^2 + 2(x * (n/m)) + (n/m)^2.$$

Якщо $x = 1$.

$$(x + n/m)^2 = 1 + 2(n/m) + (n/m)^2 = 1 + (2n)/m + n^2/m^2.$$

Тобто,

$$(x + n/m)^2 = 1 + (2nm+n^2)/m^2.$$

Дріб є цілим числом, якщо він записаний:

$$(qm+m)/m.$$

Щоб $(2nm+n^2)/m^2$ дало ціле число, потрібно, щоб було вірно:

$$n^2 = m^2.$$

Але це можливо, тільки при $m = n$. Тобто операція $(x + n/m)^2$ дасть ціле число тільки якщо n/m також ціле.

Китайська теорема про остачі

Уявіть людину, яка намагається розкласти скінчену кількість горіхів на менші однакові купи. Коли вона складає горіхи в купи по п'ять горіхів, то залишається 4 горіхи, тому що вони не складають повної купи. Коли вона намагається розкласти всі горіхи в купи по 4 горіхи, то залишиться 3 горіхи, які не утворюють повної купи. Якщо спробувати скласти купи по 7, то залишиться 2 горіхи, якщо по 9, то залишиться 6. Яка загальна кількість горіхів?

Числа 4, 5, 7, 9 є взаємно простими числами.

Взаємно прості числа — натуральні або цілі числа, які не мають спільних дільників більших за 1, або, інакше кажучи, якщо їх найбільший спільний дільник дорівнює 1.

Ділення з остачею позначається знаком %, наприклад, $5\%4=1$, $4\%2=0$.

Число x , яке є відповідю на вище описану задачу, повинно мати певні властивості, а саме:

1. При діленні x з остачею на 5 залишається 4, тобто $x \% 5 = 4$.
2. При діленні x з остачею на 4 ми отримуємо остачу 3, тобто $x \% 4 = 3$.
3. При діленні x з остачею на 7 ми отримуємо остачу 2, тобто $x \% 7 = 2$.
4. При діленні x з остачею на 9 ми отримуємо остачу 6, тобто $x \% 9 = 6$.

Крім того,

5. Логічно слідує, що число x повинно бути сумою чисел: $n + m + v + w$, які мають наступні властивості:

$$n \% 5 = 4, n \% 4 = 0, n \% 7 = 0, n \% 9 = 0;$$

також

$$m \% 5 = 0, m \% 4 = 3, m \% 7 = 0, m \% 9 = 0;$$

також

$$v \% 5 = 0, v \% 4 = 0, v \% 7 = 2, v \% 9 = 0;$$

також

$$w \% 5 = 0, w \% 4 = 0, w \% 7 = 0, w \% 9 = 6.$$

Тобто кожне з чисел n , m , v , w ділить всі числа без остачі, крім одного. Щоб знайти числа n , m , v , w спочатку знайдемо найменші спільні кратні кожних 3 чисел, наступним чином:

$$4 * 5 * 7 = 140.$$

$$5 * 7 * 9 = 315.$$

$$4 * 7 * 9 = 252.$$

$$4 * 5 * 9 = 180.$$

Отримані числа не діляться націло на число з набору (4, 5, 7, 9), яке не було множником при їх отриманні.

Для того, щоб отримати числа $n + m + v + w = x$, знайдемо число при множенні на яке число 140 при діленні на 9 буде мати остачу 1, це буде число 2, бо $(140 * 2) \% 9 = 1$. Отже, число $w = 140 * 2 * 6 = 1680$, де 6 остача згідно з умовою задачі. Знайдемо число, яке при множенні на 315 дасть число, що при діленні з остачею на 4 дасть 1, а саме, це буде число 3. Тобто $m = 315 * 3 * 3 = 2835$.

Те саме зробимо для чисел 7 і 5, отримаємо числа:

$$n = 252 * 3 * 4 = 3024,$$

$$v = 180 * 3 * 2 = 1080.$$

Отримаємо $x = 1680 + 2835 + 3024 + 1080 = 8619$.

$$8619 \% 4 = 3.$$

$$8619 \% 5 = 4.$$

$$8619 \% 7 = 2.$$

$$8619 \% 9 = 6.$$

Одна з відповідей на задачу буде 8619, але щоб знайти найменшу відповідь потрібно поділити діленням без остачі це число на найменше спільне кратне всіх чотирьох чисел, а саме, на $4 * 5 * 7 * 9 = 1260$.

$$x = 8619 \% 1260 = 1059.$$

Число 1059 відповідає умові задачі.

Пробне ділення є трудомістким, але найлегшим для розуміння серед алгоритмів ціличисельної факторизації, тобто розкладу натурального числа на прості множники. Пробне ділення вперше описав Фібоначчі у своїй праці "Книга абака" (1202). Основна ідея тестів пробного ділення в тому, щоб побачити, чи можна ціле число n , яке потрібно розкласти на множники, по черзі поділити на кожне (просте) число, яке менше n .

Наприклад, для цілого числа $n = 12$ єдиними числами, які його ділять є 1, 2, 3, 4, 6, 12.

Вибір лише найбільших степенів простих чисел у цьому списку дає що $12 = 3 * 4 = 3 * 2 * 2$.

Розкладання числа n на множники

% означає ділення з остачею. % - ділення з остачею. $5 \% 2 = 1$; $4 \% 2 = 0$;

Початок.

1. $n = 84$, $p = 2$;
2. $n \% p = 0$; (2)
3. $n / p = 42$, тоді $n = 42$.
4. $n \% p = 0$; (2)
5. $n / p = 21$, тоді $n = 21$.
6. $n \% p = 1$, тоді $p = 3$;
7. $n \% p = 0$; (3)
8. $n / p = 7$, тоді $n = 7$;
9. $n \% p = 1$, тоді $p = 5$;
10. $n \% p = 1$, тоді $p = 7$;
11. $n \% p = 0$; (7)
12. $n / p = 1$;

Кінець.

Результат:

$$84 = 2 * 2 * 3 * 7;$$

$$84 = 2^2 * 3 * 7;$$

Основна теорема арифметики стверджує:

Кожне натуральне число $n > 1$ можна подати у вигляді $n = p_1 * p_2 * \dots * p_k$, де p_1, p_2, \dots, p_k — прості числа, причому таке подання єдине, аж до порядку множників.

Що таке аналітична геометрія?

Аналітична геометрія — це галузь математики, яка об'єднує алгебру та геометрію.

В аналітичній геометрії водиться простір як базове поняття, а всі інші об'єкти як точки, лінії, фігури являються тільки підмножинами цього простору. Наприклад, точка (x, y) , відрізок $[a, b]$.

Аналітичну геометрію почали активно використовувати з 17 століття. Рене Декарт в своїй роботі "Геометрія" (1637) спробував об'єднати алгебру та геометрію, використовуючи координати для опису точок у просторі.

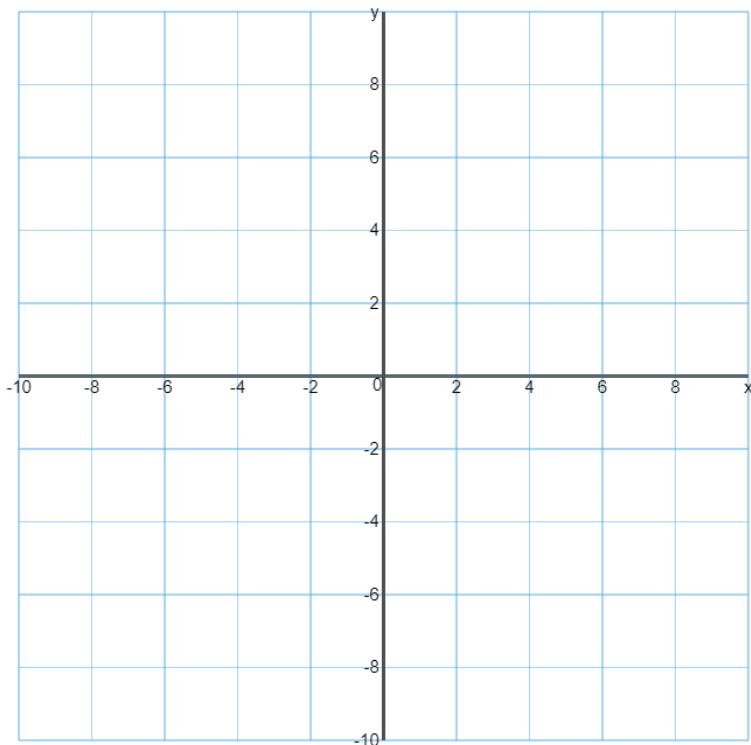
Аналітична геометрія дала змогу задавати фігури з допомогою функцій.

Вона дозволяє виразити геометричні поняття та відносини за допомогою алгебраїчних виразів та рівнянь. Ключовим елементом є використання координат для представлення точок, ліній, площин та інших геометричних об'єктів у просторі.

Наприклад, на площині точку можна представити парою чисел (x, y) , де x - це координата по горизонталі, а y - по вертикалі. Лінії та площини можна виразити рівняннями вигляду $Ax + By = C$.

Системи координат, зокрема сферичну систему координат, використовувати вже давньогрецькі астрономи. Прямокутну систему координат назвали Декартова система координат, хоча вона використовувалася до часів Рене Декарта.

Декартова система координат



Рівняння кола має вигляд $(x - a)^2 + (y - b)^2 = r^2$, де a і b — координати центру (a, b) , а r — радіус.

Рівняння лінії має вигляд $y = mx + b$, де m - це нахил (коєфіцієнт нахилу), а b - зсув по вісі y (ордината),

Косинус кута g між двома відрізками $[x_1, y_1, x_2, y_2]$ і $[x_3, y_3, x_4, y_4]$ обчислюється за формулою:

$$\cos(g) = ((x_2 - x_1) * (x_4 - x_3) + (y_2 - y_1) * (y_4 - y_3)) / (\sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)} * \sqrt{((x_4 - x_3)^2 + (y_4 - y_3)^2)});$$

Метричний простір

Метричний простір — це математична структура, що визначається парою (M, d) , де M - множина, а d - метрика, яка задає відстань між елементами цієї множини. Метрика d визначається як відображення, яке призначає кожній парі елементів з множини M деяке дійсне число (або, в деяких випадках, нескінченість), що задає їх відстань один від одного.

Метрика визначається наступними аксіомами:

1. Позитивність: Для будь-яких двох елементів x та y з метричного простору відстань між ними ($d(x, y)$) завжди не менше 0, і дорівнює нулю тільки в тому випадку, коли x і y співпадають.
2. Визначеність: Для будь-яких двох елементів x та y з метричного простору дистанція між ними завжди визначена, тобто не може бути нескінченності.
3. Симетричність: Для будь-яких двох елементів x та y з метричного простору відстань між ними дорівнює відстані між y та x . Формально: $d(x, y) = d(y, x)$.
4. Нерівність трикутника: Для будь-яких трьох елементів x , y та z з метричного простору, відстань між будь-якими двома з них завжди менше або дорівнює сумі відстаней між першим та третім елементами. Формально: $d(x, z) \leq d(x, y) + d(y, z)$.

Які найпопулярніші алгоритми й формулі в геометрії та тригонометрії?

Аксіоми евклідової геометрії

На початку 20 століття впливову аксіоматику геометрії розробив Давид Гільберт. В середині двадцятого століття Альфред Тарський записав аксіоми евклідової геометрії формальною мовою. Польський математик Кароль Борсук у своїй книзі "Фундамент геометрії" (1960) виводить безліч теорем геометрії Евкліда та Лобачевського на основі аксіоматики Тарського.

Аксіоматика Тарського має багато плюсів, особливо для комп'ютерних розрахунків, але ми опишемо аксіоматику Евкліда, бо вона найкоротша, але при цьому потужна.

Аксіоми геометрії з "Елементів" Евкліда:

1. Між будь-якими двома точками можна провести лінію.
2. Пряму лінію можна продовжувати до нескінченості.
3. Навколо будь-якої точки можна описати коло.
4. Усі прямі кути рівні один одному (слідує з визначення прямого кута).
5. Через точку, що лежить поза прямою, можна провести не більше однієї прямої, що паралельна даній (Аксіому у цій формі сформулював Прокл, але ця аксіома слабша ніж оригінальна аксіома Евкліда. Варто використовувати Аксіому паралельності Евкліда, якщо працюєте з оригінальним набором аксіом.)

Аксіома паралельності Евкліда: Якщо пряма, що перетинає дві прямі, утворює внутрішні односторонні кути, які менші ніж два прямі кути, то ці дві прямі, продовжені необмежено, зустрінуться з тієї сторони, де кути менші за два прямі.

Аксіома Евкліда не виконується в геометрії Лобачевського та сферичній геометрії.

П'ятий постулат про паралельні у формі математика Прокла формулюється наступним чином: "Через точку, що лежить поза прямою, можна провести лише одну пряму, паралельну даній". П'ятий постулат у версії Евкліда є конструктивним і дає зрозуміти, яка пряма буде паралельною до даної, а у версії Прокла це не відомо. Якщо ми замінимо аксіому (постулат) Евкліда про паралельні на аксіому Прокла про паралельні, тоді нам потрібно буде додати в аксіоматику аксіому Паша, бо в аксіомі Прокла не уточнено яка саме лінія буде паралельна даній, на відміну від аксіоми Евкліда, в якій сказано, якщо лінія перетинає дві прямі та утворює по один бік два прямі кути (в сумі 180 градусів), тоді ці дві лінії паралельні.

Крім геометричних постулатів (аксіом) Евклід описує аксіоми для пропорцій, наприклад:

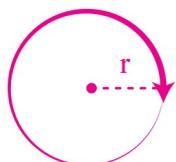
1. Ціле більше частини (Аксіома не виконується для нескінчених множин).

2. Рівні до одного й того ж, рівні між собою.

Формули площин

Geometric Formulas

Circumference of a circle:



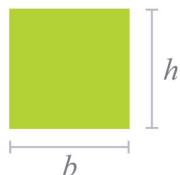
$$C = 2\pi r$$

Area of a circle:



$$A = \pi r^2$$

Area of a square:



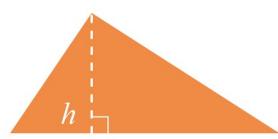
$$\begin{aligned} A &= hb \\ A &= h^2 \\ A &= b^2 \end{aligned}$$

Area of a rectangle:



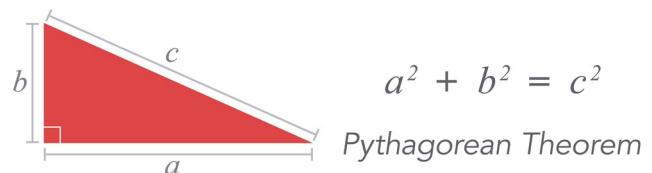
$$A = hb$$

Area of a triangle:



$$A = \frac{1}{2}hb$$

Area of a right triangle:

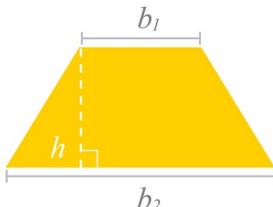


Area of a parallelogram:



$$A = hb$$

Area of a trapezoid:

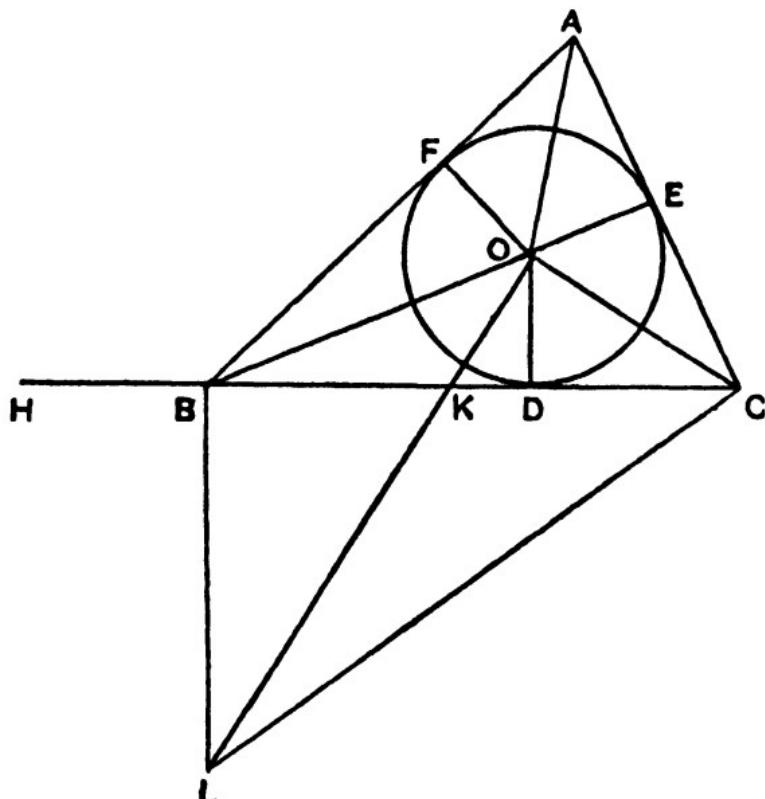


$$A = \frac{1}{2}h(b_1 + b_2)$$

Формула Герона

У своїй праці "Метрика" Герон доводить формулу, що дає можливість по сторонах трикутника знайти його площину. Формула Герона стверджує, що площа трикутника, сторони якого мають довжини a , b і c , дорівнює: $A = \sqrt{s(s - a)(s - b)(s - c)}$, де s — півпериметр трикутника, тобто $s = (a + b + c)/2$.

Доказ формули Герона.



Нехай відома довжина сторін трикутника ABC . Впишіть в нього коло DEF і нехай O буде його центром. Об'єднайте AO , BO , CO , DO , EO , FO . (D , E , F — точки в яких коло торкається трикутника).

Відповідно, $BC * OD = 2\Delta BOC$,

$CA * OE = 2\Delta COA$,

$AB * OF = 2\Delta AOB$,

звідки, шляхом додавання,

$p * OD = 2\Delta ABC$,

де p — периметр ΔABC .

Проведіть CB до H , щоб $BH = AF$.

Звідси, $AE = AF$, $BF = BD$, і $CE = CD$, ми маємо $CH = (1/2)p = s$.

Тому $CH * OD = \Delta ABC$.

Але, $CH * OD$ є ‘стороною’ добутку $CH^2 * OD^2$, тобто $\sqrt{(CH^2 * OD^2)}$, відтак, $(\Delta ABC)^2 = CH^2 * OD^2$.

Намалюйте OL під прямим кутом до OC, перетинаючи BC в точці K, і BL під прямим кутом до BC, що зустрічає OL в L. Об'єднайте CL. Тоді, оскільки кожен із кутів COL, CBL прямий, COBL є чотирикутником у колі.

Звідси маємо $\angle COB + \angle CLB = 2R$ (два прямих, тобто 180 градусів).

Але $\angle COB + \angle AOF = 2R$, оскільки AO, BO, CO ділять кути навколо O навпіл, а кути COB, AOF разом дорівнюють кутам AOC, BOF, а сума всіх чотирьох кутів дорівнює $4R$ (тобто 360 градусів). Отже, $\angle AOF = \angle CLB$.

Таким чином, прямокутні трикутники AOF, CLB подібні, тому

$$BC/BL = AF/FO = BH/OD,$$

$$\text{i, очевидно, } CB/BH = BL/OD = BK/KD;$$

$$\text{Отримуємо, } CH/HB = BD/DK.$$

Звідси випливає, що

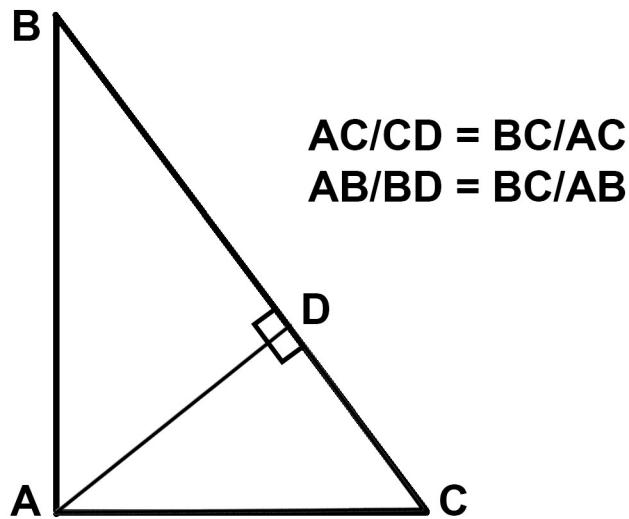
$$CH^2/CH*HB = BD*DC/CD*DK = BD*DC/OD^2, \text{ оскільки кут } COK \text{ прямий. Тому } (\Delta ABC)^2 = CH^2*OD^2 = CH*HB*BD*DC = s(s-a)(s-b)(s-c).$$

Теорема Піфагора

Теорема Піфагора в загальній формі доведена в "Елементах" Евкліда.

Теорема Піфагора

Якщо сторони прямокутного трикутника позначити, як $AB = a$, $CA = b$, $BC = c$, тоді $a^2 + b^2 = c^2$.



Доказ теореми Піфагора не методом Евкліда.

Цей доказ заснований на пропорційності сторін двох подібних трикутників, тобто на тому, що відношення будь-яких двох відповідних сторін подібних трикутників однакове незалежно від розміру трикутників. Нехай ABC являє собою прямокутний трикутник з прямим кутом (90°) в точці A .

Проведіть висоту з точки A і назвіть D її перетином зі стороною BC . Точка D ділить довжину гіпотенузи c на частини d і e . Новий трикутник CAD подібний до трикутника ABC , оскільки вони обидва мають прямий кут (за визначенням висоти), і вони мають спільний кут у C , отже, що третій кут також буде однаковим в обох трикутниках, позначених як f . За аналогічним міркуванням трикутник ABD також подібний до ABC . Подібність трикутників призводить до рівності відношень відповідних сторін: $AC/CD = BC/AC$ і $AB/BD = BC/AB$. Перший результат прирівнює косинуси кутів f , тоді як другий результат прирівнює їх синуси. Ці співвідношення можна записати так $AC^2 = BC * CD$ і $AB^2 = BC * BD$ через властивість пропорцій. Результатом підсумування цих двох рівностей є $AC^2 + AB^2 = (BC * CD) + (BC * BD) = BC * (CD + BD) = BC^2$, яка після спрощення виражає теорему Піфагора: $AC^2 + AB^2 = BC^2$.

Існує формула для знаходження піфагорових трійок, тобто чисел, що задовольняють теорему Піфагора, наприклад, $3,4,5$, бо $3^2 + 4^2 = 5^2$.

Якщо (a, b, c) – піфагорова трійка, то і (ka, kb, kc) – піфагорова трійка для будь-якого натурального числа k . Наприклад, $(3, 4, 5)$ та $(6, 8, 10)$. Примітивна піфагорова трійка – це трійка, в якій a , b і c взаємно прості (тобто вони не мають загального дільника більше 1). Існує 16 примітивних піфагорійських трійок чисел до 100: $(3, 4, 5)$, $(5, 12, 13)$, $(8, 15, 17)$, $(7, 24, 25)$, ...

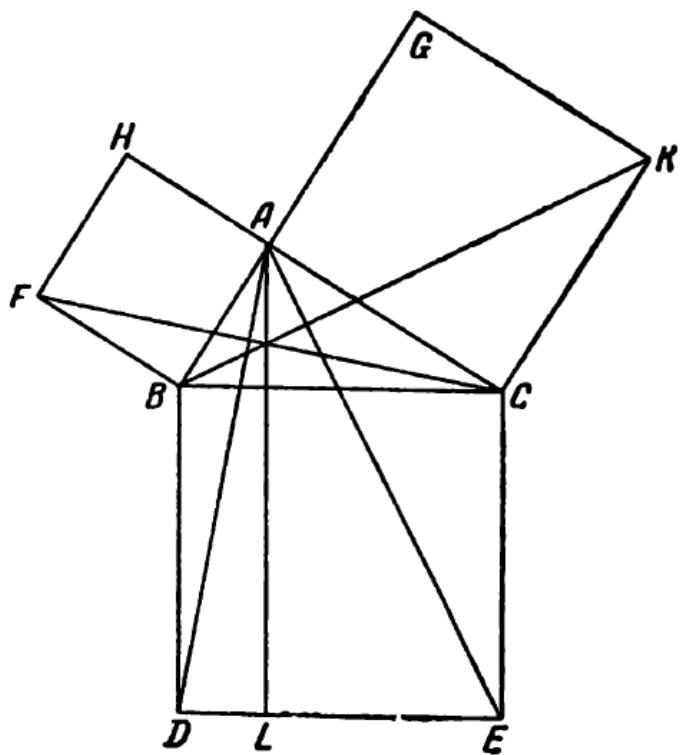
Піфагорові трійки вмів знаходити Евклід.

Доказ теореми Піфагора методом Евкліда

Для доказу теореми Піфагора, нам ще знадобиться теорема про те, що трикутники, які мають однакову довжину основи й висоту, як наслідок, мають однакову площину. З цієї теореми логічно слідує, що прямокутник, який має однакову з трикутником довжину основи та однакову висоту, має вдвічі більшу площину ніж цей трикутник.

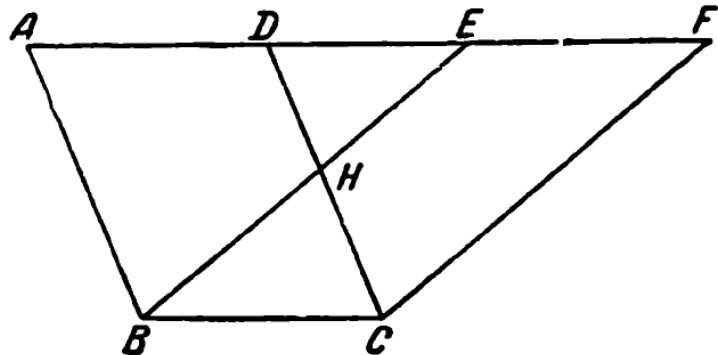
Для побудови прямих кутів, паралельних ліній, відрізка, що рівний даному, Евклід використовує кола та їх перетини з іншими колами, чи лініями.

Доказ теореми Піфагора з 47 пропозиції першої книги “Елементів” Евкліда.



Трикутник ВКС рівний трикутнику АЕС. Через те, що трикутник ВКС має спільну основу з квадратом AGKC і також однакову висоту, його площа рівна половині площині квадрата, тобто $\text{ВКС} * 2 = \text{ВКС}$. Подібний розсуд проводимо для трикутника АЕС і прямокутника LC, тобто $\text{АЕС} * 2 = \text{LC}$. Отже, прямокутник LC рівний за площею квадрату AGKC, бо трикутник ВКС рівний трикутнику АЕС. Оскільки трикутники FCB і ADB також рівні, використовуючи вище описану логіку, ми отримуємо, що квадрат FHAB рівний за площею прямокутнику BL. Отже, квадрат BCDE рівний за площею квадрату AGKC і FHAB.

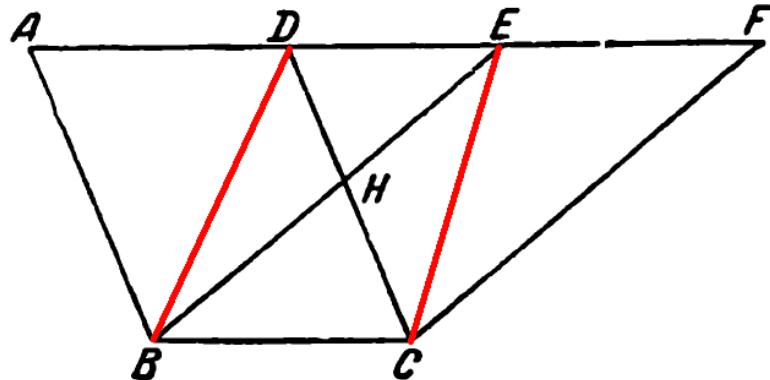
Пропозиція 35 з першої книги твору Евкліда: Паралелограмми, які розташовані на одній основі й між тими ж паралельними, рівні між собою за площею.



Доказ.

Через те, що фігура ABCD є паралелограмом, пряма AD рівна прямій BC. За подібним розсудом, виходить, що пряма EF рівна прямій BC. Очевидно, з умови теореми, що пряма AD рівна EF і пряма AE рівна DF. Очевидно, що пряма AB рівна DC. Отимуємо, що EA плюс AB рівні по довжині FD і DC, а кут FDC рівний куту EAB. Значить, EB рівна FC, а трикутник EAB рівний трикутнику DFC. Віднімаємо загальну частину DHE, отимуємо, як залишок, трапецію ABHD, яка рівна трапеції EHCF. Додамо спільний трикутник HBC, значить, весь паралелограм ABCD рівний всьому паралелограму EBCF.

На основі вище доведеної теореми 35, Евклід доводить теорему номер 37 про те, що трикутники з однаковою основою і висотою мають однакову площину. Трикутники, що розташовані на однаковій основі та між тими ж паралельними, є однаковими по площі.



Трикутники BDC і трикутник BEC рівні за площею, бо ділять однакові за площею паралелограми ADBC і EFBC навпіл.

Число пі

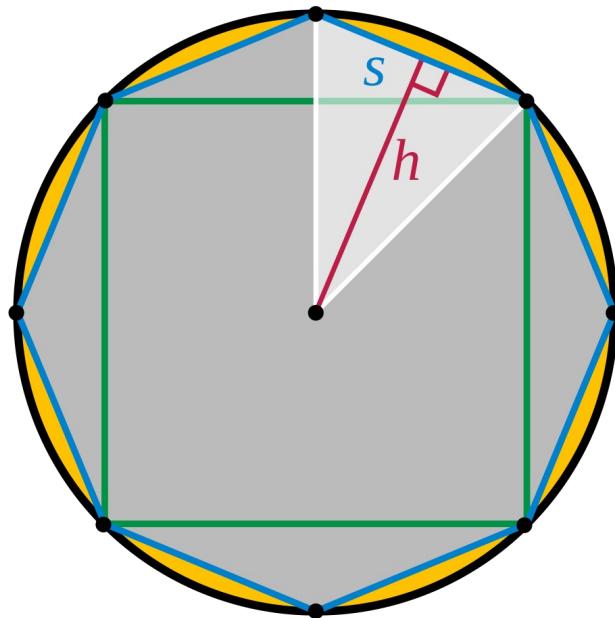
Архімед один з пionерів математичного аналізу. Ключовими поняттями математичного аналізу є ліміт послідовності, числовий ряд, інтеграл, похідна функція.

Архімед у своїй праці "Вимірювання кола" знайшов формулу периметра і площині кола.

Архімед вписав в коло правильний багатокутник в якого 96 сторін і кутів. Площу і периметр будь-якого правильного багатокутника можна знайти ділячи його на трикутники. Якщо помножити висоту трикутника на величину його основи ви отримаєте його площину.

Архімед точно встановив перші три знаки числа Пі, а саме: 3,14. Число Пі є периметром кола з діаметром одиниця. Архімед побачив, що при збільшенні числа сторін вписаного багатокутника цифри 3,14 залишаються не змінні, змінюються інші розряди числа.

Архімед не тільки вписав багатокутник, а й описав багатокутник навколо кола, чим встановив верхній і нижній ліміт числа Пі. Периметр — це довжина контуру фігури. Периметр кола знаходиться за формулою: $2\pi r$, тобто Пі помножити на два і на радіус кола. Площа кола встановлюється за формулою: πr^2 . Наприклад, коло з радіусом два, буде мати площину приблизно $3,14 * 2^2$, тобто 12,56 сантиметрів квадратних.



Архімед у своїй праці "Про сферу і циліндр" визначив формулі площині та об'єму циліндра, сфери та конуса.

Збірку праць Архімеда англійською мовою видав британський історик науки Томас Хіт в 1897 році. Те, що об'єм конуса становить $1/3$ об'єму циліндра довів Евклід в 10 пропозиції 12 книги збірки "Елементи".

Циліндр — це тіло обертання, отримане при обертанні прямокутника навколо його сторони. Площа циліндра встановлюється за формулою: $(\pi r^2) * h$, тобто площа основи циліндра (кола) помножена на висоту циліндра.

Площа поверхні циліндра встановлюється за формулою: $2*(\Pi * r^2) + 2*\Pi * r * h$, де h — висота циліндра, r — радіус основи циліндра.

Якщо з однієї точки А провести пучок прямих так, щоб вони охоплювали всі точки периметра кола В, до яких не входить точка А, то така фігура називається конусом, де точка А — вершина конуса, коло В — основа конуса. Пряний конус можна розгорнути у двовимірну фігуру та отримати сектор кола. Одна радіана — це дуга кола, що дорівнює його радіусу по довжині. Якщо відомий радіус основи конуса і його висота, тоді можна по теоремі Піфагора визначити похилу l , що утворює конус і є гіпотенузою прямокутного трикутника з катетами r і h , де r є радіусом основи конуса, h його висотою. Таким чином площа конуса буде рівна сектору кола з радіусом l і кутом, що дорівнює $2*\Pi * r / l$ радіан. Коло ділиться на 360 градусів і на 2Π радіан, тобто один градус дорівнює $(2\Pi / 360)$ радіан. Площа сектора кола встановлюється за формулою: $1/2 * (a * r^2)$, де a - кут сектора в радіанах, r - радіус сектора.

Якщо радіус конуса 3, а його висота 4, тоді його твірна лінія буде мати довжину 5. Якщо цей конус розгорнути у двовимірний сектор кола, тоді кут сектора буде $6.28 * 3 / 5 = 3,768$ радіан, або приблизно 217 градусів. Площа цього конуса не враховуючи площину основи буде: $1/2 * (3,768 * 25)$, тобто 47 квадратних сантиметрів. Щоб дізнатись повну площину поверхні конуса потрібно врахувати площину його основи, тобто Πr^2 . Об'єм конуса та об'єм сфери можна встановити вписуючи в них стопку циліндрів, так щоб цилінди набували форму конуса, або сфери. У сферу можна вписувати багатокутних і так визначити приблизний об'єм сфери.

Формула площини поверхні сфери: $4*\Pi * r^2$. Формула об'єму сфери: $(4:3)*\Pi * r^3$. Об'єм конуса визначається за формулою: $(1:3)*(\Pi * r^2)*h$. Об'єм конуса з радіусом основи r і висотою h плюс об'єм сфери (кулі) з радіусом r дорівнює об'єму циліндра з радіусом r і висотою h .

Geometric Formulas: Volumes

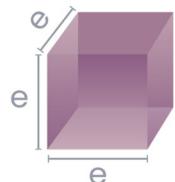
$$V = lwh$$



паралелепіпед

l = length w = width h = height

$$V = e^3$$



куб

e =edge

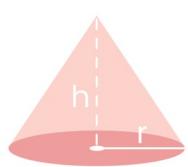
$$V = \pi r^2 h$$



циліндр

r = radius h = height

$$V = \frac{1}{3} \pi r^2 h$$



конус

r = radius h = height

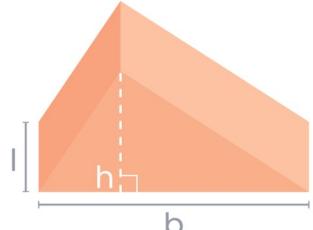
$$V = \frac{4}{3} \pi r^3$$



сфера

r = radius

$$V = \frac{1}{2} bhl$$



призма

b = base h = height l = length

До Ньютона математичний аналіз розвивали: Архімед, Кавальєрі, Ферма, Валліс.

Відома робота Валліса: “Арифметика нескінченно малих” (1655) і формула:

$$\pi/2 = 2/1 * 2/3 * 4/3 * 4/5 * 6/5 * 6/7 * 8/7 * 8/9 * \dots,$$

$$\text{або } \pi/2 = \prod_{n=1}^{\infty} ((2n)^2 / ((2n-1)(2n+1))$$

$$\pi \text{ або } \Pi = 3,14\dots$$

Формула Віста для π

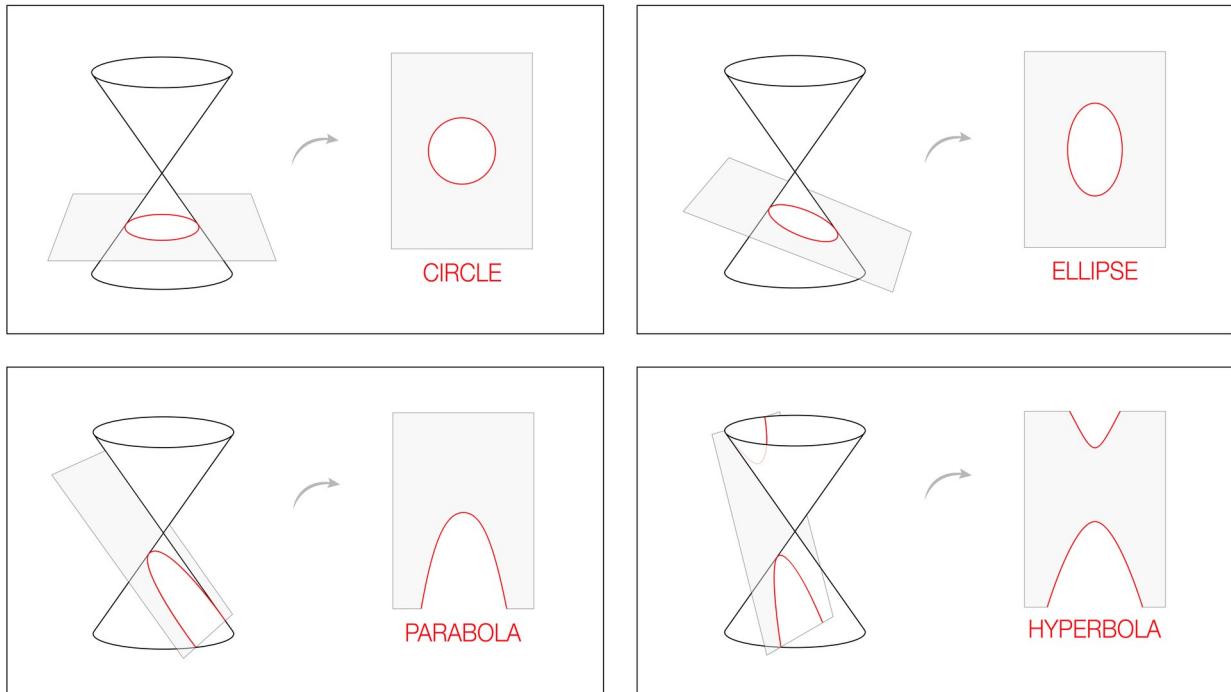
Лейбніц також відомий тим, що створив ряд Лейбніца, але, як виявилося, індійський математик Мадхава (XIV ст.) вже використовував подібний ряд, тому цей ряд називають рядом Мадхави-Лейбніца. (Ряд Лейбніца: $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + \dots$). Ряд сходиться відносно повільно, тому замість нього можна використовувати ряд Віста для числа Пі.

Формула Віста для Пі.

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2 + \sqrt{2}}}{2} \cdot \frac{\sqrt{2 + \sqrt{2 + \sqrt{2}}}}{2} \dots$$

Ці ряди були отримані в результаті аналізу методу Архімеда, що оснований на вписувані правильних багатокутників в коло.

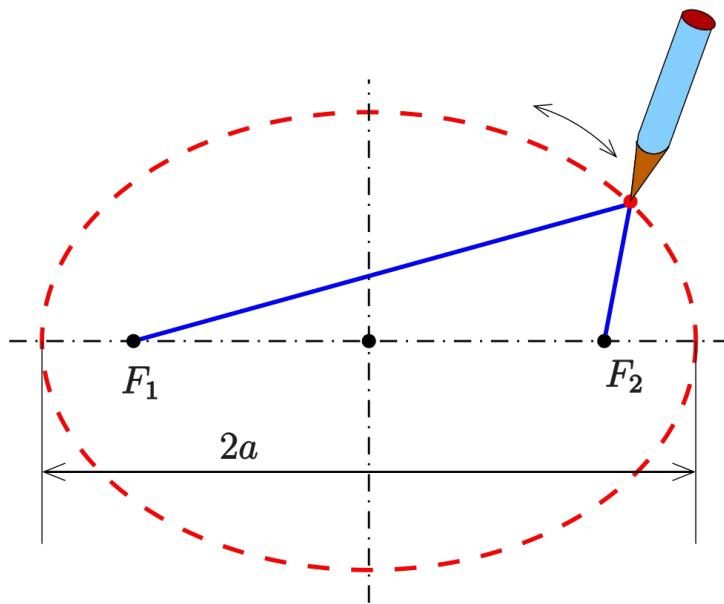
Зображення конічних перетинів.



Аполлоній і Евклід називали лінію, що має одну спільну точку з контуром конічного перетину "дотичною прямою". Якщо у двох місцях площини забити цвяхи та прив'язати до них нитку, а потім потягнути нитку олівцем і намалювати фігуру, то ми отримаємо еліпс. Спосіб малювання еліпса з допомогою нитки описаний в "Геометрії" (1637) Декарта.

Під час побудови звичайного кола з допомогою нитки, ми беремо нитку, фіксуємо її кінець у певній точці (яка буде центром кола), і, натягуючи нитку, малюємо коло.

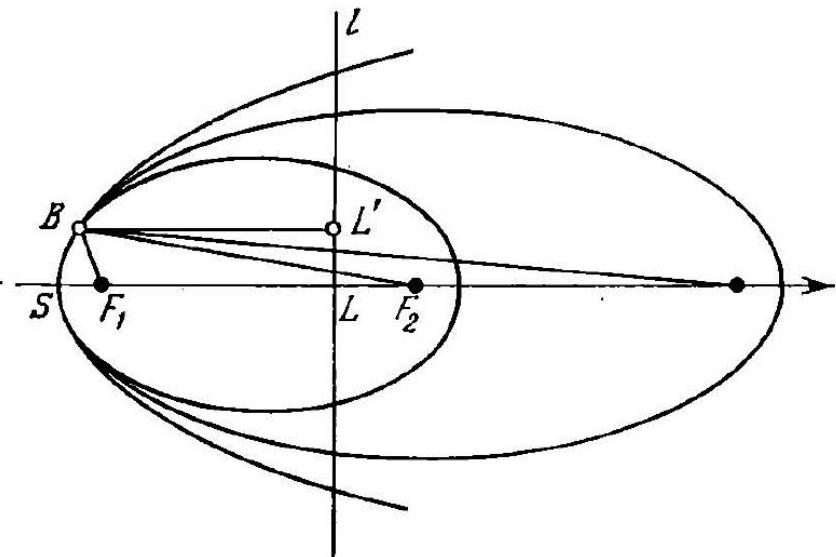
Якщо закріпити нитку не в одній, а у двох точках, то отримаємо криву, схожу на коло, яку називають еліпсом. Обидві точки кріплення нитки називаються фокальними точками еліпса.



Ретельно розглянув конічні перетини англійський математик Джон Валліс у своєму творі “Про конічні перетини” (1655).

З еліпса можна отримати параболу. Для цього залишимо зафікованим один фокус F_1 еліпса і найближчу вершину S еліпса. Вершини еліпса — точки перетину контуру еліпса з правою лінією, що сполучає його фокуси, тобто з великою піввіссю. Тепер ми розглянемо еліпси, отримані шляхом зміщення другого фокусу еліпса F_2 все далі й далі від точки F_1 , ці еліпси будуть йти все ближче й ближче до кривої, яка називається параболою.

Перетворення еліпсу в параболу.



Форма еліпса і ступінь його подібності до кола характеризуються відношенням $e = c / a$, де c — відстань від центра еліпса до його фокуса (фокусна відстань), а — велика піввісь. Величина e називається ексцентриситетом еліпса. Коли $c = 0$, а отже, $e = 0$, еліпс перетворюється на коло. Ексцентриситет — числови характеристики конічного перерізу, що показує ступінь його відхилення від кола. Ексцентриситет еліпса можна виразити через відношення малої (b) і великої (a) півосі: $e = \sqrt{1 - (b^2/a^2)}$.

Великою віссю еліпса називають його найбільший діаметр — відрізок, що проходить через центр і два фокуси. Велика піввісь становить половину цієї відстані та проходить від центру еліпса через фокус до його краю.

Під кутом 90° до великої півосі розташована мала піввісь — мінімальна відстань від центру еліпса до його краю.

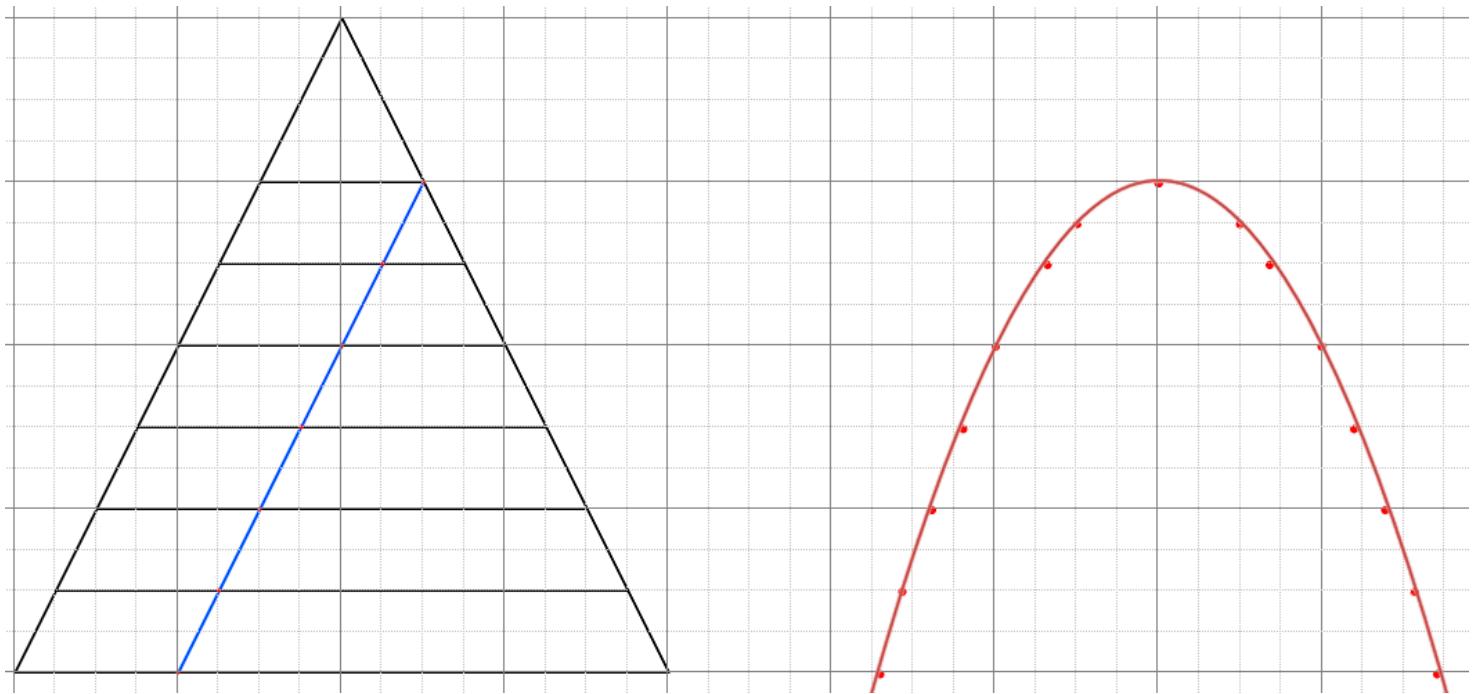
Ми можемо визначити конічний переріз виключно в термінах плоскої геометрії: це геометричне місце (множина) всіх точок P, відстань яких до нерухомої точки F (званої фокусом) можна отримати помноживши відстань від P до фіксованої лінії L (званої директрисою) на ексцентриситет e.

Ексцентриситет e — це числова характеристика конічного перетину, яка дорівнює відношенню відстаней: від будь-якої точки P конічного перетину до фокуса F та від цієї точки до директриси L. $e = d(P,F) / d(P,L)$;

Точки P параболи задані рівнянням $d(P,F) = d(P,L) * e$, де $e = 1$ (екскентриситет), $d()$ – відстань, L – директриса, F – фокус.

Фокальна відстань с - половина довжини відрізку, що з'єднує фокуси еліпса.

Ексцентриситет еліпса e характеризує його витягнутість і визначається відношенням фокальної відстані с до великої піввісі a, тобто до половини великої осі еліпса. Для еліпса ексцентриситет завжди буде $0 < e < 1$, для круга $e = 0$, для параболи $e = 1$, для гіперболи $e > 1$. Коло — це еліпс з рівними півосями, тому ексцентриситет кола дорівнює 0.



Побудова параболи.

Довжина хорди кола визначається за формулою: $\sqrt{(r^2 - h^2)}$, де r – радіус, h – відстань від центра кола до хорди.

Таким чином можна розрахувати точки параболи. На малюнку, відстань другої точки параболи від осі буде $\sqrt{(3^2 - 1^2)} = 2.8\dots$

Якщо ми маємо дві нерухомі точки A і B, то точки M, які задовольняють рівнянню $|AM| - |BM| = 0$, утворюють параболу. Один з описів параболи включає точку (фокус) і пряму (директрису). Фокус не лежить на директрисі параболи.

Парабола — це геометричне місце точок на площині, рівновіддалених як від директриси, так і від фокуса. Пряма, перпендикулярна до директриси та проходить через фокус (тобто пряма, що розділяє параболу через середину), називається "віссю симетрії".

Діаметр параболи — це лінія вздовж осі симетрії параболи від вершини параболи до її стику з основою конуса. Або діаметр параболи — це лінія, яка ділить всі хорди параболи навпіл. Точка, де парабола перетинає свою вісь симетрії, називається "вершиною".

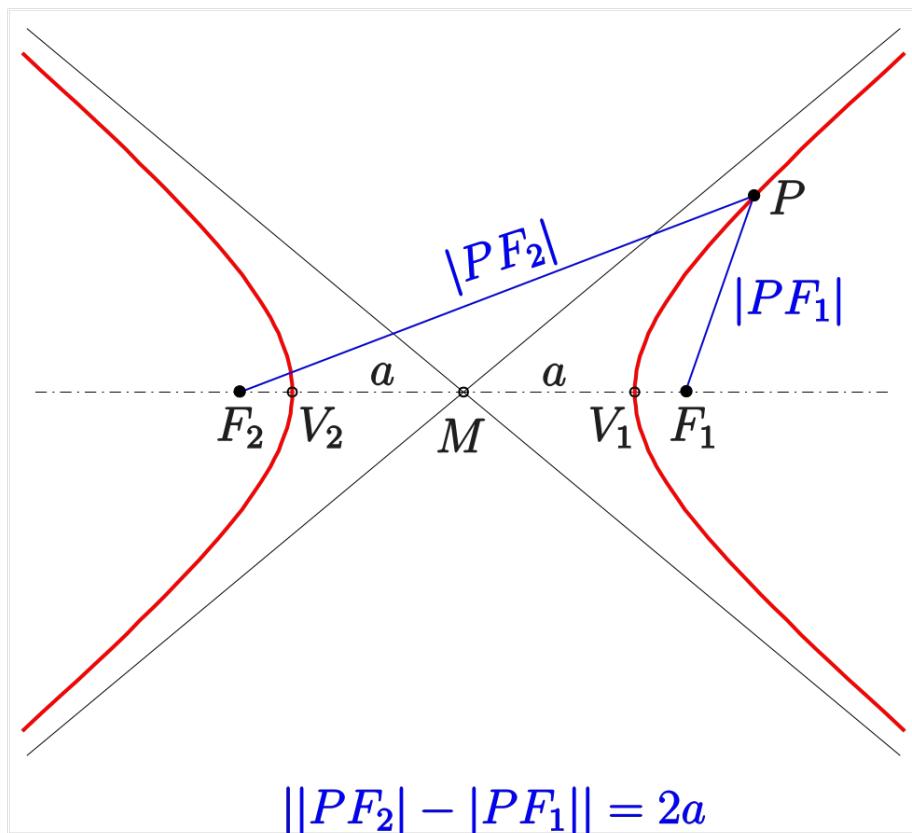
Хорда в планіметрії — відрізок, що з'єднує дві точки даної кривої (наприклад, кола, еліпса, параболи, гіперболи). Хорда знаходиться на січній прямій, що перетинає криву у двох або більше точках.

Архімед написав невеличкий твір під назвою "Квадратура параболи", в якому методом вичерпування знаходить площу параболи, тобто знаходить її площу, вписуючи в неї трикутники.

Кривина лінії в певній точці визначається по похідній другого порядку в цій точці.

Асимптота — це пряма з властивістю, що відстань від точки на кривій до цієї прямої прагне до нуля, коли точка рухається до нескінченності вздовж гілки. Вперше термін з'явився в Аполлонія Перського, хоча асимптоти гіперболи вивчав Архімед. Для гіперболи $y = 1/x$ асимптотами є осі абсцис і ординат. Крива може наблизятися до своєї асимптоти, залишаючись з одного боку від неї.

Зображення асимптоти гіперболи. Асимптоти гіперболи перетинаються в точці M на графіку.



Аполлоній у своєму творі доводить, що перетин конуса, який не є паралельний його основі не є колом. Твір Аполлонія про конічні перетини переклав на англійську мову Томас Хіт в 1896 році. Аполлоній довів також, що певні перетини конуса утворюють еліпс.

Можна довести, що замкнутий переріз конуса є еліпс (або коло, як окремий випадок еліпса).

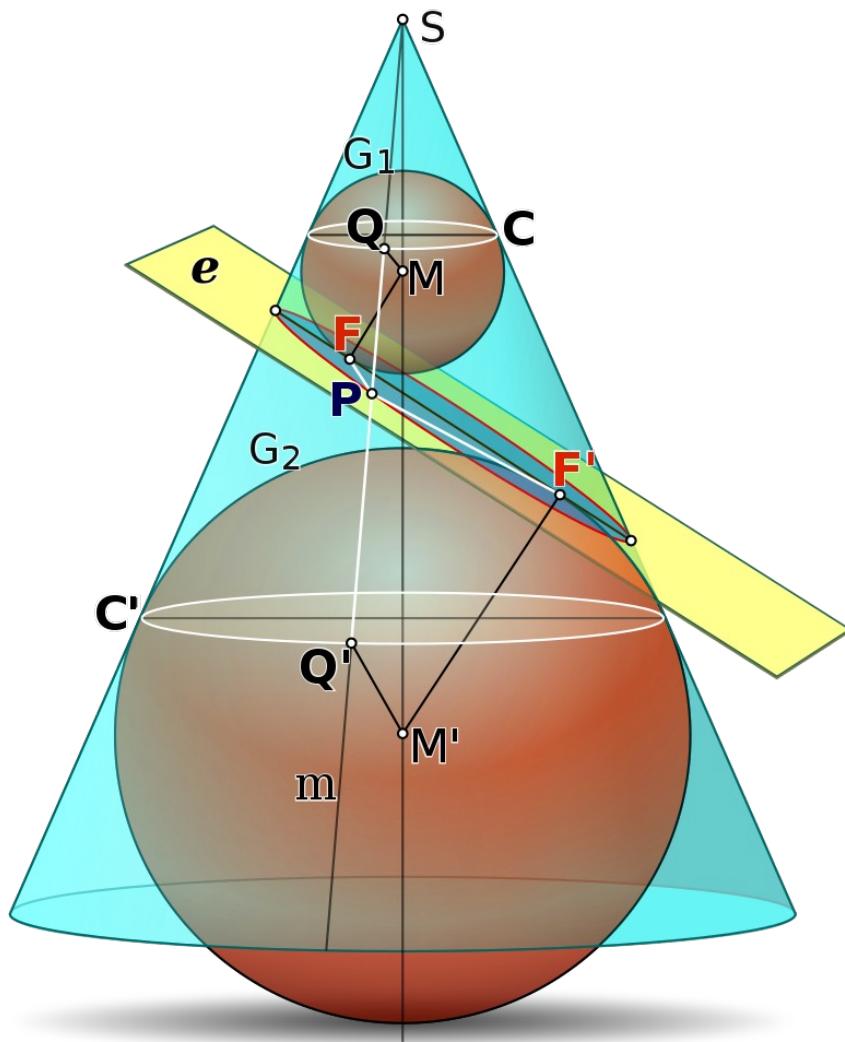
Це можна зробити з допомогою сфер Данделіна. Ми можемо показати, що замкнутий переріз конуса, тобто фігура без кінця, буде еліпсом. Ви можете пов'язати аналітичне визначення еліпса з конкретним перерізом конуса, використовуючи сфери Данделіна.

Сфери Данделіна

Ми можемо показати, що замкнутий переріз конуса, тобто фігура без кінця, буде еліпсом. Ви можете пов'язати аналітичне визначення еліпса з конкретним перерізом конуса, використовуючи сфери Данделіна.

Сфери Данделіна — це сфери, що беруть участь у геометричній конструкції, яка пов'язує планіметричне визначення еліпса, гіперболи та параболи через фокуси з їх стереометричним визначенням як переріз конуса. Сфери Данделіна запропонував бельгійський математик Жермінал Данделін у 1822 році.

Зображення сфер Данделіна



Розглянте ілюстрацію. Дві сфери Данделіна (G_1 і G_2) є дотичними як до площини, що перетинає конус, так і до самого конуса. Кожна сфера торкається конуса по колу. Точку дотику площини з G_1 позначимо через F_1 , а також для G_2 і F_2 . Нехай P — типова точка на C , де C — лінія (фігура), утворена площиною, що перетинає конус. Твердження: сума відстаней $d(P, F_1) + d(P, F_2)$ залишається постійною, коли точка P рухається вздовж кривої перетину C .

Пряма, що проходить через P і вершину S конуса, перетинає два кола, торкаючись G_1 і G_2 відповідно в точках P_1 і P_2 . Коли P рухається по кривій, P_1 і P_2 рухаються вздовж двох кіл, а їх відстань $d(P_1, P_2)$ залишається постійною. Відстань від P до F_1 така ж, як відстань від P до P_1 , оскільки відрізки PF_1 і PP_1 обидва дотичні до однієї сфери G_1 . Якщо прямі, що виходять з однієї точки, дотикаються до однієї і тієї ж сфери, то їх відстані від цієї точки до точок з'єднання зі сферою одинакові. За симетричним аргументом відстань від P до F_2 дорівнює відстані від P до P_2 . Отже, ми обчислюємо суму відстаней як $d(P, F_1) + d(P, F_2) = d(P, P_1) + d(P, P_2) = d(P_1, P_2)$, яка є постійною, коли P рухається вздовж кривої.

Формула площі Гаусса (формула шнурування)

Формула площі Гаусса (формула шнурування) — це математичний алгоритм для визначення площі простого багатокутника, вершини якого описуються декартовими координатами на площині.

Формула була описана Альбрехтом Людвігом Фрідріхом Мейстером (1724–1788) у 1769 році та базується на формулі трапеції, яку описав Карл Фрідріх Гаусс.

Простий багатокутник — це багатокутник без перетинів та дірок. Тобто, це пласка фігура, що складається з відрізків, які не перетинаються або сторін, які з'єднані попарно й утворюють замкнений шлях.

З допомогою формули трапеції Гаусса (за Карлом Фрідріхом Гауссом) можна обчислити площу простого багатокутника. Кожному краю багатокутника присвоюється трапеція, площа якої може бути додатною або від'ємною. Від'ємні частини поверхні компенсують частини додатних трапецій, що лежать поза багатокутником.

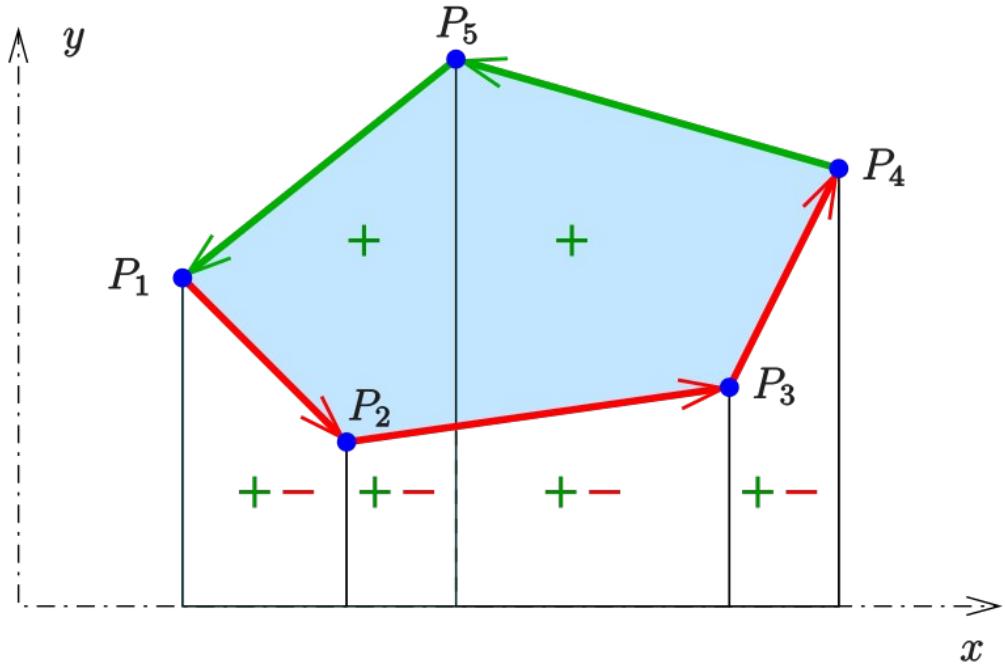
Трапеція (від гр. трапεζί — "столик") — це чотирикутник, дві протилежні сторони якого паралельні, а інші дві сторони — не паралельні.

Трапецію можна поділити на дві частинки паралелограма, отже площа трапеції буде $A = (a + b)/2 * h$, а площа паралелограма $(a * h)$, або $(b * h)$, де a та b основи трапеції, а h є її висотою.

Формула площі Гаусса

```
getAreaByShoelaceFormula([x1,y1, x2, y2, x3, y3, x4, y4]);  
function getAreaByShoelaceFormula(arg) {  
    var arr = [...arg, arg[0]];  
    var x_sum = 0;  
    var y_sum = 0;  
    for (let i = 0; i < arr.length - 1; i++) {  
        x_sum += (arr[i].x) * (arr[i + 1].y);  
        y_sum += (arr[i].y) * (arr[i + 1].x);  
    }  
    return Number((Math.abs(x_sum - y_sum) / 2).toFixed(2));  
}
```

Простий багатокутник.



$$P_i = (x_i, y_i), \quad x_i, y_i > 0$$

Функція для перевірки чи дві лінії (a, b, c, d) та (p, q, r, s) перетинаються, яка повертає координати точки перетину, якщо така є.

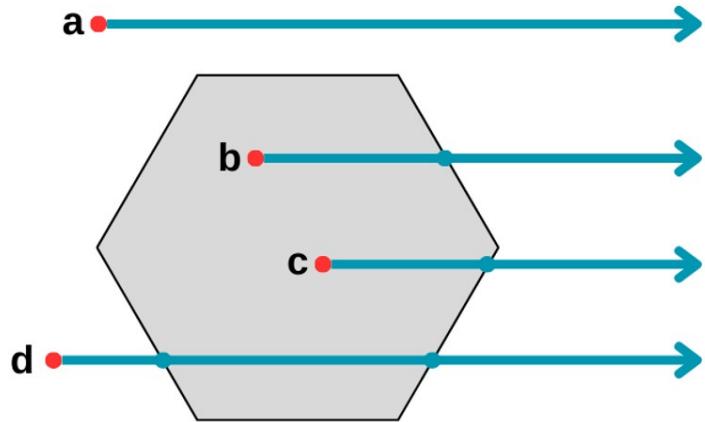
```
function intersectionPoint(a, b, c, d, p, q, r, s) {  
    var det, gamma, lambda;  
    det = (c - a) * (s - q) - (r - p) * (d - b);  
    if (det === 0) {  
        return false;  
    } else {  
        lambda = ((s - q) * (r - a) + (p - r) * (s - b)) / det;  
        gamma = ((b - d) * (r - a) + (c - a) * (s - b)) / det;  
        return 0 < lambda && lambda < 1 && 0 < gamma && gamma < 1;  
    }  
}
```

Функція, яка перевіряє чи дві лінії перетинаються і повертає булеве значення (правда, неправда). Функція базується на векторному добутку.

```
function intersects(a, b, c, d, p, q, r, s) {  
    const det = (c - a) * (s - q) - (r - p) * (d - b);  
    if (det === 0) {  
        return false;  
    } else {  
        const lambda = ((s - q) * (r - a) + (p - r) * (s - b)) / det;  
        const gamma = ((b - d) * (r - a) + (c - a) * (s - b)) / det;  
        return 0 < lambda && lambda < 1 && 0 < gamma && gamma < 1;  
    }  
}
```

Як перевірити, чи лежить дана точка всередині чи поза багатокутником?

1. Проведіть горизонтальну лінію праворуч відожної точки та продовжте її до нескінчності.
2. Підрахуйте, скільки разів лінія перетинається з ребрами багатокутника.
3. Точка знаходитьться всередині багатокутника, якщо кількість перетинів непарна або точка лежить на краю багатокутника. Якщо жодна з умов не виконується, то точка лежить зовні.



Формула для знаходження точки перетину двох ліній: (x_1, y_1, x_2, y_2) та (x_3, y_3, x_4, y_4) .

```
function intersectionPoint(x1, y1, x2, y2, x3, y3, x4, y4) {  
    let px =  
        ((x1 * y2 - y1 * x2) * (x3 - x4) - (x1 - x2) * (x3 * y4 - y3 * x4)) /  
        ((x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4));  
    let py =  
        ((x1 * y2 - y1 * x2) * (y3 - y4) - (y1 - y2) * (x3 * y4 - y3 * x4)) /  
        ((x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4));  
    return [px, py];  
}
```

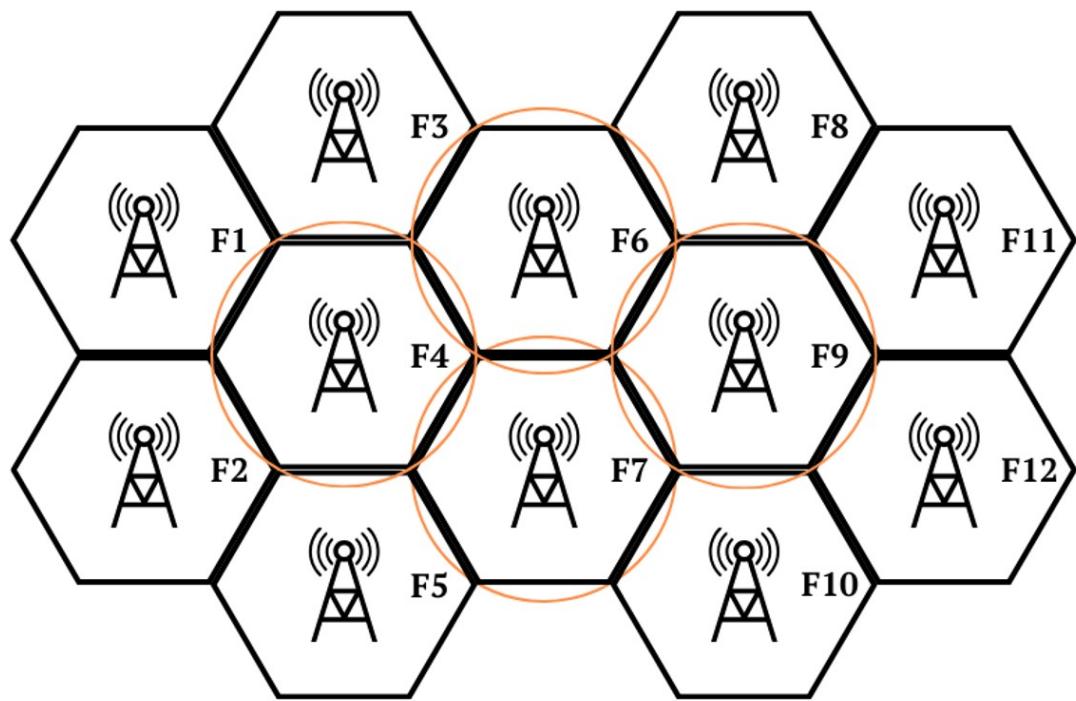
Кутовий коефіцієнт прямої (x_1, y_1, x_2, y_2) можна знайти за формулою $(y_2-y_1)/(x_2 - x_1)$;

Дві прямі паралельні, якщо мають одинаковий кутовий коефіцієнт.

Стільникова мережа

Папп Александрійський пише у своєму творі, що комірка у формі правильного шестикутника має найбільшу площину серед фігур, які можуть без прогалин покрити площину. Саме шестикутні комірки (чарунки) використовують бджоли. Таким чином вони оптимально використовують простір для зберігання меду, бо шестикутники без прогалин покривають площину, при цьому розмір чарунки максимальний. Цю властивість правильного шестикутника використовують для побудови радіомережі. Якщо вежа (вишка) на якій знаходиться антenna випромінює сигнал навколо себе, так, що сигнал покриває площину кола з певним радіусом, а вежа в центрі кола, тоді не вийде покрити такими колами цілком певну територію, бо між колами будуть пробіли. Радіовежі розташовують так, щоб вони утворювали стільниковий зв'язок, тобто кожна вежа ніби знаходиться в центрі уявної шестикутної чарунки, а набір таких радіовеж утворює стільник.

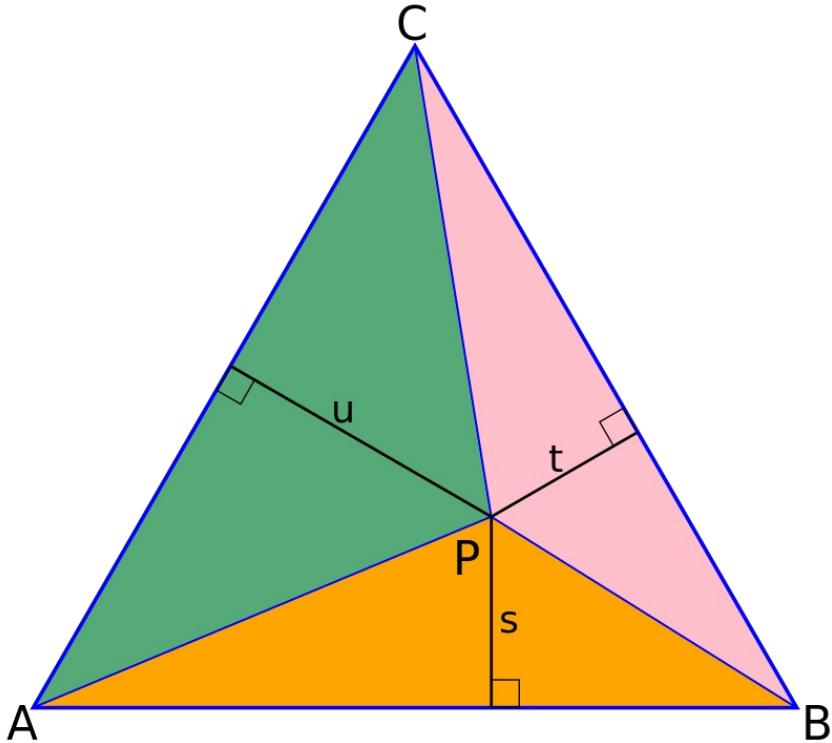
Стільникова мережа.



Стільникові мережі, Точка Ферма, Алгоритм Дейкстри, Алгоритм Пріма, Задача комівояжера, Діаграми Вороного, Цикл Ейлера, Теорема Вівіані мають широке застосування у логістиці. У найширшому сенсі логістикою називають будь-які процеси пов'язані з транспортуванням, зберіганням та обробкою будь-яких предметів.

Теорема Вівіані

Теорема Вівіані — це твердження в геометрії трикутника, згідно з яким сума відстаней від довільної точки всередині прямокутного трикутника до його сторін є постійною і дорівнює висоті трикутника. Названа на честь італійського математика Вінченцо Вівіані, який був учнем Торрічеллі та Галілея.



Теорему можна довести, порівнявши площини трикутників. Нехай $\triangle ABC$ — рівносторонній трикутник, у якому h — висота, а a — довжина кожної зі сторін. Всередині трикутника довільно вибирається точка P , і тоді t, u, s — відстані від точки P до сторін трикутника. Тоді площину $\triangle ABC$ можна визначити так:

$$S \triangle ABC = S \triangle ABP + S \triangle ACP + S \triangle BCP, (S \triangle ABC \text{ означає площину } \triangle ABC).$$

Де співвідношення:

$$ah / 2 = at / 2 + au / 2 + as / 2,$$

$$\text{отже: } h = t + u + s,$$

що треба було довести.

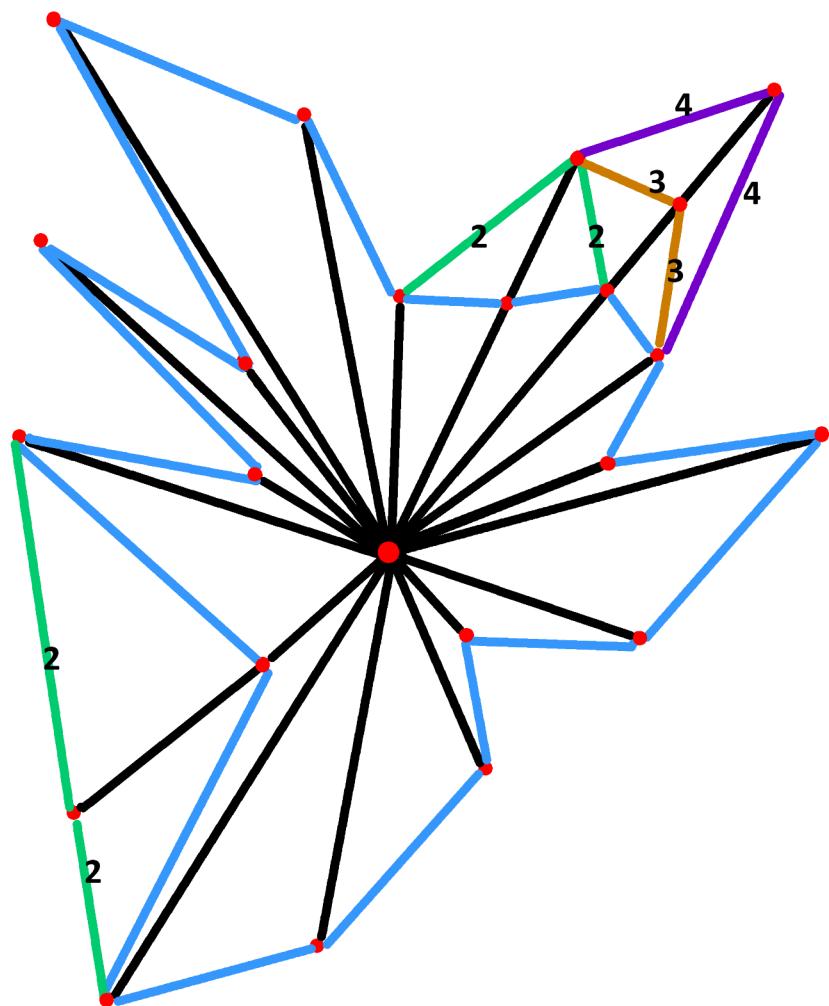
Простий алгоритм тріангуляції

Простий алгоритм тріангуляції. Припустимо, що на площині є безліч точок і між кожною з них потрібно провести пряму, але так, щоб прямі не перетиналися. По суті, потрібно зробити тріангуляцією.

Для тріангуляції можна використовувати такий алгоритм:

1. Виберіть будь-яку точку.
2. Проведіть лінії від усіх інших точок до обраної.
3. З'єднайте всі найближчі до обраної точки на лініях.
4. Повторіть крок 3.

Вибрана точка може бути умовно центром кола. Звідси випливає, що до неї можна провести пряму від інших точок, але дві інші точки можуть бути на одній прямій.

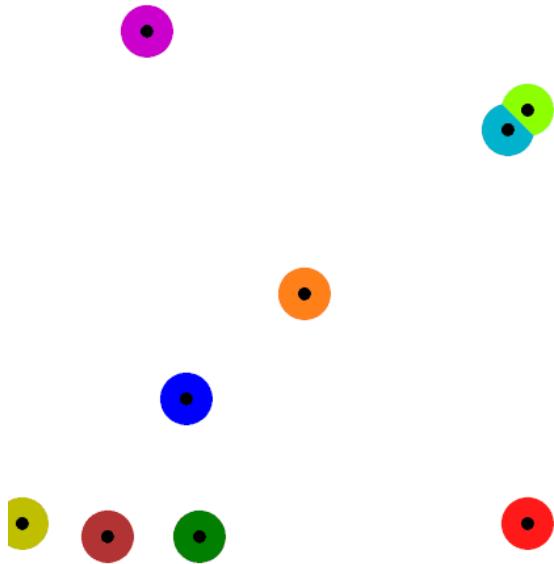


Діаграми Вороного

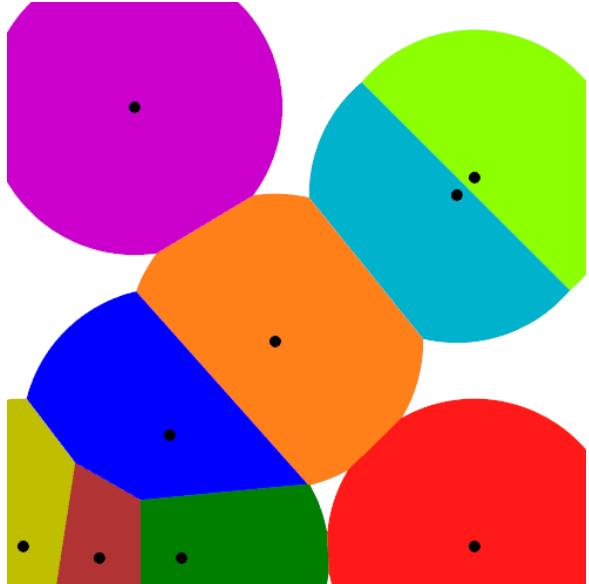
Діаграма Вороного кінцевої множини точок S на площині являє собою розбиття площини таким чином, що кожна область цього розбиття утворює набір точок, які більше до одного з елементів (точок) множини S , ніж до будь-якого іншого елемента множини. Георгій Феодосійович Вороний (1868 – 1908) – відомий український математик, який описав “Діаграму Вороного” в 1908 році.

Побудова діаграми Вороного методом заливки.

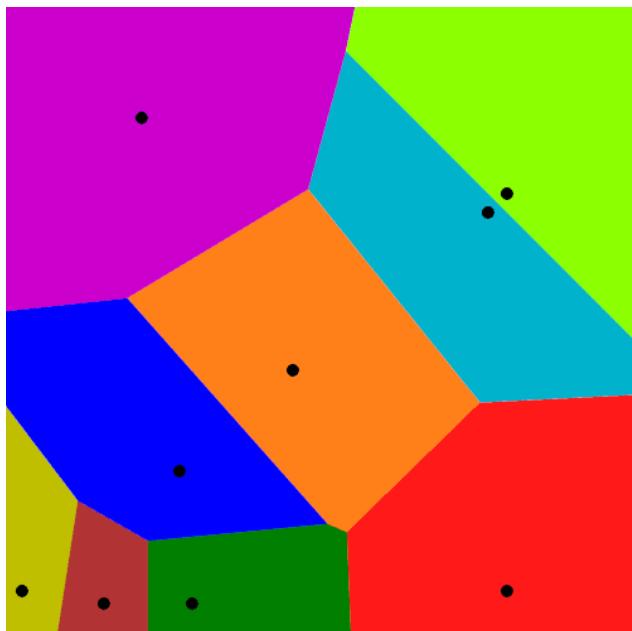
Інтуїтивно спосіб заповнення можна представити так: з кожної точки одночасно починає текти вода, рівномірно заповнюючи площину навколо точки. Коли ж де вода з двох точок зустрічається, вона зупиняється.



Продовжуємо розширення



І ми отримуємо



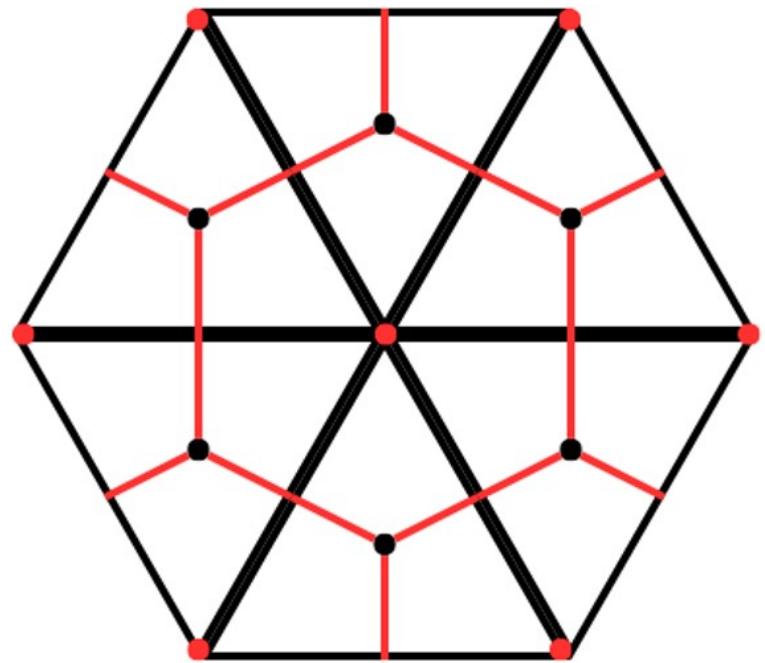
Спосіб побудови діаграми Вороного з допомогою тріангуляції.

Алгоритм.

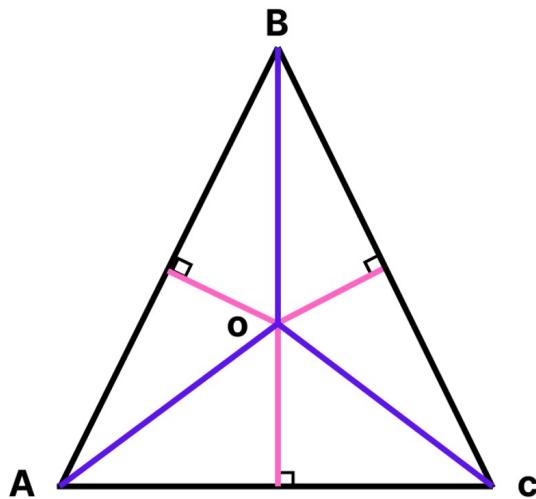
Якщо маємо лише одну або дві точки, то побудова діаграми Вороного тривіальна, це площа або дві півплощини. Якщо є три або більше точки, ми надаємо наступне:

1. З множини точок (зерен) виберіть три точки, утворивши таким чином трикутник, всередині якого не повинно бути інших точок.
2. Визначити центри сторін трикутника (це легко знаючи координати точок і користуючись теоремою Піфагора).
3. Від центрів цих сторін під прямим кутом проведіть три прямі до точки їх перетину. Якщо ця точка лежить за межею трикутника, то проведіть лінії вниз до цієї межі.
4. Зробіть це для всіх трикутників з рештою точок.

Вершини трикутників осередки, а лінії в середині трикутників формують межі діаграмами Вороного.



Теорема. Усі серединні перпендикуляри, проведені до сторін довільного трикутника, перетинаються в одній точці.



Доказ. Розглянемо два перпендикуляри до сторін AC і AB трикутника ABC і позначимо точку їх перетину літерою О.

Оскільки точка О лежить на середині перпендикуляра до відрізка AC, то через те, що кожна точка середини перпендикуляра до відрізка знаходиться на однаковій відстані від кінців цього відрізка, буде вірною рівність:

$$CO = AO.$$

Оскільки точка О лежить у середині, перпендикулярній до відрізка AB, виконується наступна рівність:

$$AO = BO.$$

Отже, справедлива рівність:

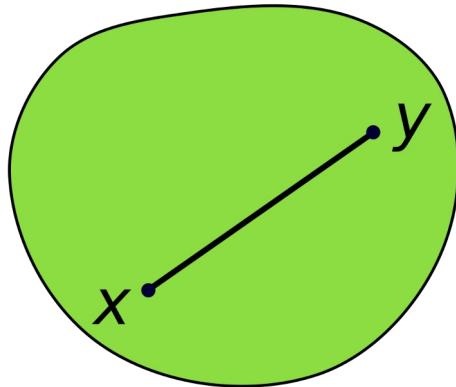
$$CO = BO,$$

звідки через те, що у випадку коли точка знаходиться на однаковій відстані від кінців відрізка, вона також лежить на медіані, перпендикулярній цьому відрізку, робимо висновок, що точка О лежить на медіані, перпендикулярній до відрізка BC. Таким чином, усі три серединні перпендикуляри проходять через одну і ту ж точку, як потрібно.

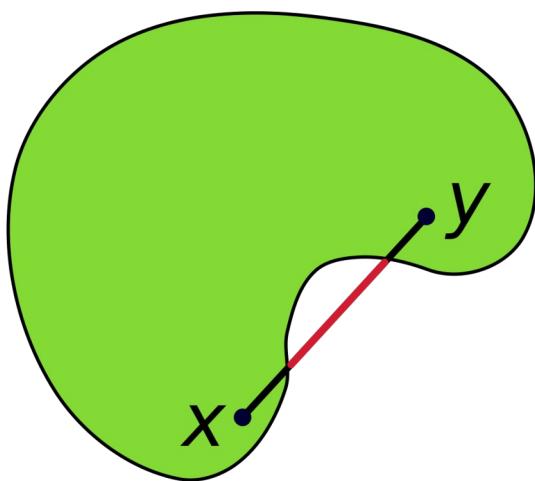
Компактна множина в метричному просторі (або топологічному просторі) — це множина, яка є обмеженою і замкненою. Іншими словами, множина є компактною, якщо вона обмежена (знаходитьться всередині деякого сферичного об'єкта) і вона є замкнutoю (містить всі свої граничні точки). Відрізок $[0, 1]$ є компактним, оскільки він є обмеженою (міститься між 0 і 1) і закритою (включає крайні точки 0 і 1) множиною на числовій прямій. Наприклад, відкритий інтервал $(0, 1)$ не є компактною множиною, бо він не замкнений.

Алгоритм Джарвіса (Jarvis) (або алгоритм упаковки подарунків) визначає послідовність елементів набору, які утворюють опуклу оболонку для цього набору.

Опукла множина в афінному або векторному просторі — це множина, в якій усі точки відрізка, утвореного будь-якими двома точками даної множини, також належать цій множині. Алгоритм названий на честь Р. Джарвіса (Jarvis), який опублікував його в 1973 році.



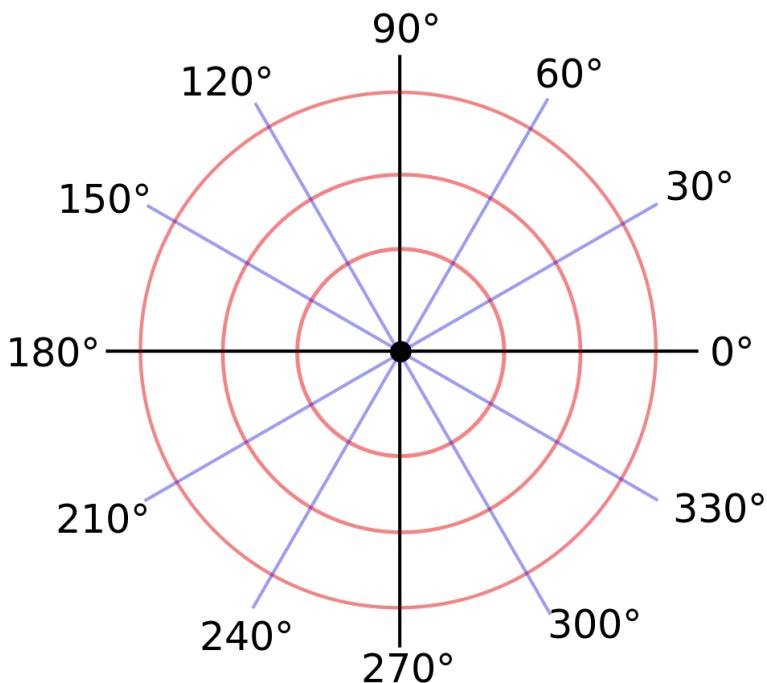
Опукла множина



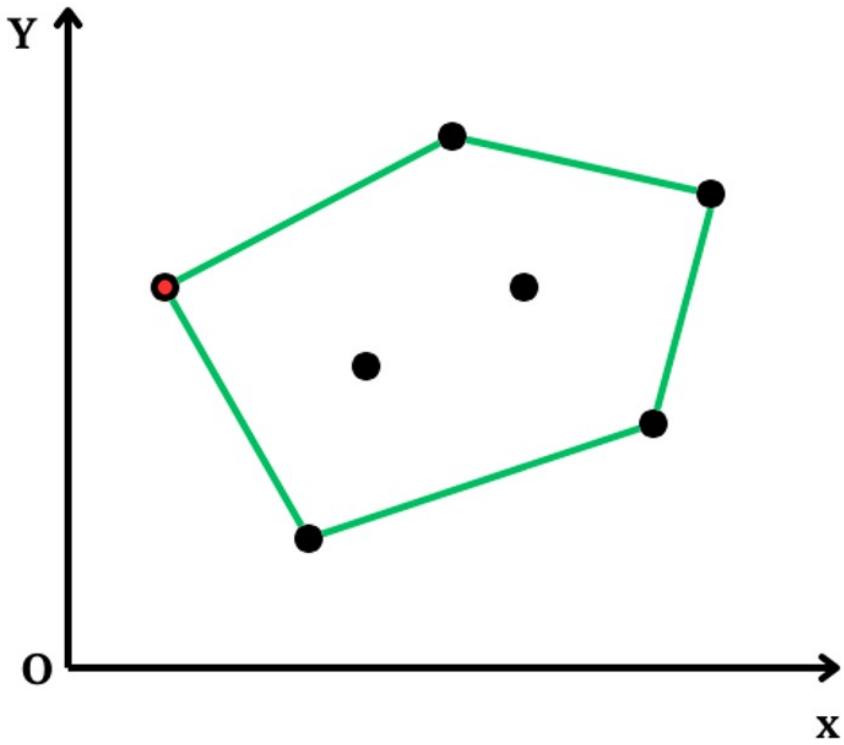
Неопукла множина

Суть алгоритму Джарвіса полягає в наступному. Припустимо, що у нас є набір точок на двовимірній площині, для яких ми хочемо побудувати двовимірну опуклу оболонку. Для цього достатньо зробити:

1. знайти крайню точку за координатою x і визначити її як початкову.
2. починаючи з початкової точки, методом перерахування знаходимо точку з множини, яка, якщо з'єднана з вихідною, утворить відрізок, який матиме найбільший кут на полярній осі координат.
3. продовжуйте виконувати крок 2, починаючи з третьої точки, для всіх точок, поки не буде побудована закрита опукла оболонка.



Набір точок після застосування алгоритму Джарвіса



Приклад використання для нормалізації форм



Алгоритм точкової агрегації на основі словника

Алгоритм точкової агрегації на основі словника (DBDA - Dictionary-based dot aggregation)

Основний принцип

Ініціалізуйте порожній словник, де ключами є мітки кластерів, а значеннями є набори точок, що належать кожному кластеру. Словник поки що не має ключів.

Для кожної точки в наборі даних перевірте, чи вона знаходиться в межах зазначеного радіуса будь-якого існуючого кластера в словнику. Вам не потрібно використовувати цикл для цього, тому що ми використовуємо словник (асоціативний масив) з ключами, які представляють порядок кластера за х і у. Отже, арифметично отримуємо координати кластера точки та перевіряємо, чи існує такий ключ у словнику. Припустимо, кластер точок матиме порядок: floor(px/радіус), floor(py/радіус), тому перевірте, чи існує цей ключ у словнику. Якщо так, додайте нову точку до відповідного набору точок для цього кластера. Якщо ні, створіть нову мітку кластера та додайте точку до словника з новою міткою.

Алгоритм точкової агрегації на основі словника не створить більше кластерів, ніж кількість точок у наборі даних. Кожна точка буде призначена одному кластеру, а нові кластери створюватимуться, лише якщо точка не входить до заданий радіус будь-якого існуючого кластера.

Точкове агрегування на основі словника бере свій початок від кластеризації на основі сітки.

Алгоритм точкової агрегації на основі словника:

Припустимо, що ми маємо масив точок. Візьміть перший пункт і помістіть його в словник (асоціативний масив). Отже, тепер масив містить один елемент. Хеш або ключ для цієї точки слід обчислити за формулою $\text{Math.floor}(x * \text{scale} / \text{radius}) + “_” + \text{Math.floor}(y * \text{scale} / \text{radius})$, яка дасть рядок, який міститиме інформацію про порядок кластера (x, y – координати точки). Ми не змінюємо положення точки, якщо вона одна в кластері.

Потім ми беремо наступну точку з початкового масиву і перевіряємо, чи вже є хеш (ключ) у нашому словнику за тією ж формулою. Якщо такого хешу немає в словнику (який представляє кластер), ми розміщуємо його в словнику як першу точку, інакше ми об'єднуємо його з точкою, яка має такий самий хеш у словнику, і розміщуємо основну точку кластера в центрі кластера. Алгоритм точкової агрегації на основі словника (точкова агрегація на основі словника) не потребує зберігання деякої сітки, яка представляє кластери. Кластери будуть формуватися під час перевірки точок.

Плюси:

-Алгоритм точкової агрегації на основі словника є відносно швидким $O(n)$ – лінійна часова складність. Також, як відомо, час доступу до хеш-таблиці $O(1)$.

-DBDA не перевіряє та не зберігає порожні та попередньо визначені кластери.

Мінуси:

-DBDA залежить від порядку точок.

-Якщо два сусідніх кластери мають лише по одній точці, вони не будуть об'єднані, тому дві точки можуть бути близче одна до одної, ніж радіус.

JavaScript Code of DBDA (Алгоритм точкової агрегації на основі словника)

```
const data = [
  {
    pos_x: 11700,
    pos_y: 9621,
    val: 29.88,
  },
  {
    pos_x: 11700,
    pos_y: 9622,
    val: 1000,
  },
  {
    pos_x: 16553,
    pos_y: -6951,
    val: 29.26,
  },
];
function gen_hash(x, y, radius, scale) {
  return (
    Math.floor((x * scale) / radius) + " " +
    Math.floor((y * scale) / radius)
  );
}
function aggregation(points, radius, scale) {
  const dictionary = new Map();
  points.forEach((point) => {
    const hash = gen_hash(point.pos_x, point.pos_y, radius, scale);
    if (dictionary.has(hash) === true) {
      const parent_point = dictionary.get(hash);
      parent_point.val += point.val;
      parent_point.aggregated = true;
      parent_point.pos_x =
        Math.floor(parent_point.pos_x / radius) * radius + radius / 2;
      parent_point.pos_y =
        Math.floor(parent_point.pos_y / radius) * radius + radius / 2;
    }
  });
}
```

```
} else {
    dictionary.set(hash, point);
}
});

return Object.values(Object.fromEntries(dictionary));
}
```

Ось деякі переваги алгоритму “точкової агрегації на основі словника” (DBDA):

Часова складність: Алгоритм DBDA має часову складність $O(n)$ у гіршому випадку, що є дуже ефективним і дозволяє йому обробляти великі набори даних без значного збільшення часу обробки. Як відомо, час доступу до хеш-таблиці $O(1)$ – константа.

Ефективність пам'яті: завдяки використанню словника для зберігання агрегованих точок алгоритм DBDA ефективно використовує пам'ять і не потребує великого обсягу пам'яті для роботи.

Гнучкість: алгоритм можна налаштувати відповідно до різних потреб, наприклад змінити розмір сітки або радіус, який використовується для групування найближчих точок.

Немає порожніх кластерів: алгоритм DBDA не страждає від проблеми порожніх кластерів, яка може бути проблемою з іншими алгоритмами кластеризації.

Важливе обмеження: Якщо два сусідніх кластери мають лише одну точку, вони не будуть об'єднані, тому дві точки можуть бути близче одна до одної, ніж радіус.

Алгоритм кластеризації даних

Цифрове зображення це масив точок (пікселів).

Знак на цифровій картинці – це певний набір точок, який можна виділити із загальної картинки за кольором, або за просторовим розміщенням.

Таким чином, розклавши картинку на шари за кольором, можна шукати знак на цих шарах або їх комбінаціях. Потрібно просто перевірити чи підходить знак через певний еталон. Для розпізнавання знаків на цифрових зображеннях використовують різні алгоритми та їх комбінації, зокрема, алгоритм кластеризації (DBSCAN), алгоритм Джарвіса для обведення, фільтрацію, розмиття (наприклад Розмивання Гауса), Метод Монте-Карло.

Якщо у вас є набір точок в декартовій системі координат і ви хочете всі скучення точок виділити в конкретну групу (кластер), тоді для цього можна використовувати алгоритм DBSCAN.

DBSCAN (density-based spatial clustering of applications with noise) — алгоритм кластеризації даних, який запропонували Мартін Естер, Ганс-Петер Крігель, Йорг Сандер та Сяовей Су в 1996 році.

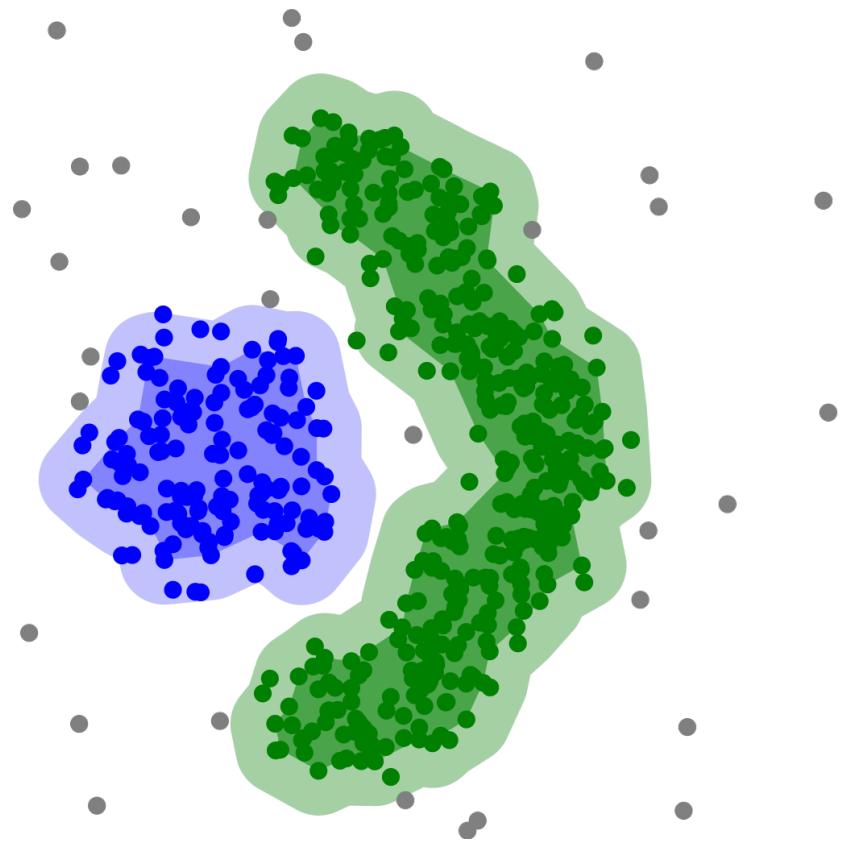
Зверніть увагу DBSCAN це не алгоритм агрегації, а саме алгоритм маркування точок на належність до певної групи. Цей алгоритм корисний для обробки зображень, наприклад, для групування точок на зображенні, які, ймовірно, складають якийсь символ. Також DBSCAN використовують у галузі машинного навчання, коли машина групует дані й присвоює їм певні категорії. Поодинокі точки, які не входять в жодну групу, алгоритм вважає шумом.

DBSCAN досить трудомісткий алгоритм по часової складності. DBSCAN має складність у гіршому випадку $O(n^2)$.

Результат DBSCAN трішки залежить від порядку точок в масиві.

Ось кроки алгоритму DBSCAN:

1. Переберіть всі точки й визначте, які з них ключові точки, відповідно до радіусу r та мінімуму сусідів minN . Точка вважається ключовою, тобто осередком, якщо вона має щонайменше minN сусідів в межах радіусу r , тобто сусідів, які не далі ніж довжина r .
2. Переберіть всі ключові точки (осередки) й розподіліть їх по кластерах. Дві ключові точки (осередки) належать одному кластеру, якщо вони перебувають в межах відстані r один від одного.
3. Переберіть всі інші (не ключові) точки Y , якщо біля них в межах радіуса r є ключова точка, приєднайте їх до кластера ключової точки в межах r . Після приєднання, точка не стає осередком Y , відповідно, не буде впливати на процес перевірки по додаванню наступних точок. Момент: Якщо біля звичайної точки в межах радіуса r є дві ключові точки, тобто осередки, тоді ця точка приєднається до осередку (точніше кластера), який буде перевірятись першим, тобто має менший порядковий номер в масиві.
4. Поодинокі точки, які не увійшли в жоден кластер при minN не дорівнює 0, будуть вважатися шумом, тобто прибрані з розгляду.

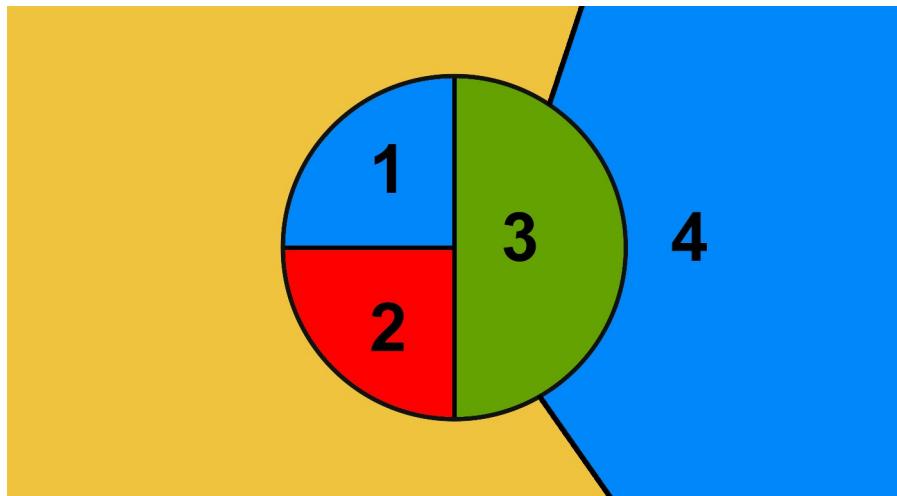


Теорема про чотири кольори

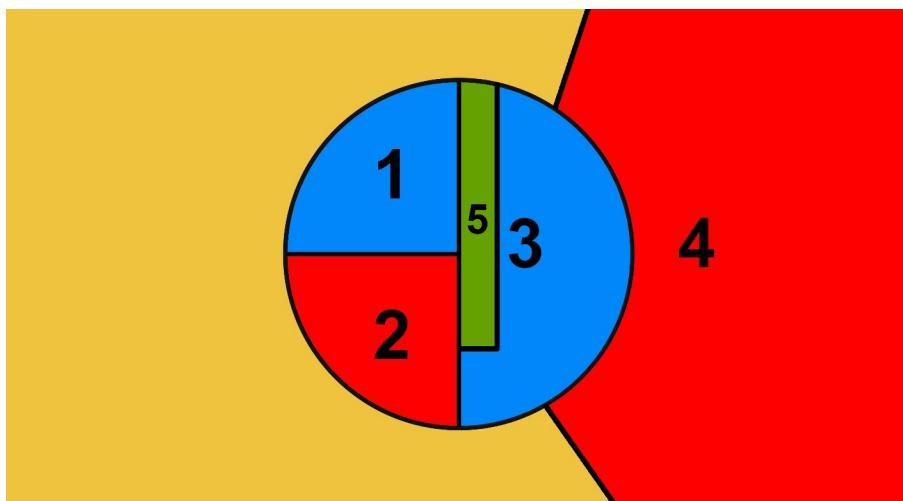
Теорема про чотири кольори — це теорема, яка стверджує, що будь-яку мапу (карту), розташовану на площині або на сфері, можна розфарбувати не більше ніж чотирма різними кольорами (фарбами), так що будь-які дві області зі спільною ділянкою кордону будуть забарвлені в різні кольори. У цьому випадку ділянки повинні бути просто з'єднані (без анклавів, тобто без включень, обривів, іншими словами, щоб будь-який замкнутий контур на мапі можна було стягнути в точку), а загальна межа (кордон) означає частину лінії, тобто стики кількох ділянок в одній точці не вважаються для них спільною межею. Іншими словами, покажіть, що хроматичне число плоского графа не перевищує 4. Хроматичне число графа — це мінімальна кількість кольорів, у яку можна розфарбувати вершини графа G так, щоб кінці будь-якого ребра мали різні кольори. Позначається як $\chi(G)$.

Наскільки відомо, вперше ця гіпотеза була висунута у 1852 році, коли Френсіс Гатрі, намагаючись розфарбувати мапу графств Англії, помітив, що потрібні лише чотири різні кольори. У той час брат Гатрі, Фредерік, був учнем Августа Де Моргана (колишнього радника Френсіса) в Університетському коледжі Лондона. Френсіс поцікавився у Фредеріка про це, який потім відніс його до Де Моргана (Френсіс Гатрі закінчив навчання в 1852 році, а пізніше став професором математики в Південній Африці). У 1890 році Персі Хівуд довів теорему про п'ять кольорів і узагальнив гіпотезу про чотири кольори на поверхні довільного роду. Зокрема, про це повідомляє Ріхард Курант в книзі “Що таке математика?” (1941). Теорема про чотири кольори була доведена в 1976 році Кеннетом Аппелем і Вольфгангом Хакеном після багатьох хибних доказів і контрприкладів (на відміну від теореми про п'ять кольорів, доведеної в 1800-х роках, яка стверджує, що п'ятирівки достатньо для фарбування карти). Щоб розвіяти будь-які сумніви щодо доказу Аппеля –Хакена, у 1997 році Робертсон, Сандерс, Сеймур і Томас опублікували простіший доказ, який використовує ті самі ідеї та все ще спирається на комп’ютери. Крім того, у 2005 році теорема була доведена Жоржем Гонтьє з допомогою програмного забезпечення для доведення теорем загального призначення. Це була перша велика теорема, доведена з допомогою комп’ютера.

Теорема. Щоб розфарбувати будь-яку мапу (тобто систему кордонів), розташовану на площині або на кулі, потрібно щонайменше 4 кольори, щоб будь-які дві області зі спільним кордоном (спільною лінією границі) були пофарбовані в різні кольори. (Залишилося довести, що потрібно не більше 4). Доведення (методом надання прикладу чи контрприкладу до зворотної теореми). Цю картку не можна розфарбувати менш ніж чотирма кольорами.



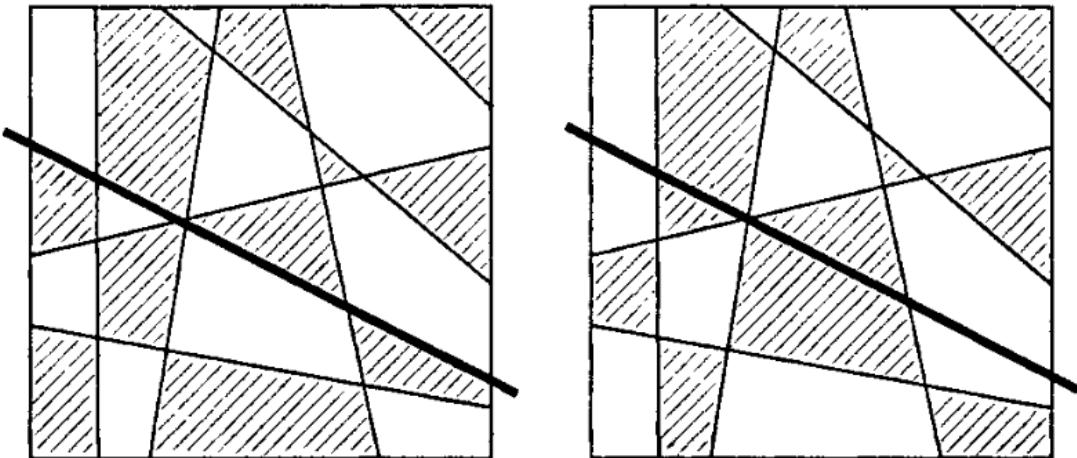
Ця карта потребує щонайменше чотири кольори, щоб розфарбувати її без поєднання областей з однаковими кольорами.



Теорема про чотири кольори

Теорема про чотири кольори (фарби). Будь-яку карту без анклавів можна розфарбувати в 4 кольори, щоб жодні сусідні області не мали однакового кольору. Анклав — територія або частина однієї держави, оточена з усіх сторін територією іншої держави.

Американський математик Мартін Гарднер (1914 - 2010) у своїй книзі “Математичні загадки” наводить такий факт (теорему) про розфарбовування площини, поділеної прямими.

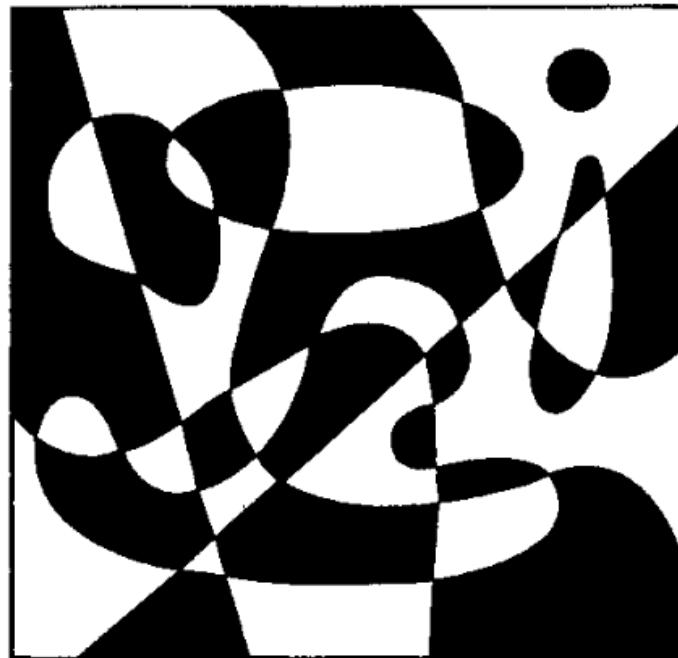


Теорема. Для розфарбовування будь-якої карти (мапи), утвореної прямими лініями, що перетинають всю її поверхню від одного краю аркуша до іншого, достатньо двох кольорів.

Доказ.

На одній правильно розфарбованій карті необхідного нам типу намалуйте (жирну) пряму, що розділяє площину на дві карти. Кожна з нових карт окремо забарвлена правильно, але нова межа (жирна лінія) щойно намалюваної прямої лінії примикає пари областей, пофарбованих у той самий колір. Щоб відновити правильное забарвлення всієї карти в цілому, вам потрібно перефарбувати одну з напівкарт, змінивши колір кожної з областей на протилежний, тобто інвертувати одну половину вихідної карти (що знаходиться з одного боку жирної лінії). Якщо дана площа, розділена на дві області однією прямою, то таку карту, звичайно, можна розфарбувати у два кольори. Намалюємо другий рядок і розфарбуємо нову карту, змінивши всі кольори з одного боку нової лінії на протилежні. Потім він малиє третю пряму лінію, і поки ми завжди можемо повторити нашу процедуру інверсії та отримати карту правильноого кольору. Отже, методом “повної математичної індукції” ми довели теорему про можливість розфарбовування двома кольорами всіх карт, утворених лініями на площині.

Теорему можна узагальнити на випадок більш різноманітних карт, наприклад, для карт, утворених або кривими, які перетинають весь аркуш від одного краю до іншого, або замкнутими кривими без самоперетину. Якщо щойно намальована крива замкнена, вам потрібно змінити колір усіх областей, які знаходяться всередині кривої, або, якщо хочете, всіх областей, що знаходяться за її межами.



Двох кольорів достатньо, щоб розфарбувати карту, утворену лініями, що йдуть від одного краю аркуша до іншого, або замкнутими кривими.

Алгоритм Боуера – Ватсона

Алгоритм Боуера – Ватсона – інкрементний алгоритм тріангуляції Делоне для скінченного набору точок у будь-якому вимірі. Алгоритм Боуера-Ватсона працює шляхом додавання точок до наявної тріангуляції Делоне по одній. Після кожної вставки всі трикутники, описані кола яких містять точку, видаляються, залишаючи структуру, яка повторно тріангулюється з використанням наступної точки.

Тріангуляція Делоне для множини точок P на площині — це така тріангуляція $DT(P)$, що жодна точка множини P не знаходиться всередині описаних довкола трикутників кіл в множині $DT(P)$.

Тріангуляція Делоне дозволяє якомога зменшити кількість малих кутів. Цей спосіб тріангуляції був винайдений Борисом Делоне в 1934 році.

Нижче описаний, мовою JavaScript, алгоритм Боуера-Ватсона для триангуляції Делоне:

Алгоритм починається з визначення супер-трикутника (`createSupertriangle`), який охоплює всі вхідні точки.

Алгоритм ітерується через кожну вхідну точку по одній.

Дляожної точки визначаються "погані трикутники" - трикутники, які вже не є трикутниками Делоне (це означає, що точка знаходиться всередині їхніх описаних кіл).

Створюються нові трикутники, з'єднуючи поточну точку з вершинами "поганих" трикутників.

Допоміжна Функція - `isPointInsideCircumcircle`:

Ця функція перевіряє, чи знаходиться точка всередині описаного кола заданого трикутника. Вона розраховує детермінант та порівнює їх, щоб визначити, чи знаходиться точка всередині описаного кола.

Після обробки всіх вхідних точок алгоритм видаляє трикутники, які містять будь-яку з вершин супер-трикутника, оскільки вони були введені лише для ініціалізації.

Результатом є масив трикутників, що представляють триангуляцію Делоне вхідних точок.

```
function delaunayTriangulation(points, superTriangle) {
    // Add super-triangle to contain all points
    let triangles = [superTriangle];

    for (const point of points) {
        const badTriangles = [];
        const newTriangles = [];

        // Find triangles that are no longer Delaunay
        for (const triangle of triangles) {
            if (isPointInsideCircumcircle(point, triangle)) {
                badTriangles.push(triangle);
            } else {
                newTriangles.push(triangle);
            }
        }

        // Create new triangles by connecting the point to the vertices of bad triangles
        for (const triangle of badTriangles) {
            const vertexA = triangle[0];
            const vertexB = triangle[1];
            const vertexC = triangle[2];
            const newTriangle = [point, vertexA, vertexB];
            newTriangles.push(newTriangle);
            newTriangle = [point, vertexB, vertexC];
            newTriangles.push(newTriangle);
            newTriangle = [point, vertexC, vertexA];
            newTriangles.push(newTriangle);
        }
    }

    return newTriangles;
}
```

```

const edges = [];
for (const badTriangle of badTriangles) {
  for (let i = 0; i < 3; i++) {
    const edge = [badTriangle[i], badTriangle[(i + 1) % 3]];
    const reversedEdge = [edge[1], edge[0]];

    // Avoid adding duplicate edges
    if (
      !edges.some(
        (e) =>
          (e[0] === edge[0] && e[1] === edge[1]) ||
          (e[0] === reversedEdge[0] && e[1] === reversedEdge[1])
      )
    ) {
      edges.push(edge);
    }
  }
}

// Add new triangles formed by the point and edges
for (const edge of edges) {
  newTriangles.push([point, edge[0], edge[1]]);
}

triangles = newTriangles;
}

// Remove triangles containing super-triangle vertices
// return triangles.filter(
//   (triangle) =>
//     !triangle.some(
//       (vertex) =>
//         vertex[0] === superTriangle[0][0] ||
//         vertex[0] === superTriangle[1][0] ||
//         vertex[1] === superTriangle[0][1] ||
//         vertex[1] === superTriangle[1][1]
//     )
// );
//);

return triangles;
}

// Helper function to check if a point is inside the circumcircle of a triangle
function isPointInsideCircumcircle(point, triangle) {
  const [a, b, c] = triangle;
  const detT = (a[0] - c[0]) * (b[1] - c[1]) - (b[0] - c[0]) * (a[1] - c[1]);
  const detA =
    (point[0] - c[0]) * (b[1] - c[1]) - (b[0] - c[0]) * (point[1] - c[1]);
  const detB =
    (a[0] - c[0]) * (point[1] - c[1]) - (point[0] - c[0]) * (a[1] - c[1]);
  const alpha = detA / detT;
  const beta = detB / detT;
  const gamma = 1 - alpha - beta;
}

```

```

return alpha > 0 && beta > 0 && gamma > 0;
}

const points = [
  [100, 100],
  [100, 200],
  [200, 200],
  [200, 100],
];

```

function createSupertriangle(points) {

```

const minX = Math.min(...points.map((point) => point[0]));
const minY = Math.min(...points.map((point) => point[1]));
const maxX = Math.max(...points.map((point) => point[0]));
const maxY = Math.max(...points.map((point) => point[1]));

const dx = maxX - minX;
const dy = maxY - minY;
const deltaMax = Math.max(dx, dy);
const midX = (minX + maxX) / 2;
const midY = (minY + maxY) / 2;

const supertriangle = [
  [midX - 2 * deltaMax, midY - deltaMax],
  [midX, midY + 2 * deltaMax],
  [midX + 2 * deltaMax, midY - deltaMax],
];

```

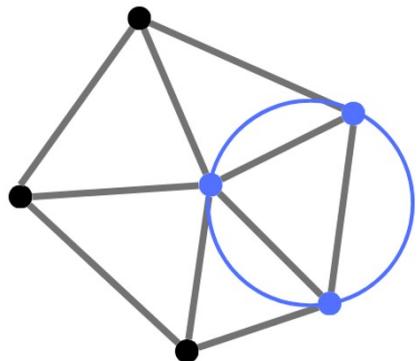
return supertriangle;

}

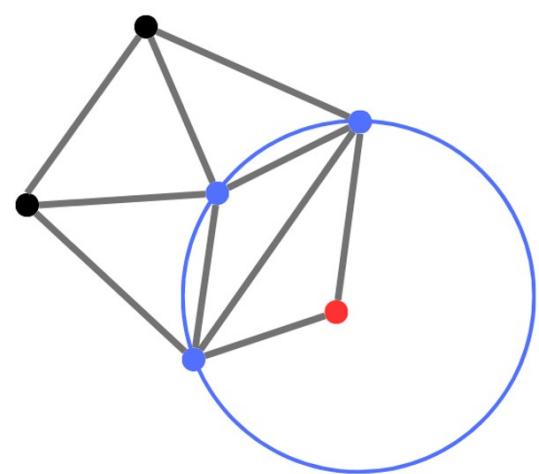
const supertriangle = createSupertriangle(points);

const result = delaunayTriangulation(points, superTriangle);

Delaunay

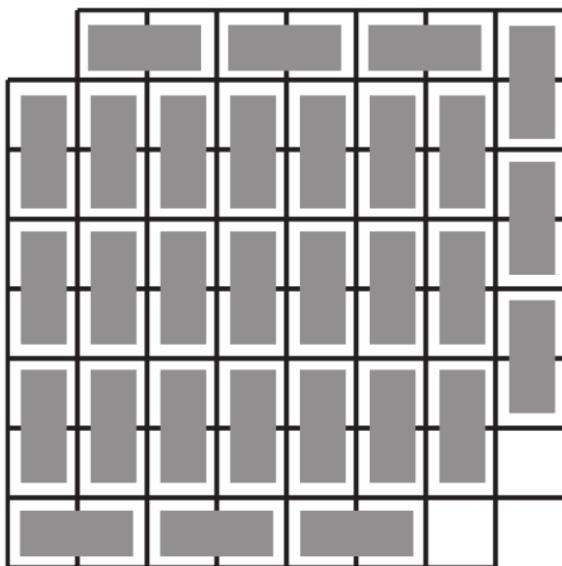


Not Delaunay



Американський популяризатор науки Мартін Гарднер (1914 — 2010) в одній зі своїх книг описує цікаву задачу, яка демонструє красу математики, має практичне значення та демонструє доведення від супротивного. Суть задачі така: Ми маємо необмежену кількість килимів сталого розміру, а саме, кожен килим розміром 1 умовну одиницю в ширину та 2 умовні одиниці в довжину (тобто прямокутний килим). В нас є кімната 8 на 8 умовних одиниць, тобто її площа 64 квадрати. Але ми знаємо що у верхньому лівому квадраті стоять меблі під які не можна ставити килим і у правому нижньому квадраті стоять меблі під які не можна ставити килим. Питання чи можна покрити (замостили) всю нашу кімнату килимами (тобто 62 квадрати) не покриваючи квадрати з меблями й не ріжучи килими.

Схема площи, яку потрібно покрити килимами, з невдалим варіантом покриття.

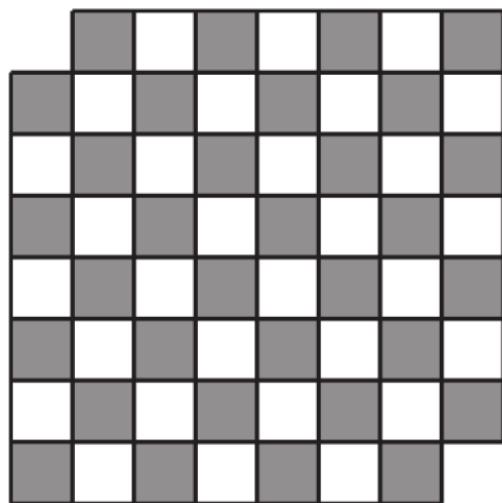


а) Якщо наша квадратна кімната має непарну кількість квадратних метрів, тоді ми не зможемо її покрити прямокутними килимами (1×2).

б) Якщо наша кімната має парну кількість квадратних метрів, це ще не означає, що її можна покрити прямокутними килимами.

Щоб дізнатися чи покриття можливе, потрібно зробите наступне: Умовно розфарбуйте кожен квадратний метр білим та чорним кольором в шаховому порядку. Покриття килимами буде можливе тільки у випадку, якщо кількість квадратних метрів парна та кількість чорних і білих клітинок парна. Кожен килим повинен покрити одну чорну й одну білу клітинки, тобто два квадратних метри.

Правильна відповідь на поставлене питання буде негативною, тобто ні. Не можна замостили описану площину відповідними килимами (1×2), якщо їх не різати. Щоб довести коректність нашої відповіді ми скористаємося доведенням від супротивного. Уявно розфарбуємо квадрати нашої площини в шаховому порядку, тобто методом чергування білого й чорного квадратів. Таким чином ми побачимо, що килим, розміщений в описаній кімнаті, обов'язково покриє один білий квадрат і один чорний квадрат і тільки так. Але в нас чорних квадратів більше ніж білих, бо два білих зайняті меблями, тому ми не можемо покрити килимами нашу площину не різавши їх, бо інакше ми б мали рівну кількість чорних та білих квадратів.



Подібну логіку можна застосовувати для доказу можливості покриття площини у різних сферах, наприклад, коли потрібно покрити підлогу керамічною плиткою.

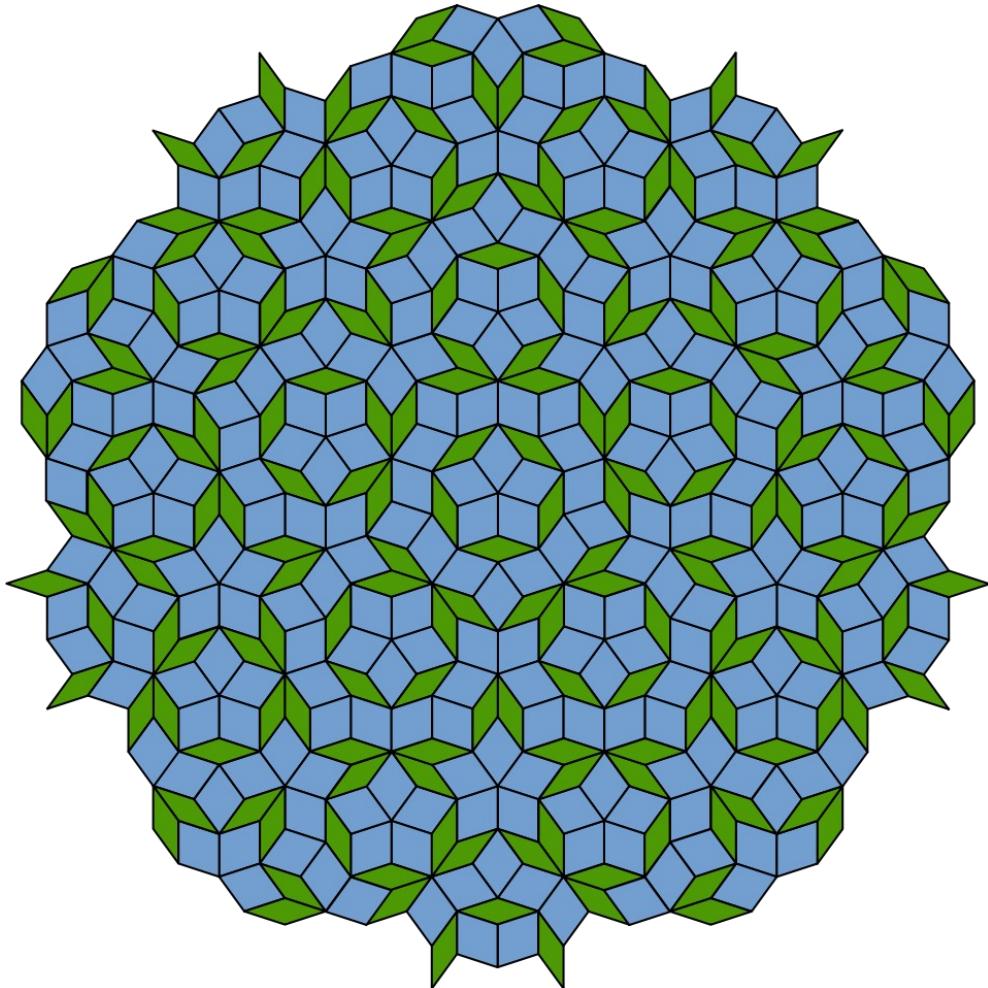
Мозаїка Пенроуза

Мозаїка Пенроуза — загальна назва трьох особливих типів неперіодичного розбиття площини, які названі на ім'я англійського математика Роджера Пенроуза, який досліджував їх у 1970-і роки.

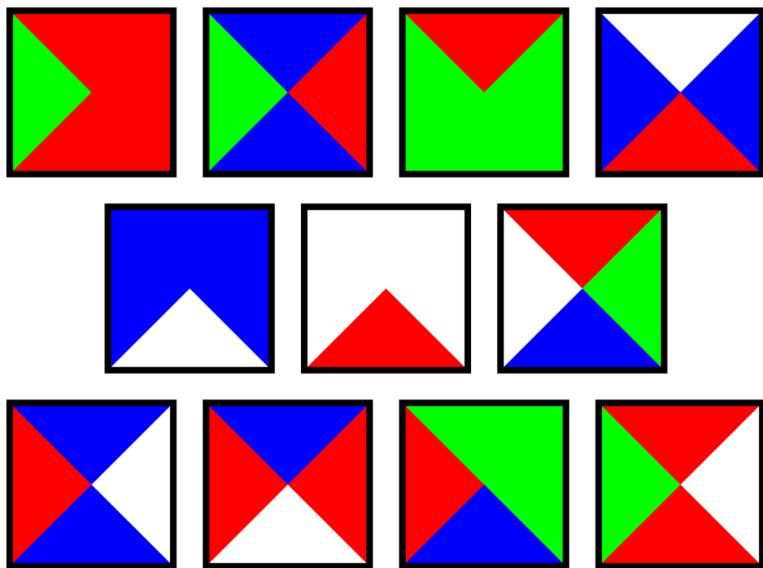
Всі три типи, як і будь-які аперіодичні мозаїки, мають такі властивості:

- неперіодичність — відсутність трансляційної симетрії,
- повторюваність (також звана самоподібністю, що, однак, не пов'язано з одноіменною властивістю фракталів) - будь-який скільки завгодно великий фрагмент мозаїки Пенроуза зустрічається в мозаїці нескінченну кількість разів, хоч і через нерівні відстані,
- квазікристалічність — при дифракції на мозаїці, як на фізичній структурі, дифракційна картина показує наявність далекого порядку та симетрії п'ятого порядку.

Осьова симетрія n -го порядку — симетричність щодо поворотів на кут $360^\circ / n$ навколо будь-якої осі.

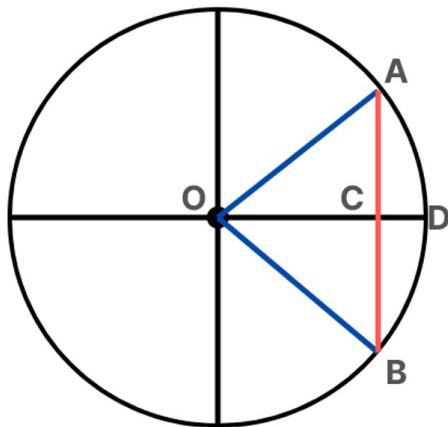


Подумайте про наявність набору квадратних плиток, які можна використовувати для покриття площини, але для з'єднання плиток між собою потрібно використовувати специфічні взаємодії, аналогічні пазлам (тобто дві плитки з набору можна об'єднати тільки при наявності в них відповідних сторін). Питання таке: чи існує алгоритм, який може визначити, чи можливо використовувати даний набір плиток для покриття площини? Математик Роберт Бергер в 1966 році довів, що не існує загального алгоритму для розв'язання цього завдання, продемонструвавши аперіодичні плитки. Роберт Бергер, народився в 1938 році, є відомим прикладним математиком і здобув славу завдяки відкриттю першої аперіодичної плитки.



Цей набір із 11 плиток Ванга покриватиме площину плиткою, але лише аперіодично. Плитки можна з'єднувати тільки сторонами з однаковим кольором. Для цього набору з 11 плиток не існує алгоритму, який відповів би чи вони можуть покрити прямокутну площину.

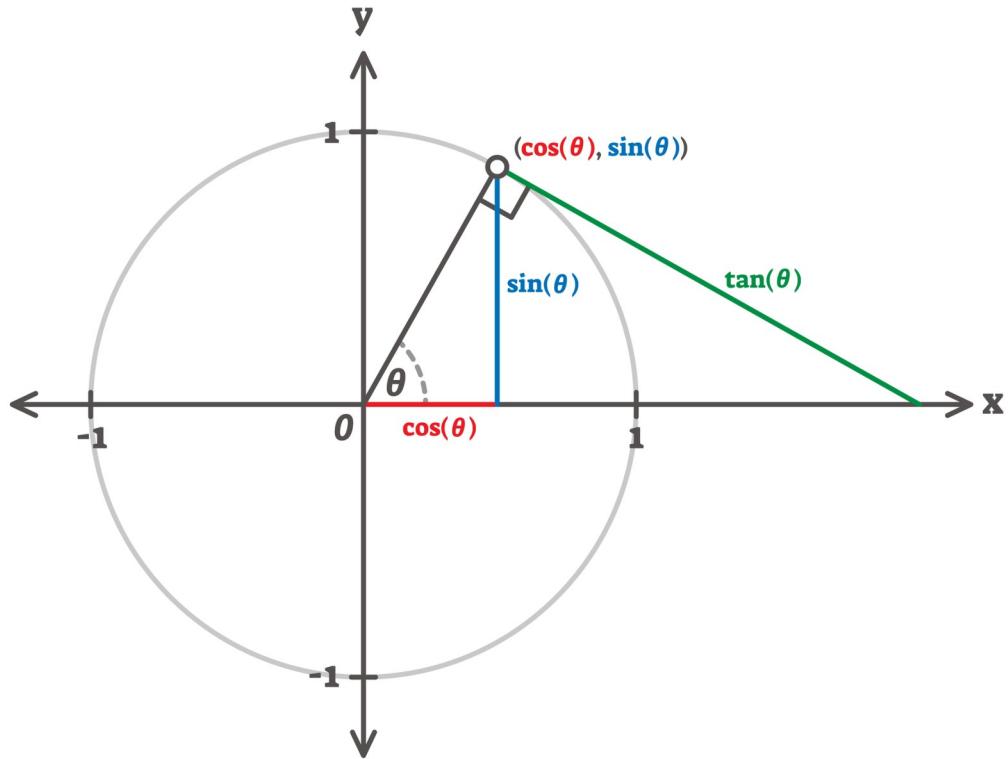
Вже астрономічний трактат Аль-Хорезмі (бл. 780 - бл. 850) містив таблиці синусів, які він запозичив у індійських астрономів. У грецьких математиків хорди (AB) кутів грали роль синусів. Характерною особливістю індійської тригонометрії є те, що вони використовували у своїх обчисленнях не повну хорду даної дуги, а синус цієї дуги, тобто півхорду (AC або CB).



$$\sin(\angle AOC) = AC/OA;$$

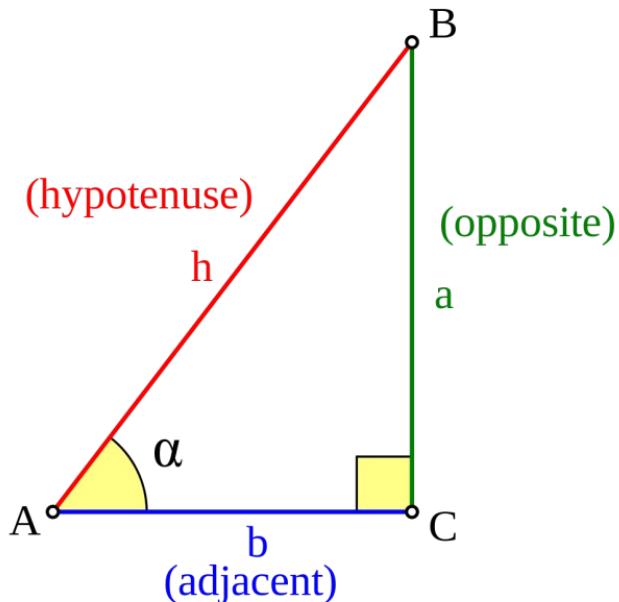
$$\cos(\angle AOC) = OC/OA;$$

Арабський математик 9 століття Аль-Баттані у зв'язку з астрономічними обчисленнями запровадив у вжиток тригонометричні функції (синус, тангенс і котангенс). Тангенсом кута називається відношення довжини протилежного катета до довжини прилеглого катета.



Всі тригонометричні функції можна задати через функцію синус, а саму функцію синус можна наблизено задати через ряди Тейлора.

У математиці Ряд Тейлора — представлення функції у вигляді нескінченної суми доданків, які обчислюються зі значень функцій похідних в одній точці. В 1715 році англійський математик Брук Тейлор запропонував загальний метод побудови цих рядів для усіх функцій, для яких вони існують. Ряд Тейлора для квадратного кореня: $\sqrt{x} = 1 + (1/2*x) - (1/8*x^2) + (1/16 * x^3) - (5/128*x^4) \dots$. Ряд Тейлора для функції синус: $\sin(x) = x - (x^3/3!) + (x^5/5!) - (x^7/7!) + (x^9/9!) \dots$, де $3! = 1*2*3 = 6$.



Синус: $\sin(a) = BC/BA$.

Косинус: $\cos(a) = \sin(a + (\pi/2)) = AC/BA$.

Тангенс: $\tan(a) = \sin(a) / \cos(a) = BC/AC$.

Котангенс: $\cot(a) = \cos(a) / \sin(a)$.

Секанс: $\sec(a) = 1 / \cos(a)$.

Косеканс: $\csc(a) = 1 / \sin(a)$.

Обернені тригонометричні функції:

$\arcsin(\sin(x)) = x$,

$\arccos(\cos(x)) = x$,

$\arctan(\tan(x)) = x$,

$\text{arccot}(\cot(x)) = x$.

Ось приклад функції синуса на мові JavaScript, яка використовує ряд Тейлора:

```
// Функція для обчислення факторіалу
function factorial(n) {
    if (n === 0 || n === 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

// Функція для обчислення синуса за допомогою ряду Тейлора
function calculateSinusUsingTaylor(angleInDegrees, terms) {
    // Перетворення кута з градусів в радіани
    const angleInRadians = angleInDegrees * (Math.PI / 180);

    // Обчислення синуса за рядом Тейлора
    var sinusValue = 0;
    for (var i = 0; i < terms; i++) {
        sinusValue += Math.pow(-1, i) * Math.pow(angleInRadians, 2 * i + 1) / factorial(2 * i + 1);
    }

    return sinusValue;
}

// Приклад використання функції
const angle = 30;
const numberOfTerms = 10; // Кількість термінів для ряду Тейлора
const sinusResult = calculateSinusUsingTaylor(angle, numberOfTerms);
console.log("Синус кута " + angle + " градусів за допомогою ряду Тейлора: " + sinusResult);
```

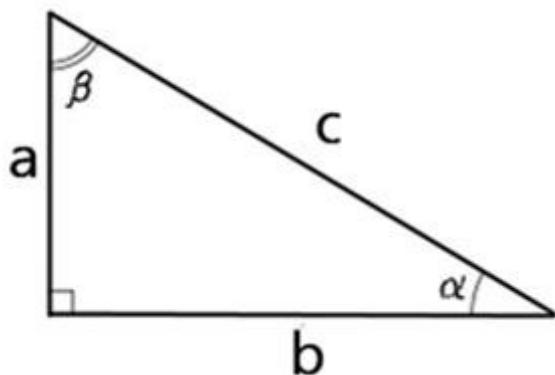
Якщо довжини сторін a , b , c трикутника відомі, а кути невідомі, то для знаходження значень кутів можна скористатися такими формулами:

$$\angle a = \arct(a/b);$$

$$\angle b = \arct(b/a);$$

a , b , c - сторони трикутника.

$$\angle a + \angle b + \angle c = 180;$$



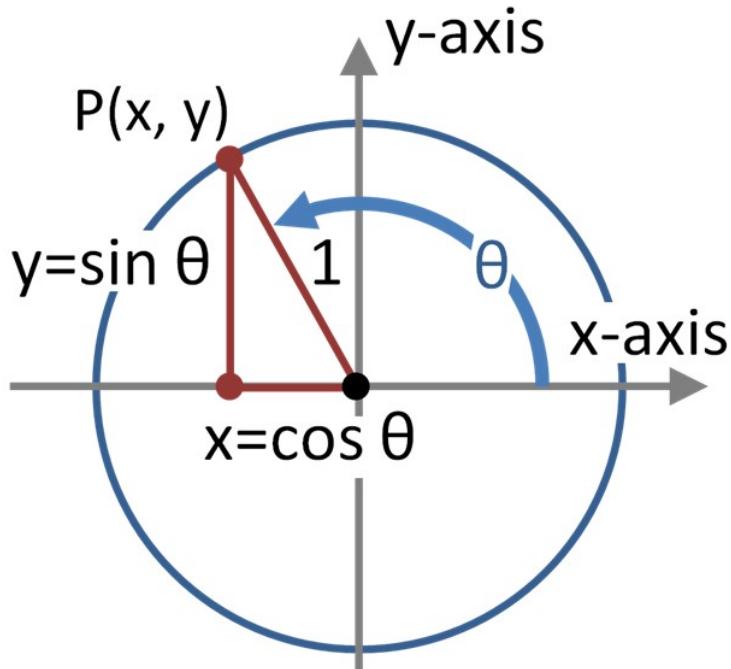
Якщо є дві точки (x_1, y_1) та (x_2, y_2) в декартовій системі координат, тоді кут між лінією, яку вони формують і абсцисою (віссю x) системи координат можна визначити за формулою:

$$\arct((y_2 - y_1) / (x_2 - x_1)) \text{ в радіанах і } \arct((y_2 - y_1) / (x_2 - x_1)) * (100/\pi) \text{ в градусах.}$$

Піфагорійська тригонометрична тотожність

Піфагорійська тригонометрична тотожність:
 $\sin(\theta)^2 + \cos(\theta)^2 = 1$.

Доказ.

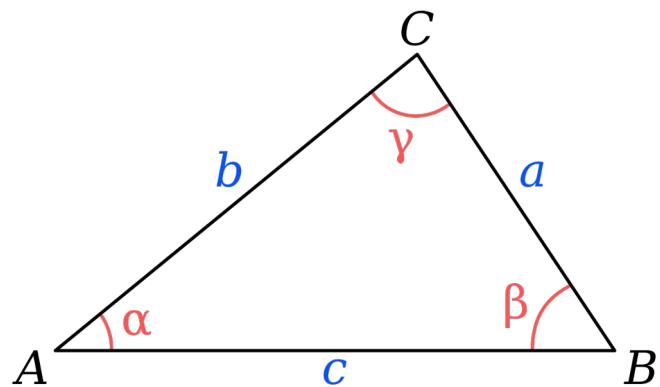


Це твердження очевидне, якщо використовувати коло з радіусом одиниця. В такому колі півхорда буде синусом і водночас катетом прямокутного трикутника з довжиною гіпотенузи 1, та другим катетом, який дорівнює косинусу кута. З теореми Піфагора слідує, що квадрат гіпотенузи дорівнює сумі квадратів катетів. Таким чином якщо ми маємо $\sin(\theta)$, тоді $\sqrt{1 - \sin(\theta)^2} = \cos(\theta)$.

В 1595 році німецький математик Бартоломеус Пітікус видав книгу "Тригонометрія, або трактат про рішення трикутників". Пітікус зробив внесок у розвиток тригонометрії, в тому числі запропонував сам термін "тригонометрія" в ролі назви цієї науки .

В 13 столітті Насір Тусі заснував астрономічну обсерваторію в місті Мераге, що в Ірані. Насір Тусі розробив "пару Тусі" і довів теорему синусів. Теорему косинусів довів Аль-Каші, який працював в обсерваторії Улугбека в Самаркандрі, місто в Узбекистані.

Теорема синусів

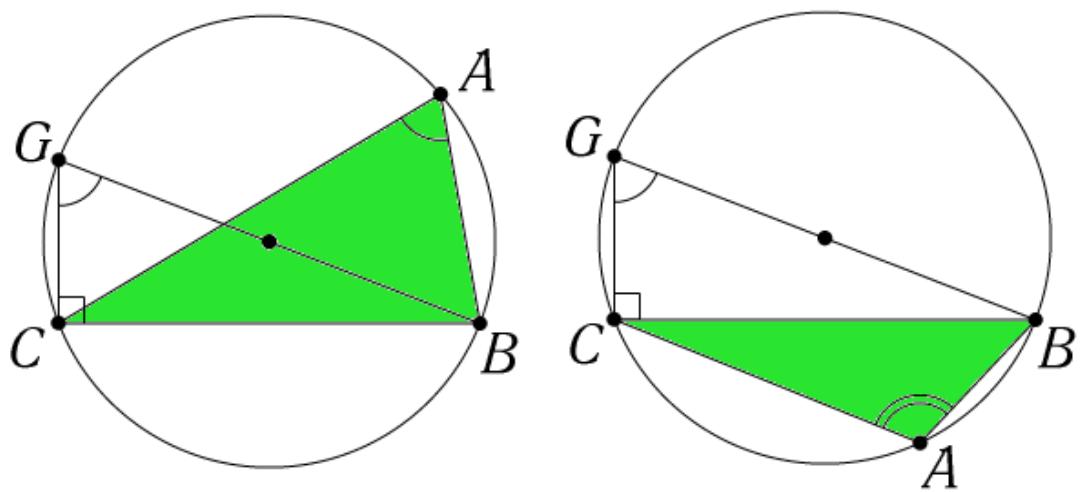


Теорема синусів для довільного трикутника стверджує рівність:

$$(a / \sin(\alpha)) = (b / \sin(\beta)) = (c / \sin(\gamma)) = 2R,$$

де a, b, c — сторони трикутника, α, β, γ — протилежні до них кути, а R — радіус кола, описаного навколо трикутника.

Доказ.



Накресліть діаметр $|BG|$ для описаного кола. За властивістю кутів, вписаних в коло, кут GCB прямий (це слідує з теореми Евкліда про вписані кути), а кут CGB або α , якщо точки A і G лежать по одній стороні від прямої BC , або $180 - \alpha$ в іншому випадку. Оскільки $\sin(180 - \alpha) = \sin(\alpha)$, то в обох випадках отримуємо:

$$a = 180 * \sin(\alpha), \text{ тоді } a / \sin(\alpha) = 180.$$

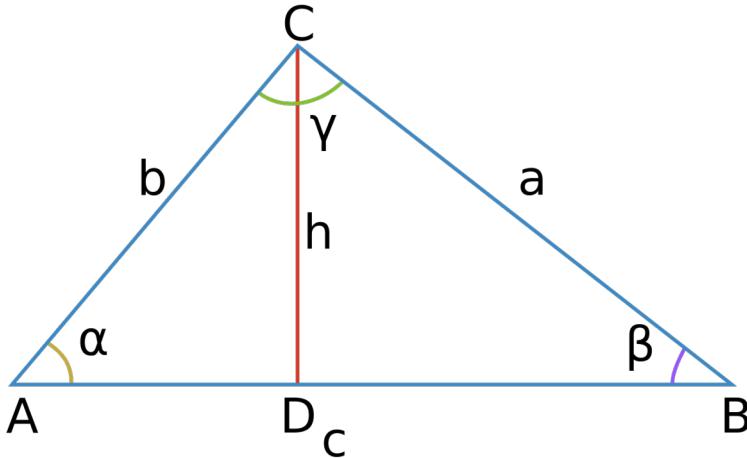
Повторюючи те саме міркування для двох інших сторін трикутника, отримуємо:

$$(a / \sin(\alpha)) = (b / \sin(\beta)) = (c / \sin(\gamma)) = 180.$$

Теорема косинусів

Для трикутника зі сторонами a , b , c і кутом α , протилежним стороні a , справедливе співвідношення:

$$a^2 = b^2 + c^2 - 2 * b * c * \cos(\alpha).$$



Доказ Розглянемо трикутник ABC . Висота CD опускається з вершини C на сторону AB . З трикутника ADC випливає:

$$AD = b * \cos(\alpha),$$

$$DB = c - b * \cos(\alpha)$$

Напишіть теорему Піфагора для двох прямокутних трикутників ADC і BDC :

$$1. h^2 = b^2 - (b * \cos(\alpha))^2$$

$$2. h^2 = a^2 - (c - b * \cos(\alpha))^2$$

Прирівняйте праву частину рівнянь (1) і (2):

$$b^2 - (b * \cos(\alpha))^2 = a^2 - (c - b * \cos(\alpha))^2$$

або

$$a^2 = b^2 + c^2 - 2b * c * \cos(\alpha).$$

Теорему косинусів можна використовувати для знаходження косинусів кутів трикутника:

$$\cos(\alpha) = (b^2 + c^2 - a^2)/2bc.$$

Зокрема,

Якщо $b^2 + c^2 - a^2 > 0$, кут α — гострий кут,

Якщо $b^2 + c^2 - a^2 = 0$, кут α — прямий кут,

Якщо $b^2 + c^2 - a^2 < 0$, кут α — тупий кут,

Теорема косинусів для двох інших кутів:

$$c^2 = a^2 + b^2 - 2ab * \cos(\gamma),$$

$$b^2 = a^2 + c^2 - 2ac * \cos(\beta).$$

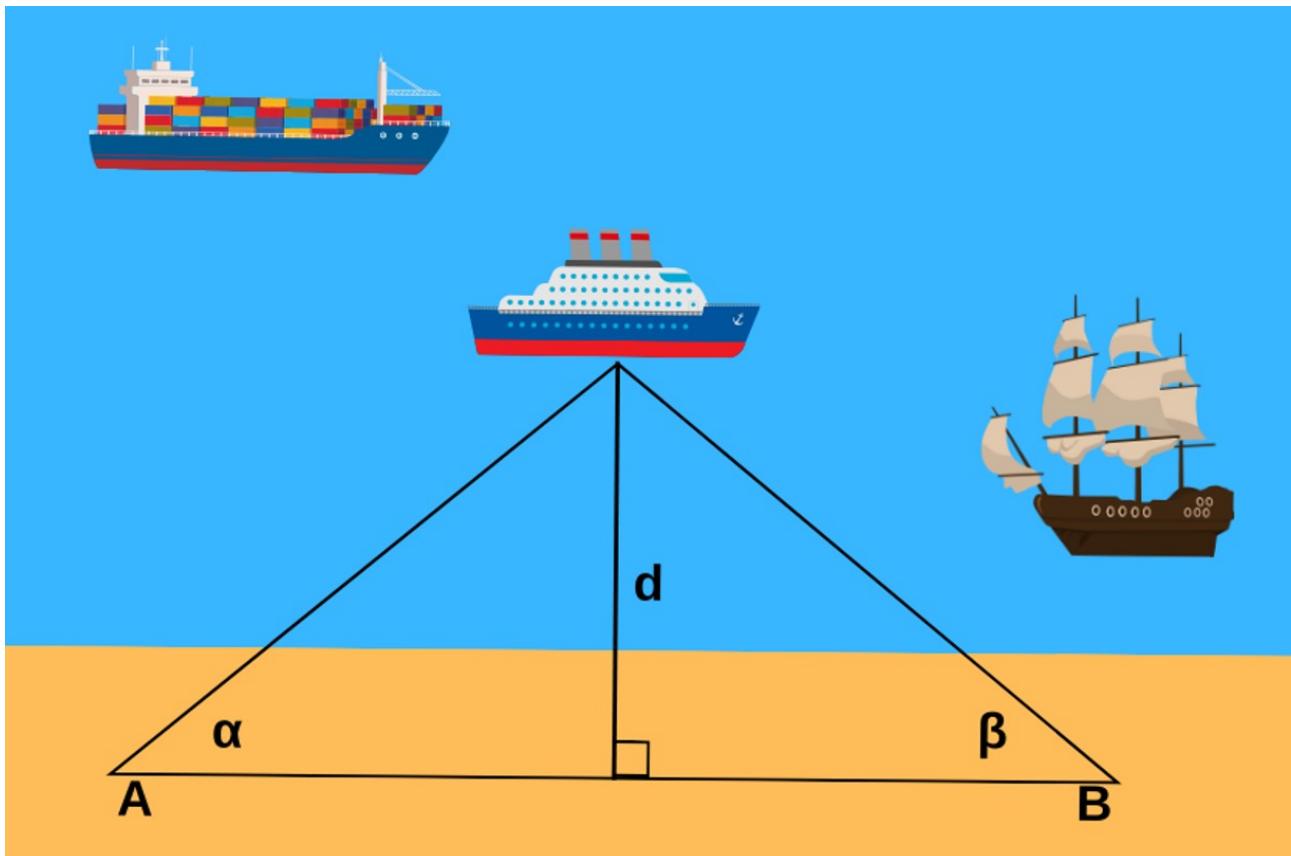
Кути можна виразити з них і за основною формулою:

$$\alpha = \arccos((b^2 + c^2 - a^2) / 2bc),$$

$$\beta = \arccos((a^2 + c^2 - b^2) / 2ac),$$

$$\gamma = \arccos((a^2 + b^2 - c^2) / 2ab).$$

Тріангуляція є одним із методів створення мережі геодезичних контрольних пунктів. Полягає в геодезичній побудові системи точок на місцевості, що утворюють трикутники, в яких вимірюються всі кути та довжини деяких базових сторін.



Тріангуляція може бути використана для визначення положення корабля, коли відомі положення A і B. Спостерігач у точці A вимірює кут α , а спостерігач у точці B вимірює β .

Положення будь-якої вершини трикутника можна обчислити, якщо відомі положення однієї сторони та двох кутів. Наступні формули суворо правильні тільки для плоскої поверхні. Якщо потрібно враховувати кривину Землі, то слід використовувати сферичну тригонометрію.

Розрахунок

Якщо l є відстанню між A і B, ми маємо:

$$l = d/\tan(a) + d/\tan(b)$$

Використовуючи тригонометричні тотожності $\tan \alpha = \sin \alpha / \cos \alpha$ і $\sin(\alpha + \beta) = \sin(\alpha) \cos(\beta) + \cos(\alpha) \sin(\beta)$, це еквівалентно:

$$l = d(\cos(a)/\sin(a) + \cos(b)/\sin(b)) = d(\sin(a+b)/(\sin(a)\sin(b)));$$

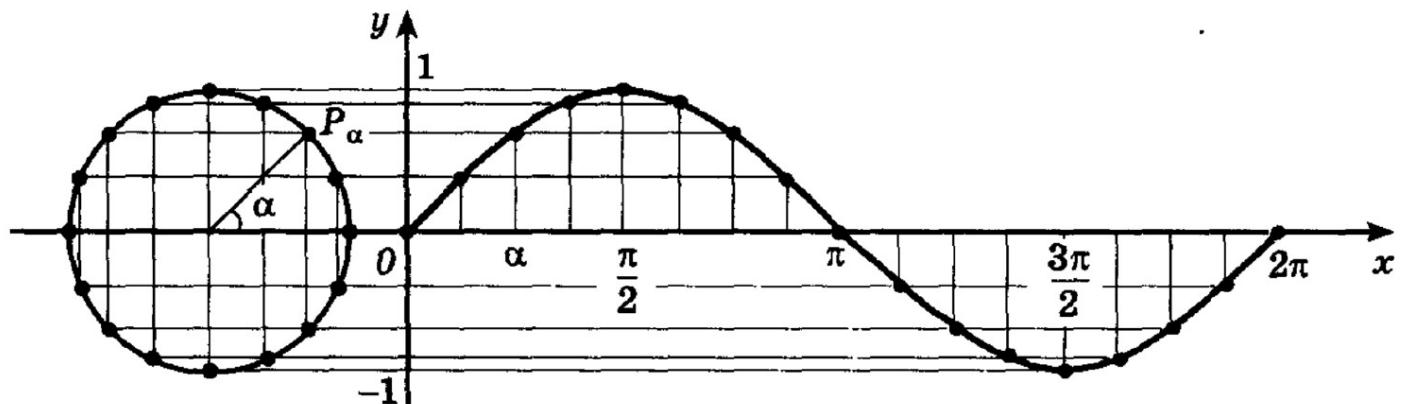
тому:

$$d = l(\sin(a)\sin(b)/\sin(a+b));$$

З цього легко визначити відстань невідомої точки від будь-якої точки спостереження, її зміщення північ/південь та схід/захід від точки спостереження і, нарешті, її повні координати.

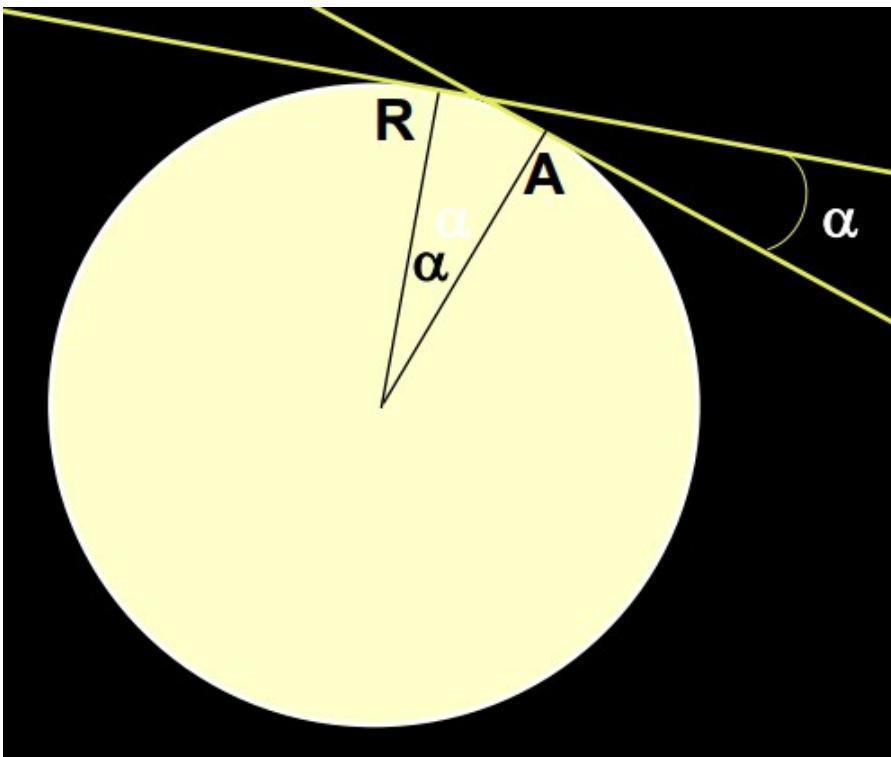
Тангенсом (tan) кута називається відношення довжини протилежного катета до довжини прилеглого катета.

Німецький художник і математик Альбрехт Дюрер (1471 - 1528) у своїй книзі “Установи геометрії” (1535) намалював графік функції синуса, хоча, звичайно, поняття функції в той час не використовувалося. Варто зазначити, що Дюрер фактично використовує аналітичну геометрію, суть якої в тому, що у є функцією від x .



В 1 столітті до нашої ери жив давньогрецький філософ Посідоній, який запропонував спосіб виміру радіуса Землі. Суть способу така: Наведіть пряму паралельну до Землі на певну зірку, пройдіть на південь, наприклад, 800 кілометрів, і виміряйте кут знов між цією зіркою і прямою, що паралельна до Землі в новій точці, він і буде кутом між двома радіусами Землі, що ведуть до першої вашої точки і до точки через 800 кілометрів, тому один кут земного кола дорівнює $800/a$, де a вимірюний кут. Кут між прямыми, що перпендикулярні двом різним радіусам кола, буде рівний куту між самими радіусами.

Метод Посідонія.



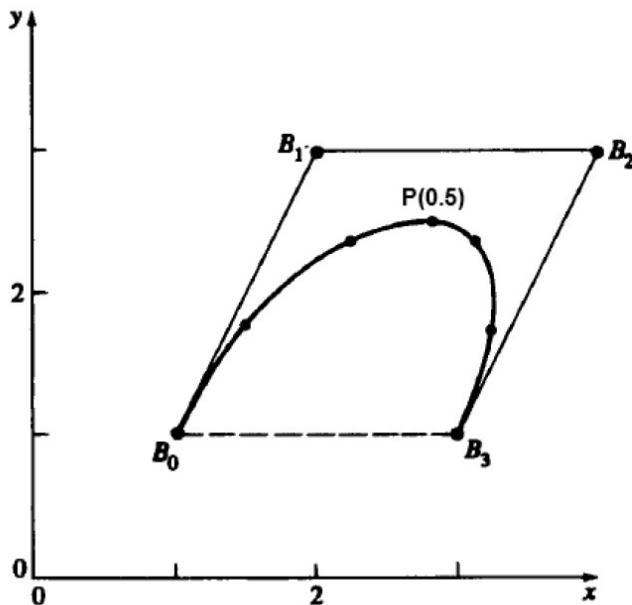
Посідоній відомий своєю спробою визначити розміри земної кулі (попередня спроба належить Ератосфену, і вони описані у творі Клеомеда). За оцінкою Посідонія, Канопус височив в Александрії на $1/48$ кола, на Родосі він був показаний тільки на горизонті, а лінійна відстань між цими містами (які Посідоній помилково вважав розташованими на одному меридіані) був оцінений у 5000 стадій. Канопус — найяскравіша зоря у південному сузір'ї Кіля та друга за яскравістю зірка на нічному небі (після Сіріуса). Отже, коло Землі становило, за оцінкою Посідонія, 240 000 стадій (існують різні значення стадій: від 172,5 до 230,4 метра), що дуже близько до реального кола (периметра) Землі 40 000 км.

Крива Безье

Крива Безье — це параметрично задана крива, яка використовується для представлення кривих у комп'ютерній графіці та дизайні. Вона отримала свою назву на честь французького інженера П'єра Безье.

Ця крива визначається за допомогою контрольних точок (анкерних точок), які визначають форму та напрямок кривої. Крива Безье може бути простою лінією між двома точками або складною кривою, що проходить через багато контрольних точок. Вона широко використовується у графічних редакторах для створення плавних та асиметричних форм, таких як криві векторного зображення.

Нехай дано вершини багатокутника Безье: $B_0[1, 1]$, $B_1[2, 3]$, $B_2[4, 3]$, $B_3[3, 1]$. Потрібно знайти 3 точки, що лежать на відповідній кривій Безье.



Крива Безье задаються рівнянням:

$$P(t) = \sum_{i=0}^n B_i (C(n,i) t^i (1-t)^{n-i}).$$

$$\text{Де, } C(n, k) = n! / ((n-k)! * k!),$$

t - умовно позначає час від початку ковзання.

В нашому випадку, $n = 3$, бо маємо 4 вершини.

Виконаємо необхідні розрахунки:

$$P(0) = B_0 = [1, 1];$$

$$P(0.5) = 0.125B_0 + 0.375B_1 + 0.375B_2 + 0.125B_3 = [2.75, 2.5].$$

$$P(1) = B_3 = [3, 1];$$

Функція для знаходження точок кривої Безье, мовою JavaScript

```
function bezierCurve(t, controlPoints) {
    const n = controlPoints.length - 1;

    function calculateBernstein(i, n, t) {
        const binomialCoefficient = factorial(n) / (factorial(i) * factorial(n - i));
        return binomialCoefficient * Math.pow(t, i) * Math.pow(1 - t, n - i);
    }

    function factorial(k) {
        if (k === 0 || k === 1) {
            return 1;
        }
        return k * factorial(k - 1);
    }

    var p = { x: 0, y: 0 };

    for (let i = 0; i <= n; i++) {
        const bernstein = calculateBernstein(i, n, t);
        p.x += bernstein * controlPoints[i].x;
        p.y += bernstein * controlPoints[i].y;
    }

    return p;
}

// Приклад використання з трьома контрольними точками
const controlPoints = [
    { x: 50, y: 250 },
    { x: 150, y: 50 },
    { x: 350, y: 50 },
    { x: 450, y: 250 }
];

for (var t = 0; t <= 1; t += 0.01) {
    const point = bezierCurve(t, controlPoints);
    // Використовуйте координати точки за потреби, наприклад, малюйте на полотні
}
```

Властивості кривої Безье:

1. безперервність заповнення сегмента між початковою та кінцевою точками;
2. крива завжди розташовується всередині фігури, утвореної лініями, що з'єднують контрольні точки;
3. при наявності лише двох контрольних точок сегмент являє собою пряму лінію;
4. пряма лінія утворюється лише тоді, коли контрольні точки розташовані на одній прямій;
5. крива Безье симетрична, тобто обмін місцями між початковою та кінцевою точками (зміна напрямку траекторії) не впливає на форму кривої;
6. масштабування та зміна пропорцій кривої Безье не порушує її стабільності, оскільки вона з математичної точки зору “аффінно інваріантна”;
7. зміна координат хоча б однієї з точок веде до зміни форми всієї кривої Безье;
8. будь-який частковий відрізок кривої Безье також є кривою Безье;
9. степінь кривої завжди на одиницю менший від кількості контрольних точок. Наприклад, при трьох контрольних точках форма кривої — парабола;
10. коло не може бути описане параметричним рівнянням кривої Безье;
11. крива Безье є окремим випадком поліномів Бернштейна, описаних радянським математиком Сергієм Бернштейном в 1912 році.

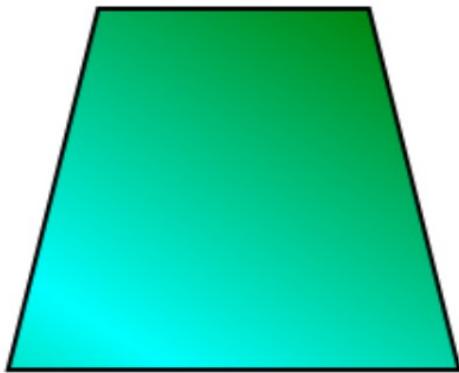
Формула лінійної перспективи

Лінійна перспектива вивчає способи побудови перспективних зображень ліній контуру предметів на поверхні проекцій — площині.

Формула мовою JavaScript:

```
const plane = {  
    position: { x: marginX, y: marginY },  
    topLeft: { x: -100, y: -100, z: 200 },  
    topRight: { x: 100, y: -100, z: 200 },  
    bottomRight: { x: 100, y: 100, z: 0 },  
    bottomLeft: { x: -100, y: 100, z: 0 },  
};  
  
// Function to project 3D point to 2D  
function project3DTo2D(x, y, z) {  
    const perspectiveFactor = 300;  
    const scale = perspectiveFactor / (z + perspectiveFactor); // Perspective projection  
    const projectedX = plane.position.x + x * scale;  
    const projectedY = plane.position.y + y * scale;  
    return { x: projectedX, y: projectedY };  
}  
  
const topLeft = project3DTo2D(plane.topLeft.x, plane.topLeft.y, plane.topLeft.z);  
const topRight = project3DTo2D(plane.topRight.x, plane.topRight.y, plane.topRight.z);  
const bottomRight = project3DTo2D(plane.bottomRight.x, plane.bottomRight.y, plane.bottomRight.z);  
const bottomLeft = project3DTo2D(plane.bottomLeft.x, plane.bottomLeft.y, plane.bottomLeft.z);
```

Результат вище описаної формули



Зберігся твір "Оптика" арабською мовою, що приписується Евкліду. В "Оптиці" Евклід вивчає властивості людського зору з допомогою геометрії, наприклад, він пояснює чому виникає перспектива, чому далекі предмети видно гірше, тобто з меншою роздільною здатністю, чому ми бачимо весь предмет не одразу, але цього не помічаемо, чому тіла, що рухаються в далечі здаються повільнішими ніж є насправді.

Евклід ввів поняття "конус зору". Евклід пропонує уявити, що зіниця — це як би вершина конуса, а основа конуса — лінія горизонту.

Уявлення про те, що двовимірна картина це по суті перетин конуса зору мав італійський архітектор Філіппо Брунеллескі (1377 — 1446). Достатньо якісно перспектива зображена на картині "Пір Ірода" італійського художника Філіппо Ліппі (1406 — 1469).

П'єро делла Франческа (1420—1492) — італійський художник, автор відомих на той час трактатів з геометрії та теорії перспективи. Вважається що стиль делла Франчески сформувався під впливом флорентійської школи, Мазаччо, Філіппо Брунеллескі, а також нідерландського живопису. Картина "Ідеальне місто" (1480), яку вважають твором П'єро делла Франческа, виділяється активним використанням теорії перспективи.

Леонардо да Вінчі застосовував теорію перспективи у своїй фресці "Таємна вечірня" (1497). Леонардо почав використовувати повітряну перспективу, зокрема, ефект коли об'єкт в далечі малюється бліднішим, бо перед ним більший шар повітря (і також через Релеївське розсіювання). Ключовими поняттями теорії перспективи є: конус зору і його перетин, сітка деформацій, нескінченно віддалена точка. Німецький художник Альбрехт Дюрер опублікував працю "Керівництво до вимірювання циркулем та лінійкою" (1525) з цікавими ілюстраціями щодо художньої перспективи.

Рене Декарт в "Діоптриці" чітко описав систему ока.

Світло від верхньої частини об'єкта потрапляє на нижню частину сітківки, а світло від лівої частини об'єкта потрапляє на праву частину сітківки, тобто зображення на сітківці інвертоване по вертикалі й горизонталі.

В 1928 році львівський професор Казимир Бартель опублікував твір "Малярська перспектива" (Художня перспектива) в якому з допомогою геометрії розповідає про перспективу.

Які найпопулярніші алгоритми в топології?

1. Топологічна рівність: У топології два простори вважаються топологічно еквівалентними (або гомеоморфними), якщо існує неперервна біекція (гомеоморфізм) між ними. Гомеоморфізми зберігають топологічні властивості, що означає, що якщо два простори гомеоморфні, то вони мають однакові топологічні особливості. Наприклад, коло і квадрат є топологічно еквівалентними, оскільки ви можете безперервно перетворити одне в інше без рвання або склеювання.
2. Характеристика Ейлера: Характеристика Ейлера, позначена символом χ (читається як "хі"), є топологічним інваріантом, який описує форму топологічного простору, зазвичай компактної, зв'язаної та орієнтованої поверхні. Для поверхні χ можна обчислити за формулою: $\chi = V - E + F$, де V представляє кількість вершин, E - кількість ребер і F - кількість граней в комбінаторному поділі поверхні. Характеристика Ейлера одна така для всіх топологічно еквівалентних поверхонь, що робить її корисним інструментом у топології.
Поряд із поняттям метричного простору, топологічний простір є одним з різновидів просторів у геометрії.
3. Теорема Брауера про фіксовану точку: Ця теорема названа на честь нідерландського математика Лейтзена Брауера і є фундаментальним результатом в алгебраїчній топології. Вона стверджує, що будь-яка неперервна функція від закритого інтервалу до себе має принаймні одну фіксовану точку. Іншими словами, для будь-якої функції, яка бере точку в інтервалі і відображає її назад в цей інтервал, існує принаймні одна точка, яка не зміщується; тобто $f(x) = x$ для деякого x в інтервалі. Ця теорема має важливі наслідки в різних галузях, включаючи економіку, теорію ігор і диференціальні рівняння, і є ключовим результатом в теорії топології.

Характеристика Ейлера

У математиці, а точніше в алгебраїчній топології та комбінаториці багатогранників, характеристика Ейлера (або число Ейлера) є топологічним інваріантом, числом, яке описує форму або структуру топологічного простору незалежно від способу його вигину. Його зазвичай позначають χ . Леонард Ейлер, на честь якого названо це поняття, ввів його для багатогранників, але не зміг суворо довести, що воно є інваріантом. Характеристика Ейлера χ була класично визначена для поверхонь багатогранників за формулою $\chi = V - E + F$, де V , E і F — відповідно кількість вершин (кутів), ребер і граней даного багатогранника. Поверхня будь-якого опуклого багатогранника має характеристику Ейлера: $V - E + F = 2$. Це рівняння, сформульоване Леонардом Ейлером (Ойлером) у 1758 році, відоме як формула багатогранника Ейлера. Це відповідає ейлеровій характеристиці кулі (тобто $\chi = 2$) і однаково застосовується до сферичних багатогранників.

Теорема Ейлера для багатогранників — це теорема, яка встановлює зв'язок між кількістю вершин, ребер і граней для багатогранників, які топологічно еквівалентні кулі. Нехай V — кількість вершин опуклого багатогранника, P — кількість його ребер, а F — кількість граней. Тоді рівність $V - P + F = 2$.

У 1620 р. Рене Декарт показав, що сума кутів усіх граней багатогранника дорівнює $360 * (P - F)$ і $360 * (V - 2)$ одночасно. У 1750 році Леонард Ейлер довів тотожність для опуклих багатогранників. Теорема Ейлера закладає основу нової галузі математики — топології. Більш суворе доведення надав Коші в 1811 році. Довгий час вважалося, що співвідношення Ейлера справедливе для будь-якого багатогранника. Перший контрприклад дав Сімон Л'Юльє в 1812 році; розглядаючи колекцію мінералів, він звернув увагу на прозорий кристал польового шпату, всередині якого знаходився чорний кубічний кристал сульфіду свинцю. Л'Юльє зрозумів, що куб з кубічною порожниною

всередині не підкоряється формулі Ейлера. Пізніше були виявлені інші контрприклади (наприклад, два тетраедри, склеєні вздовж ребра або мають спільну вершину), і формулювання теореми було уточнено: це справедливо для багатогранників, які топологічно еквівалентні кулі (образно кажучи, якщо можна їх накачати так, щоб вони набули форми кулі, якби були з гуми).

Німецький математик Йоган Бенедикт Лістинг (1808 - 1882) ввів слово “топологія” і заклав основи математичної теорії вузлів.

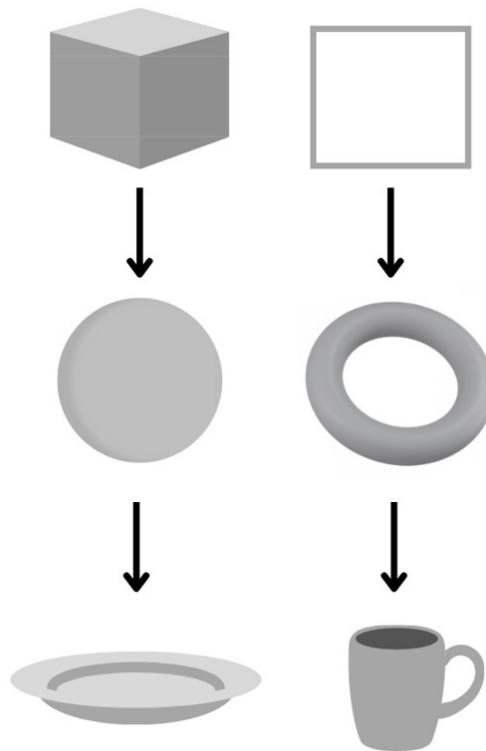
У математиці топологія (від грецьких слів топос, “місце” та логос, “наука”) займається властивостями геометричного об’єкта, які зберігаються при безперервних деформаціях, таких як розтягнення, скручування, змінання та вигин, тобто без закриття отворів, відкриття отворів, розриву, склеювання або проходження крізь себе.

Французький математик Каміль Жордан (1838 - 1922) зауважив, що на поверхні, яку можна деформувати без розривів, наприклад, розтягувати, замкнена крива може втратити свої початкові ознаки, але все ж буде ділити площину на внутрішню й зовнішню частини, тобто деякі точки будуть в середині замкненої кривої, а інші за її межами.

Топологічні простори досліджуються без використання метрики (відстані), і їх властивості визначаються за допомогою відкритих множин та їхніх взаємозв'язків. У метричних просторах, навпаки, відстань між точками визначена метрикою, і вони досліджуються за допомогою цих метричних властивостей. Таким чином, топологічні простори є більш загальним поняттям, ніж метричні простори, і вони можуть мати більш різноманітні структури.

Гомеоморфізм (грец. гоміос — схожий, морфі — форма) — взаємно однозначне і взаємно безперервне відображення топологічних просторів. Гомеоморфні простори топологічно однакові.

На малюнку фігури в стовпцях гомеоморфні, а в рядах ні.

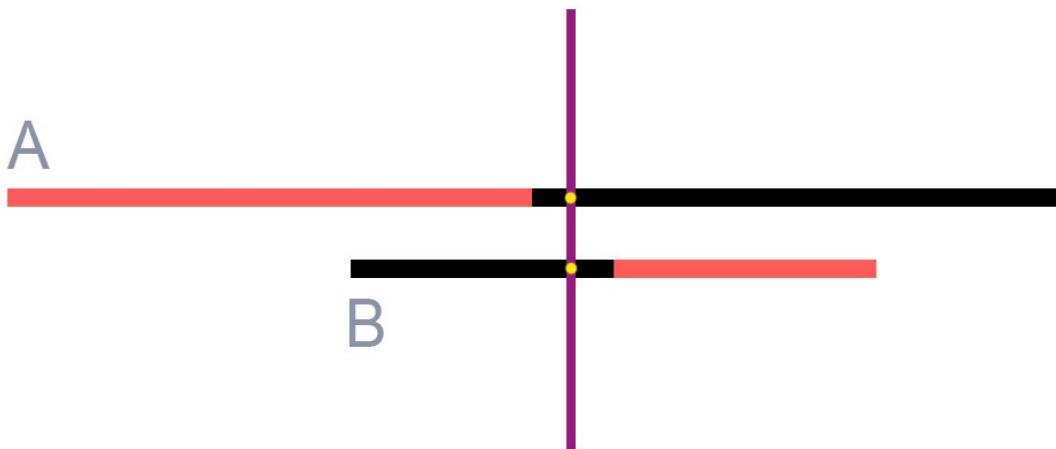


Теорема Брауера про нерухому точку

Зазвичай теорема формулюється так: Будь-яке безперервне відображення замкнutoї кулі в себе в скінченнонімірному евклідовому просторі має нерухому точку. Більш детально розглянемо замкнену кулю в n -вимірному просторі $B^n \subset R^n$. Нехай $f: B^n \rightarrow B^n$ — деяке безперервне відображення цієї кульки в себе (не обов'язково строго всередину, не обов'язково біективне, тобто навіть не обов'язково сюр'ективне). Тоді існує така точка $x \in B^n$, що $f(x) = x$. Теорему довів голландський математик Лейтzen Егберт Ян Брауер (1881-1966).

На площині: Будь-яке неперервне відображення f із замкнутого диска в себе допускає принаймні одну нерухому точку.

На малюнку показано два одинакових відрізки, один з них ми перевернули по горизонталі та стиснули, але очевидно, що все ж в них буде місце де вони зійдуться однією й тією ж точкою.



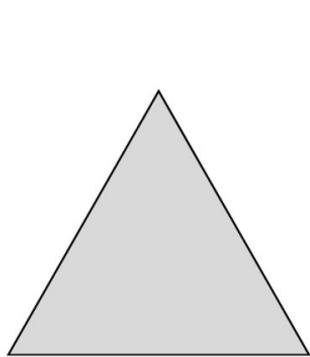
Цікавий розсуд. Уявіть собі, що автобус ходить з 12 ранку до 12 вечора, тобто 12 годин. Автобус відправляється з пункту А рівно о 12 годині ночі та прибуває рівно о 12 годині в пункт В. Якщо ви їдете тією ж дорогою, що й автобус, і прибуваєте рівно о 12 годині, як автобус, не має значення, коли ви виїхали з пункту А і з якою швидкістю ви їхали, ви обов'язково зустрінетесь з автобусом по дорозі, а отже, в один і той же час дня будете в одній точці шляху. Подібний доказ описано в науково-популярній книзі Мартіна Гарднера з математики.

Стефан Банах у 1922 році довів теорему про нерухому точку. Теорема Банаха про нерухому точку: Всяке стискальне відображення повного метричного простору в себе має єдину нерухому точку (яку можна знайти методом послідовних наближень, починаючи з будь-якої точки цього простору).

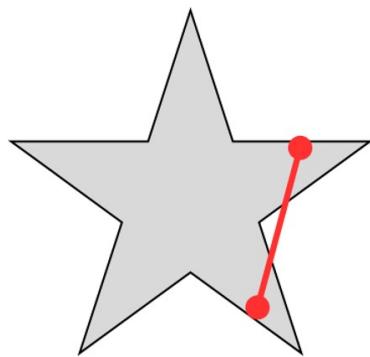
Компактна множина в метричному просторі (або топологічному просторі) — це множина, яка є обмеженою і замкненою. Іншими словами, множина є компактною, якщо вона обмежена (знаходитьться всередині деякого сферичного об'єкта) і вона є замкнutoю (містить всі свої граничні точки). Відрізок $[0, 1]$ є компактним, оскільки він є обмеженою (міститься між 0 і 1) і закритою (включає крайні точки 0 і 1) множиною на числовій прямій.

Опукла множина в афінному або векторному просторі — це множина, в якій усі точки відрізка, утвореного будь-якими двома точками даної множини, також належать цій множині.

Алгоритм Джарвіса (або алгоритм упаковки подарунків) визначає послідовність елементів набору, які утворюють опуклу оболонку для цього набору. Алгоритм названий на честь Р. Джарвіса, який опублікував його в 1973 році.



опукла множина



неопукла множина

Які найпопулярніші алгоритми в теорії графів?

Як знайти вихід з лабіринту?

У давнину багато хто вважав, що зі складного лабіринту взагалі неможливо вибратися, якщо немає нитки Аріадни. Хоча навіть з такою ниткою впоратися з деякими лабірінтами буде складно без алгоритму виходу з лабіринту. Згідно з грецьким міфом, цар Мінос сховав чудовисько Мінотавра в лабірінті на Криті, побудованому легендарним інженером Дедалом в місті Кносс (його сином був Ікар), куди людей кидали, щоб їх пожерло чудовисько. Дедал дійова особа одноїменної сатирівської драми Софокла.

Мінотавр — чудовисько з тілом людини та головою бика. Згідно з давньогрецьким міфом, коли Тесей, син афінського царя Егея, вирішив убити Мінотавра, якому афіняни, на прохання Міноса, щорічно надсилали данину з семи юнаків і семи дівчат, він отримав клубок нитки від закоханої в нього Аріадни (дочки Міноса), який вивів його з лабіринту, де жив Мінотавр (Дедал навчив Аріадну користуватися ниткою). Один кінець нитки був зав'язаний біля входу в лабірінт, а потім вони йшли, розмотуючи клубок, що давало їм можливість повернутись до початку. Стіни та дороги лабіринту не мали знаків, вони були однорідними. Легенда про Кносський лабірінт Дедала описана Овідієм у 8 книзі “Метаморфоз” і Плутархом у біографії Тесея. Виявляється, щоб вибратися з лабіринту, не потрібно володіти надприродними здібностями або хитромудрими артефактами, достатньо знати правильний алгоритм (тобто набір дій) для виходу. Математики 18-19 століття розробили алгоритм виходу з лабіринту.

Класичний лабірінт складається всього з трьох складових: перехрестя, петлі, тупики. Лабірінт можна представити у вигляді математичного графа, тому Ейлер, ймовірно, знову алгоритм виходу з лабіринту. Класичний алгоритм можна представити у вигляді графу з циклами (якщо в лабірінті є петлі).

Щоб вибратися з класичного лабіринту, достатньо виконати наступний алгоритм:

1. Ідіть вперед і позначте свій шлях стрілками (доріжку можна позначити крейдою на стіні або насічками). При поворненні по пройденій дорозі назад стрілки малювати не потрібно.
2. Якщо ви потрапили в глухий кут, то поверніться (за стрілками) до найближчого перехрестя й заблокуйте шлях у глухий кут (закресліть).
3. Якщо ви йшли новим шляхом і зустріли стрілку, тобто шлях, по якому ви вже йшли, то перегородіть цю дорогу з двох кінців, один кінець, що йде до дороги, на яку ви вийшли, а інший кінець по дорозі назад на найближчому до вас перехресті. Тобто поверніться до найближчого до вас перехрестя і перегородіть шлях по цій дорозі (з обох сторін).
4. Продовжуйте виконувати кроки 1-3, доки не знайдете шлях, що веде до виходу.

Цей алгоритм базується на алгоритмі під назвою “пошук у глибину”.

Часова складність цього алгоритму лінійна й залежить від кількості вершин та ребер графа, тобто $O(|V| + |E|)$, де $|V|$ - кількість вершин, $|E|$ - кількість ребер.

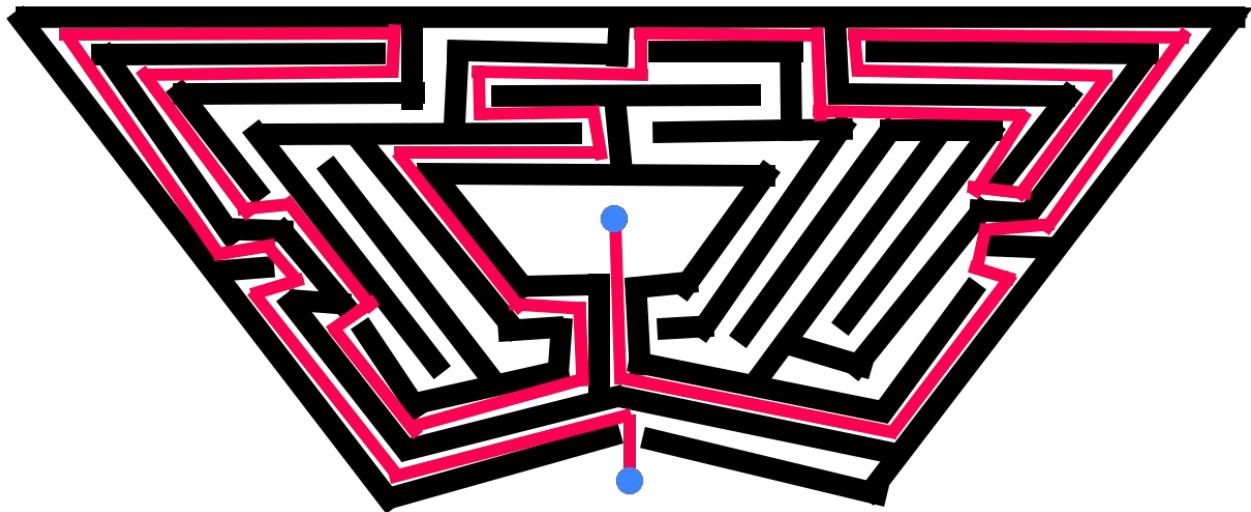
Дерево Тремо неоріентованого графа G — це кістякове дерево графа G з виділеним коренем зі властивістю, що будь-які дві суміжні вершини в графі G пов'язані відношенням предок/нащадок. Всі дерева пошуку в глибину і всі гамільтонові шляхи є деревами Тремо. Дерева Тремо названо на честь Чарльза П'єра Тремо, французького автора XIX століття, який використовував варіант пошуку в глибину як стратегію виходу з лабіринту.

Гамільтонів граф — в математиці це граф, що містить гамільтонів цикл.

Гамільтонів шлях — шлях, що містить кожну вершину графу рівно один раз. Гамільтонів шлях, початкова і кінцева вершини якого збігаються, називається гамільтоновим циклом.

Гамільтонові шляхи, цикл і граф названі на честь ірландського математика Вільяма Гамільтона.

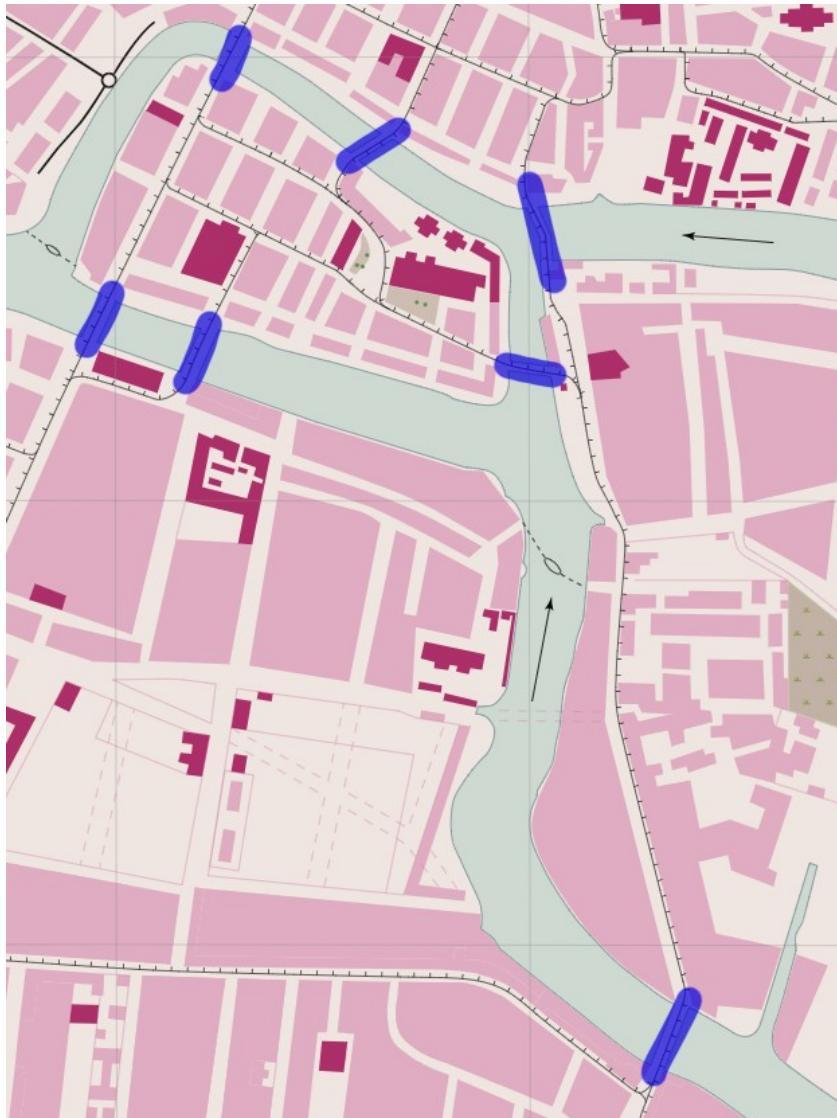
В 1950 році американський програміст і математик Клод Шенон розробив автомат, який міг знаходити вихід з лабіринту і назвав його “Тесей”. Це була механічна машинка у вигляді миші, яка могла рухатись по лабіринту з 25 квадратів. Конфігурацію лабіринту можна було довільно змінювати шляхом перестановки рухомих перегородок. Пройшовши лабіринт, миша могла бути розміщена в будь-якому місці, де вона була раніше, і завдяки своєму попередньому досвіду вона могла йти прямо до цілі (виходу). Під корпусом лабіринту розміщувався комп'ютер на основі електромеханічних реле, який запам'ятовував рух миші по лабіринту.



Hampton Court Maze (UK)

Проблема семи Кенігсберзьких мостів

Здавна серед мешканців Кенігсберга поширина така загадка: як пройти через усі міські мости (через річку Преголю), не пройшовши жодним із них двічі. Багато хто намагався розв'язувати цю проблему як теоретично, так і практично, під час ходьби. Однак довести чи спростувати можливість такого маршруту ніхто не міг.

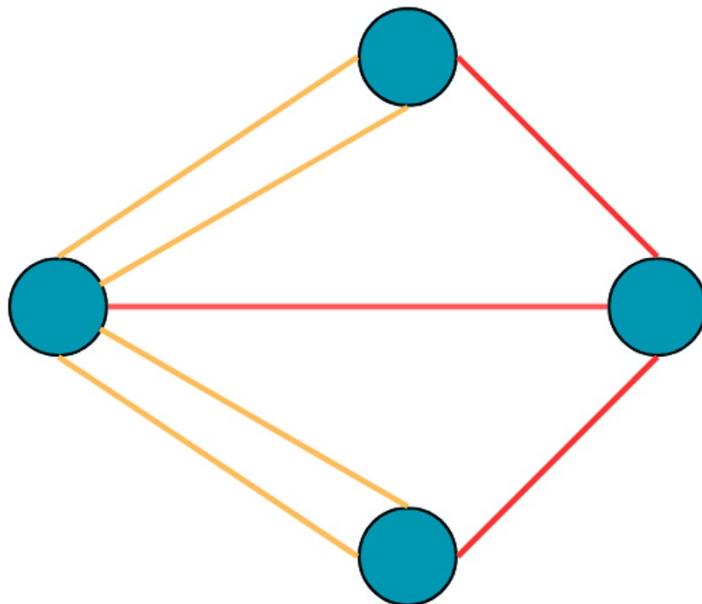


Карта Кенігсберга з мостами в 1905 році (нині не всі мости збереглись). Ліцензія зображення [CC BY-SA 3.0](#).

У 1736 р. проблема семи мостів зацікавила видатного математика, члена Петербурзької академії наук Леонарда Ейлерса, про що він писав у листі до італійського математика й інженера Йоганна Якоба Маріоні від 13 березня 1736 р. У цьому листі Ейлер дає правило, з допомогою якого легко визначити, чи можна перетнути всі мости, не перетнувши жодного з них двічі. У цьому випадку

відповідь була “ні”. У листі до Карла Готліба Елера від 3 квітня 1736 року Ейлер обґруntовує знайдене ним правило, а пізніше на цю тему Ейлер публікує статтю в науковому журналі Петербурзької Академії наук.

Розв’язання задачі за Леонардом Ейлером.



На спрощеній схемі міста (графа) мости відповідають лініям (ребрам графа), а частинам міста — точкам з'єднання ліній (вершинам графа). У ході своїх міркувань Ейлер прийшов до наступних висновків: кількість непарних вершин (вершин, до яких веде непарна кількість ребер) графа має бути парною. Не може бути графа з непарною кількістю непарних вершин. Якщо всі вершини графа парні, то можна намалювати цей граф, не відригаючи олівця від паперу, і можна почати з будь-якої вершини графа і закінчити його в цій же вершині. Якщо рівно дві вершини графа непарні, то можна намалювати цей граф, не відригаючи олівця від паперу, при цьому потрібно починати з однієї з непарних вершин і закінчувати її в іншій непарній вершині. Граф з більш ніж двома непарними вершинами не можна намалювати одним штрихом. Граф Кенігсберзьких мостів мав чотири непарні вершини (тобто всі) — отже, неможливо пройти через усі мости, не пройшовши по жодному з них двічі.

Теорія графів стала дуже розвиненою. Щоб знайти найкоротші шляхи на графах, використовуйте алгоритм Пріма, Крускала, Дейкстри. Відомі графи та цикли Гамільтона й Ейлера.

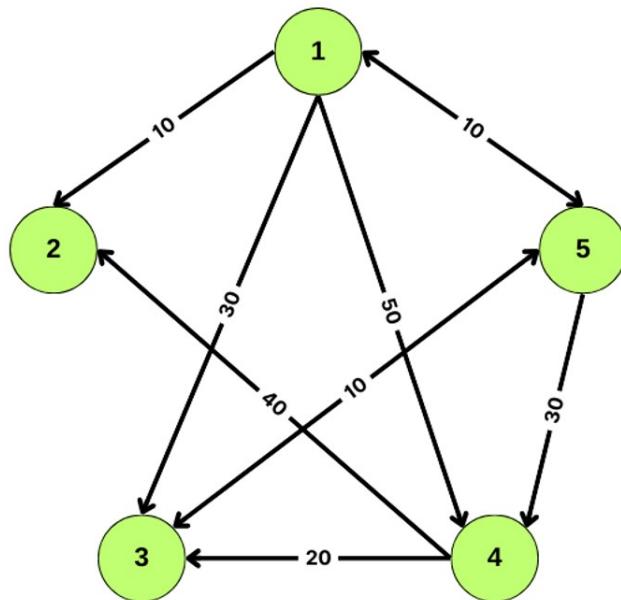
Граф — це скінчена множина V , яка називається множиною вершин, і скінчена множина E двоелементних підмножин множини V , яка називається множиною ребер.

Зазвичай кінцевий граф зображують у вигляді діаграми, на якій вершини позначаються точками, а ребра, що з'єднують дві вершини, зображуються лініями між цими точками.

Приклад

Кругами позначені вершини, лініями — шляхи між ними (ребра графа).

Орієнтований граф з циклом, зваженими вершинами та ребрами.



Неорієнтованим графом є $G(V, E)$ або $G = \{V, E\}$, де $V = \{a, b, c, \dots\}$, $E = \{\{a, b\}, \{a, c\}, \{c, b\}, \dots\}$.
 Орієнтованим графом є $G(V, E)$ або $G = \{V, E\}$, де $V = \{a, b, c, \dots\}$, $E = \{<a, b>, <a, c>, <c, b>, \dots\}$. Як бачимо, множина E орієнтованого графа складається з впорядкованих пар, що тим самим визначає напрямок ребра.

Зважений граф за ребрами — граф, кожному ребру якого присвоюється певне значення (вага ребра).
 Наприклад, $G(V, E)$, де $V = \{a, b, c, \dots\}$, $E = \{<\{a, b\}, 7>, <\{a, c\}, 3>, \dots\}$

Зважений граф за вершинами - граф, кожній вершині якого присвоєно певне значення (вага вершини).
 Наприклад, $G(V, E)$, де $V = \{<a, 2>, <b, 2>, <c, 2>, \dots\}$, $E = \{\{a, b\}, \{a, c\}, \{c, b\}, \dots\}$.

Цикл графа – це шлях ненульової довжини, який з'єднує вершину з собою і не містить повторюваних ребер.

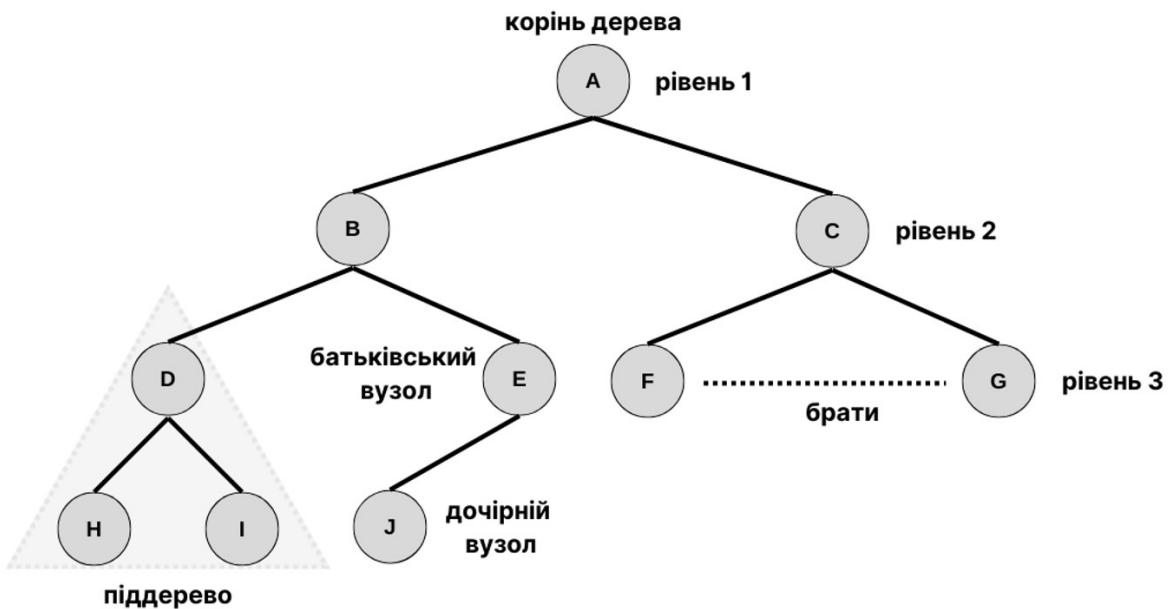
$G = \{\{a, b, c, \dots\}, \{<\{a, b\}, 7>, <\{a, c\}, 3>, \dots\}\}$.

Цикли в графі G можна позначити як $abca$, $c bacb$.

Дерево — це граф без циклів.

Ступінь вершини — це кількість інцидентних їй ребер.

Структура даних “Дерево” (граф)



Якщо ми розглядаємо простий ненаправлений граф без петель, кожне ребро з'єднує дві вершини. Таким чином, у такому графі кількість ребер можна обчислити за формулою:

$$E = (V * (V - 1)) / 2,$$

де E - кількість ребер, а V - кількість вершин у графі.

Задача комівояжера полягає в тому, щоб знайти найбільш вигідний маршрут, який хоча б один раз проходить через вказані міста, а потім повертається в початкове місто. В умовах задачі вказується критерій рентабельності того чи іншого маршруту (найкоротший шлях, найдешевший шлях тощо). Проблему комівояжера можна вирішити з допомогою жадібного алгоритму (якщо порядок відвідування міст чітко заданий).

До жадібних алгоритмів належать алгоритми, в яких на кожному етапі виконується оптимальне рішення даного етапу. Сума оптимальних розв'язків усіх кроків алгоритму обов'язково дає глобальне оптимальне рішення, якщо рішення попереднього етапу не блокує оптимальне рішення наступного етапу. Наприклад, від міста А до міста В є шляхи 4 км і 3 км, а від міста В до міста С є шляхи 5 км і 4 км, але ми знаємо, що шлях від А до С становить 2 км. Якщо нам потрібно їхати послідовно, тобто по порядку, до міст А, В, С, найкоротшим шляхом, то на кожному етапі достатньо вибрати найкоротший шлях з міста до іншого міста. У нашому випадку це А, 3 км, В, 4 км, С. Загальний шлях 7 км. Причому на кожному етапі ми робимо найкращий вибір, тобто вибираємо найменшу відстань. Але якщо нам потрібно прокласти інтернет-кабель між містами А, В, С, щоб витратити найменшу кількість кабелю, то нам потрібно прокласти кабель від міста А до міста С і від С до міста В. Таким чином, А, 2 км, С, 4 км, В. Загалом витратимо 6 км кабелю.

Алгоритм Флойда-Воршелла

Алгоритм Флойда-Воршелла — це алгоритм для знаходження найкоротших шляхів між усіма парами вершин в орієнтованому або неорієнтованому зваженому графі з можливо від'ємними вагами на ребрах. Алгоритм було опубліковано у звичній сьогодні формі Робертом Флойдом 1962 року.

Всі вершини (V) графа пронумеровані від 1 до n .

Основний крок алгоритму полягає в поетапному оновленні матриці відстаней між парами вершин. Кожен раз, коли алгоритм знаходить коротший шлях між двома вершинами через іншу вершину, він оновлює відстань між цими двома вершинами.

Матриця відстаней (матриця суміжності) графа будується на основі зв'язків між вершинами графа. Кожен рядок та кожен стовпець у матриці відповідає одній вершині графа. Якщо між вершинами i та j є зв'язок (ребро), то відповідний елемент матриці буде ненульовим (зазвичай це вага ребра). Якщо зв'язку між вершинами немає, елемент матриці буде умовно нескінченим або відсутнім.

У випадку негативного циклу, сума ваг ребер цього циклу від'ємна. Під час проходження по циклу ітерації алгоритму Флойда-Воршелла може бути знайдена вершина, для якої вартість шляху до неї може бути зменшена до від'ємної величини, що свідчить про наявність негативного циклу. У такому випадку, зазвичай, алгоритм припиняє свою роботу або повідомляє про наявність циклу.

Алгоритм Флойда-Воршелла гарантовано знаходить усі найкоротші шляхи та може робити це зі швидкістю $O(|V|^3)$, де V – кількість вершин графа, навіть якщо в графі може бути $|V|^2$ ребер. Це робиться шляхом поступового покращення оцінки на найкоротшому шляху між двома вершинами, поки оцінка не стане оптимальною.

Ось кроки побудови матриці відстаней в алгоритмі Флойда-Воршелла:

Ініціалізуйте матрицю відстаней так, щоб вона містила відстані між всіма парами вершин. Якщо вершини з'єднані ребром, значення відстані буде вагою цього ребра, якщо ні — велике число, наприклад, нескінченість.

Проходьте по кожній вершині та розглядайте її як проміжну вершину у всіх можливих парах вершин. Перевірте, чи можливо скоротити шлях між цими парами за допомогою поточної проміжної вершини. Якщо так, оновіть відстань в матриці відстаней.

Повторюйте крок 2 для кожної вершини у графі.

Після завершення алгоритму, матриця відстаней буде містити довжини найкоротших шляхів між усіма парами вершин.

Алгоритм Флойда-Воршелла багаторазово змінює довжини шляху між усіма парами вершин (i, j) , включаючи ті, де $i = j$;

Початкова довжина шляху (i, i) рівна 0;

Шлях $i \rightarrow j \rightarrow \dots \rightarrow i$ може покращити початкову довжину, якщо він має довжину меншу за нуль, тобто позначає негативний цикл;

Таким чином, після виконання алгоритму, найкоротший шлях (i, i) буде від'ємним, якщо існує негативний цикл — шлях від'ємної довжини від i назад до i .

Приклад коду на JS:

```
function floydWarshall(graph) {
    const n = graph.length;
    const dist = [...graph];

    for (let k = 0; k < n; k++)
        for (let i = 0; i < n; i++)
            for (let j = 0; j < n; j++)
                dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);

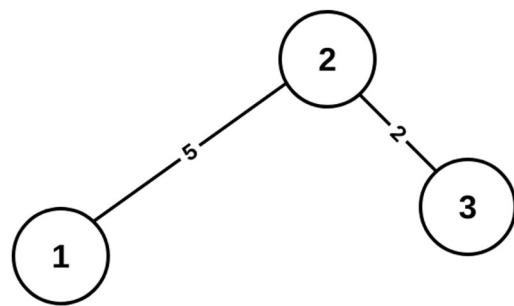
    return dist;
}

const INF = Number.MAX_SAFE_INTEGER;

// Приклад використання
const graph = [
    [0, 5, INF],
    [5, 0, 2],
    [INF, 2, 0],
];
const shortestDistances = floydWarshall(graph);
console.log(shortestDistances);
```

Матриця відстаней для графа на малюнку

```
const graph = [  
    [0, 5, INF],  
    [5, 0, 2],  
    [INF, 2, 0],  
];
```



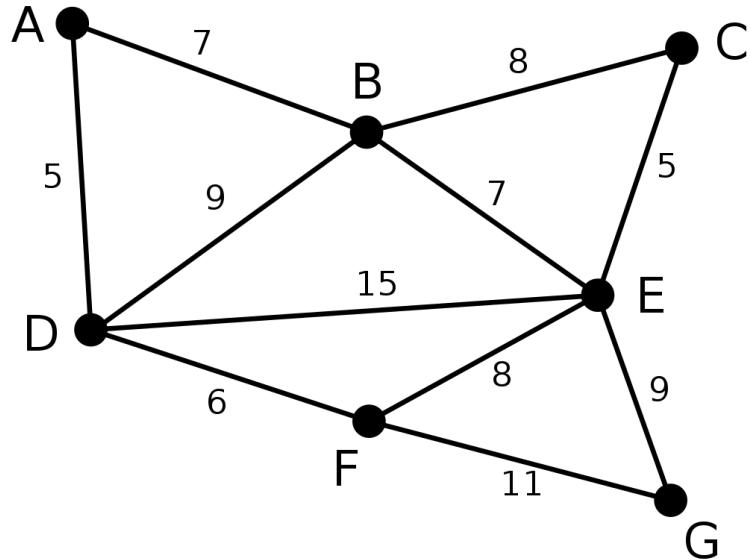
Алгоритм Пріма

Алгоритм Пріма використовується для побудови мінімального кістякового дерева зваженого зв'язного графа. Алгоритм був розроблений американським математиком Робертом Прімом у 1957 році.

Принцип алгоритму

На вхід алгоритму подається зв'язний неоріентований граф. Для кожного ребра встановлюється його вартість. Спочатку береться довільна вершина і виявляється ребро, яке інцидентне цій вершині та має найменшу вартість. Знайдене ребро і дві з'єднані ним вершини утворюють дерево. Потім розглядаються ребра графа, один кінець якого є вершиною, що вже належить дереву, а інший — ні, з цих ребер вибирається ребро найменшої вартості. Ребро, яке ви обираєте на кожному кроці, прикріплюється до дерева. Зростання дерева продовжується до тих пір, поки не будуть вичерпані всі вершини вихідного графа. Результатом роботи алгоритму є охоплююче дерево мінімальних витрат.

Приклад



1. Дано початковий зважений графік $G(V, E)$, де:

$$V = \{A, B, C, D, E, F, G\},$$

$$E = \{\langle A, B \rangle, 7\rangle, \langle C, B \rangle, 8\rangle, \langle A, D \rangle, 5\rangle, \langle C, E \rangle, 5\rangle, \langle B, D \rangle, 9\rangle, \langle B, E \rangle, 7\rangle, \langle E, D \rangle, 15\rangle, \langle D, F \rangle, 6\rangle, \langle E, F \rangle, 8\rangle, \langle F, G \rangle, 11\rangle, \langle E, G \rangle, 9\rangle\}$$

Числа біля ребер показують їх ваги, які можна розглядати як відстань між вершинами.

2. Початковою довільно вибираємо вершину D. Кожна з вершин A, B, E і F сполучена з D одним ребром. Вершина A є найближчою вершиною до D і вибирається як друга вершина разом з ребром AD.

3. Наступна вершина є найближчою вершиною до будь-якої з вибраних вершин D або A. В дорівнює 9 від D і 7 від A. Відстань до E дорівнює 15, а до F дорівнює 6. F є найближчою вершиною, тому вона входить в дерево F разом з ребром DF.

4. Аналогічно вибирається вершина B на відстані 7 від A.

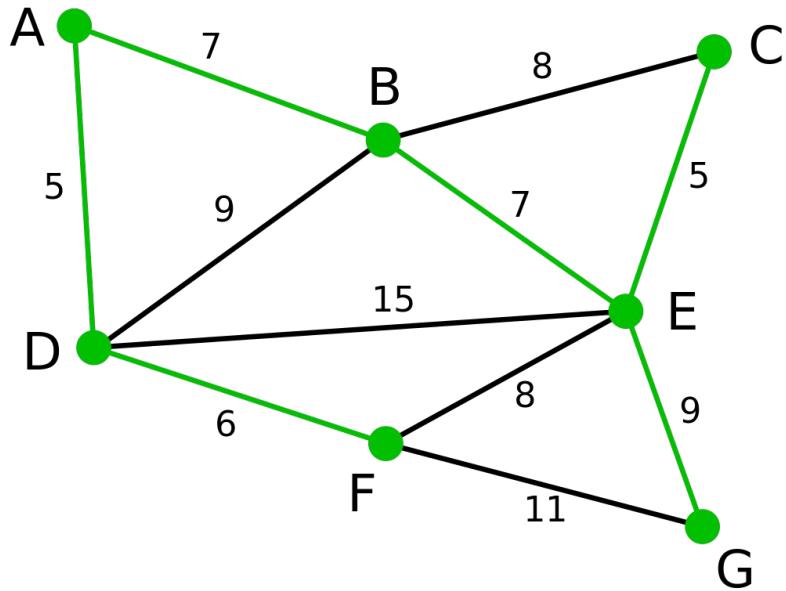
5. У цьому випадку можна вибрати або C, або E, або G. С знаходиться на відстані 8 від B, E — на 7 від B, а G — на 11 від F. E — найближча вершина, тому E та BE вибираються.

6. Тут доступні лише вершини C і G. Відстань від E до C дорівнює 5, а до G - 9. Вибрано вершину C і

ребро EC.

7. Залишилася лише вершина G. Відстань від F до неї дорівнює 11, від E - 9. Е ближче, тому вибираються вершина G і ребро EG.

8. Вибрано всі вершини, будується мінімальне кістякове дерево (виділено зеленим). В цьому випадку його вага становить 39.

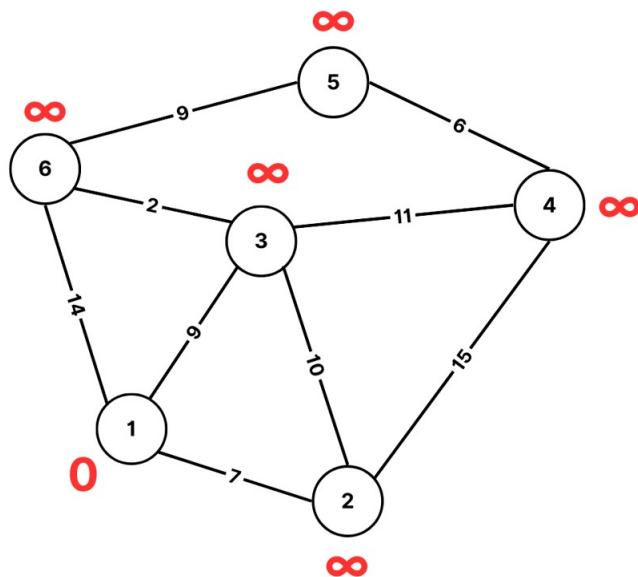


Алгоритм Дейкстри

Алгоритм Дейкстри використовується для знаходження найкоротшої відстані від певної вершини зв'язного графа до всіх інших.

Розглянемо виконання алгоритму Дейкстри на прикладі графа, показаного на малюнку.

У колах розміщено номери вершин, над ребрами вказано їх вагу, тобто довжину шляху.



Алгоритм починається з ініціалізації графа.

Біляожної вершини є червона мітка, що вказує довжину найкоротшого шляху до цієї вершини від вершини 1, яку ми довільно обрали як відправну точку для алгоритму. Мітка самої вершини 1 встановлюється рівною 0, мітки решти вершин вважаються рівними нескінченності (∞). Це зображає той факт, що відстані від вершини 1 до інших вершин поки невідомі, і це матиме певне практичне призначення при виконанні алгоритму.

Перший крок алгоритму

Визначаємо всі вершини, які з'єднані спільним ребром з нашою початковою вершиною 1, у нашему випадку це вершини 2, 3, 6.

Другий крок

Знайдіть найкоротші шляхи, які не проходять через інші вершини до цих сусідніх вершин (2, 3, 6) від вершини 1.

Найкоротший шлях визначається певним чином, а саме, вага гілки, що веде до сусідньої вершини, додається до мітки поточної вершини (тобто до 0), і ця сума записується як мітка цієї сусідньої вершини, якщо він менший за поточну мітку сусідньої вершини.

Оскільки шлях від вершини 1 до вершини 2 буде дорівнювати $0 + 7$, що менше поточної мітки вершини 2, тобто нескінченності, то $0 + 7$, або 7, записується як мітка вершини 2.

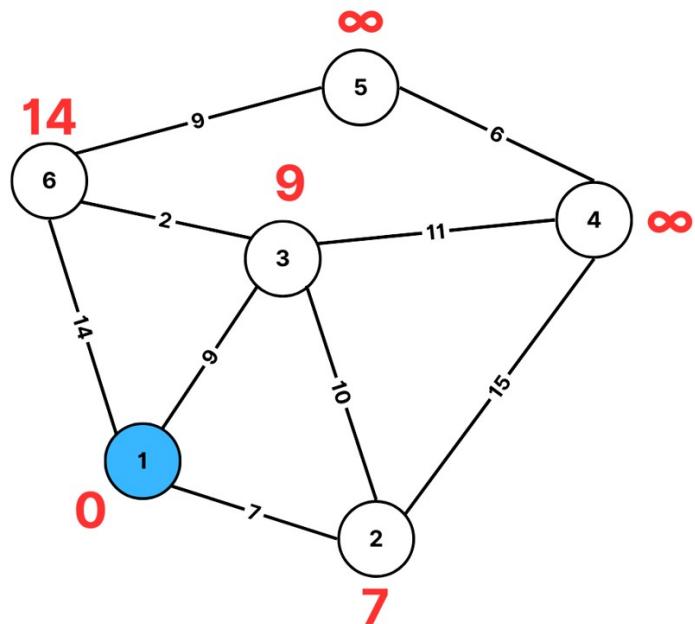
Для вершини 2 мітка буде $0 + 7$.

Для вершини 3 мітка буде $0 + 9$.

Для вершини 6 мітка буде $0 + 14$.

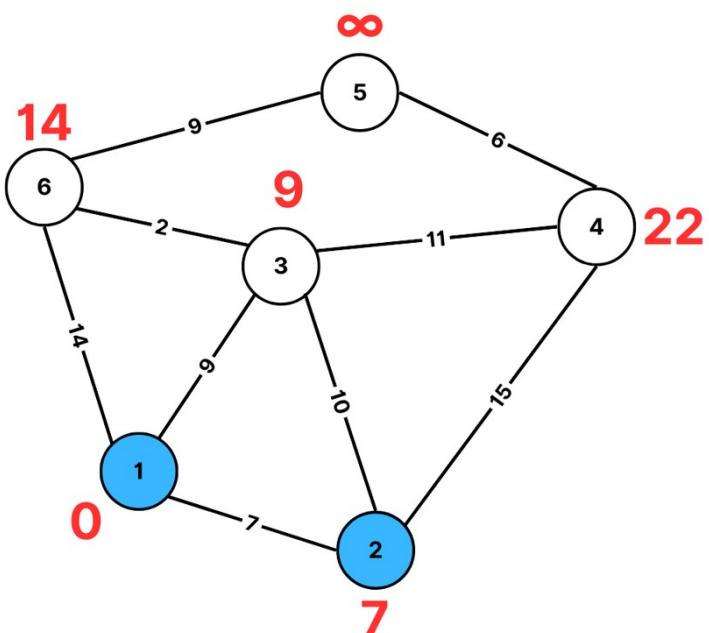
Третій крок

Позначаємо вершину 1 як оброблену і переходимо до сусідньої вершини, до якої найкоротша відстань, у нашому випадку суміжні вершини 2, 3, 6, а вершина 2 розташована на найкоротшій відстані від вершини 1. Тобто переходимо до вершини номер 2.



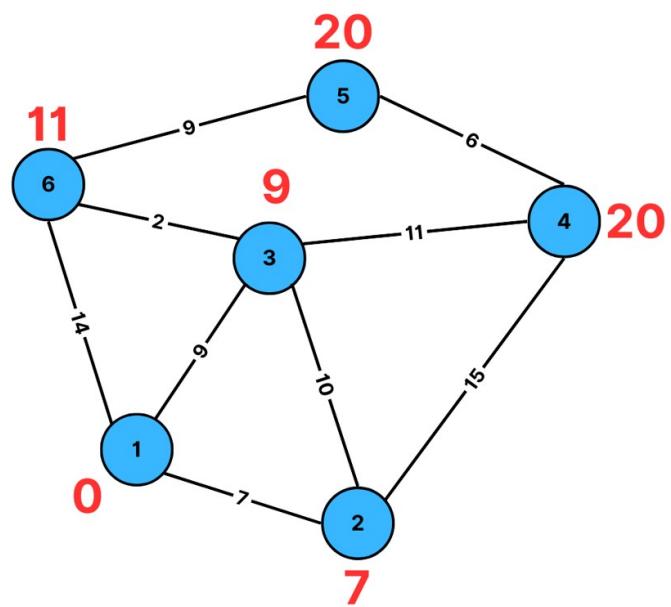
Четвертий крок

Повторюємо два кроки для вершини 1, 2, 3.



П'ятий крок

Виконайте крок 4 для всіх гілок, поки вони не будуть позначені як оброблені.



Кінець алгоритму

В результаті над кожною вершиною буде позначений мінімальний шлях до неї від початкової вершини 1.

Жадібний алгоритм — простий і прямолінійний евристичний алгоритм, який приймає найкраще рішення, виходячи з наявних на кожному етапі даних, не зважаючи на можливі наслідки, сподіваючись урешті-решт отримати оптимальний розв'язок.

Наприклад, використання жадібної стратегії для задачі комівояжера породжує такий алгоритм: "На кожному етапі вибирати найближче з невідвіданих міст".

Жадібний алгоритм добре розв'язує деякі задачі, а інші — ні. Більшість задач, для яких він спрацьовує добре, мають дві властивості: по-перше, до них можливо застосувати принцип жадібного вибору, по-друге, вони мають властивість оптимальної підструктури. Необхідно, щоб жадібний вибір на першому етапі не унеможливлював шлях до оптимального розв'язку.

Кажуть, що принцип жадібного вибору застосовується до проблеми оптимізації, якщо послідовність локально оптимальних варіантів дає глобально оптимальне рішення. Як правило, доказ оптимальності відбувається за цією схемою:

Доведено, що жадібний вибір на першому кроці не закриває шляху до оптимального рішення: для кожного рішення існує інше, узгоджене з жадібним вибором і не гірше первого.

Показано, що підзадача, яка виникає після жадібного вибору на першому кроці, схожа на вихідну. Аргумент завершується індукцією.

Задача комівояжера полягає в тому, щоб знайти найбільш вигідний маршрут, який хоча б один раз проходить через вказані міста, а потім повертається в початкове місто. В умовах задачі вказується критерій рентабельності того чи іншого маршруту (найкоротший шлях, найдешевший шлях тощо). Проблему комівояжера, в деяких випадках, можна вирішити з допомогою жадібного алгоритму.

До жадібних алгоритмів належать алгоритми, в яких на кожному етапі виконується оптимальне рішення даного етапу. Сума оптимальних розв'язків усіх кроків алгоритму обов'язково дає глобальне оптимальне рішення, якщо рішення попереднього етапу не блокує оптимальне рішення наступного етапу. Наприклад, від міста А до міста В є шляхи 4 км і 3 км, а від міста В до міста С є шляхи 5 км і 4 км, але ми знаємо, що шлях від А до С становить 2 км. Якщо нам потрібно їхати поєднано, тобто по порядку, до міст А, В, С, найкоротшим шляхом, то на кожному етапі достатньо вибирати найкоротший шлях з міста до іншого міста. У нашому випадку це А, 3 км, В, 4 км, С. Загальний шлях 7 км.

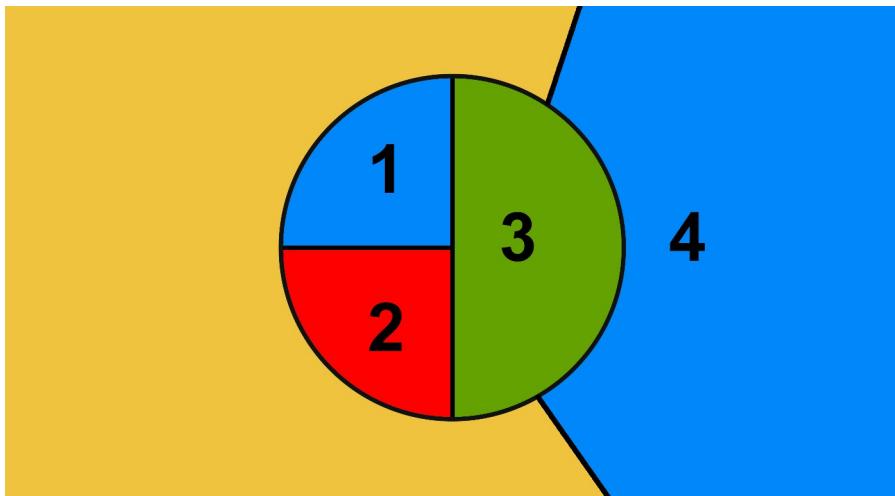
Причому на кожному етапі ми робимо найкращий вибір, тобто вибираємо найменшу відстань. Але якщо нам потрібно прокласти інтернет-кабель між містами А, В, С, щоб витратити найменшу кількість кабелю, то нам потрібно прокласти кабель від міста А до міста С і від С до міста В. Таким чином, А, 2 км, С, 4 км, В. Загалом витратимо 6 км кабелю.

Теорема про чотири кольори

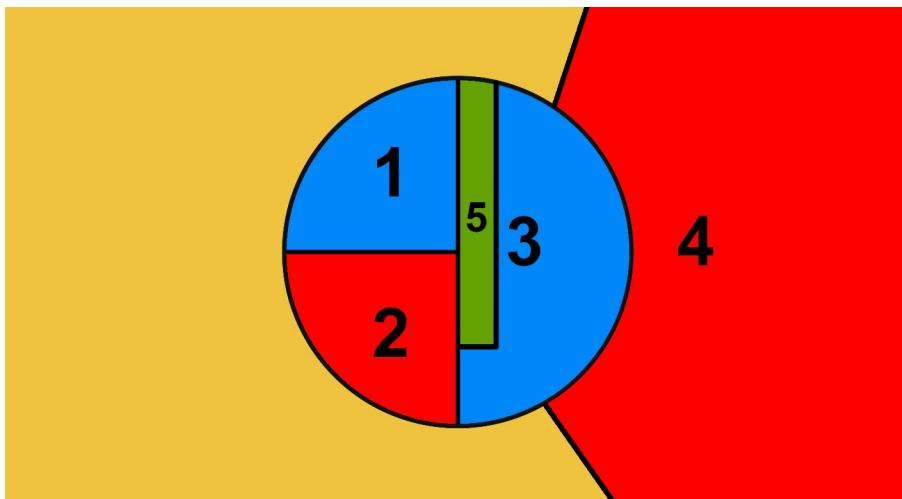
Теорема про чотири кольори — це теорема, яка стверджує, що будь-яку мапу (карту), розташовану на площині або на сфері, можна розфарбувати не більше ніж чотирма різними кольорами (фарбами), так що будь-які дві області зі спільною ділянкою кордону будуть забарвлені в різні кольори. У цьому випадку ділянки повинні бути просто з'єднані (без анклавів, тобто без включен, обривів, іншими словами, щоб будь-який замкнутий контур на мапі можна було стягнути в точку), а загальна межа (кордон) означає частину лінії, тобто стики кількох ділянок в одній точці не вважаються для них спільною межею. Іншими словами, покажіть, що хроматичне число плоского графа не перевищує 4. Хроматичне число графа — це мінімальна кількість кольорів, у яку можна розфарбувати вершини графа G так, щоб кінці будь-якого ребра мали різні кольори. Позначається як $\chi(G)$.

Наскільки відомо, вперше ця гіпотеза була висунута у 1852 році, коли Френсіс Гатрі, намагаючись розфарбувати мапу графств Англії, помітив, що потрібні лише чотири різні кольори. У той час брат Гатрі, Фредерік, був учнем Августа Де Моргана (колишнього радника Френсіса) в Університетському коледжі Лондона. Френсіс поцікавився у Фредеріка про це, який потім відніс його до Де Моргана (Френсіс Гатрі закінчив навчання в 1852 році, а пізніше став професором математики в Південній Африці). У 1890 році Персі Хівуд довів теорему про п'ять кольорів і узагальнив гіпотезу про чотири кольори на поверхні довільного роду. Зокрема, про це повідомляє Ріхард Курант в книзі “Що таке математика?” (1941). Теорема про чотири кольори була доведена в 1976 році Кеннетом Аппелем і Вольфгангом Хакеном після багатьох хибних доказів і контрприкладів (на відміну від теореми про п'ять кольорів, доведеної в 1800-х роках, яка стверджує, що п'яти кольорів достатньо для фарбування карти). Щоб розвіяти будь-які сумніви щодо доказу Аппеля –Хакена, у 1997 році Робертсон, Сандерс, Сеймур і Томас опублікували простіший доказ, який використовує ті самі ідеї та все ще спирається на комп’ютери. Крім того, у 2005 році теорема була доведена Жоржем Гонтьє з допомогою програмного забезпечення для доведення теорем загального призначення. Це була перша велика теорема, доведена з допомогою комп’ютера.

Теорема. Щоб розфарбувати будь-яку мапу (тобто систему кордонів), розташовану на площині або на кулі, потрібно щонайменше 4 кольори, щоб будь-які дві області зі спільним кордоном (спільною лінією границі) були пофарбовані в різні кольори. (Залишилося довести, що потрібно не більше 4). Доведення (методом надання прикладу чи контрприкладу до зворотної теореми). Цю картку не можна розфарбувати менш ніж чотирма кольорами.



Ця карта потребує щонайменше чотири кольори, щоб розфарбувати її без поєднання областей з однаковими кольорами.

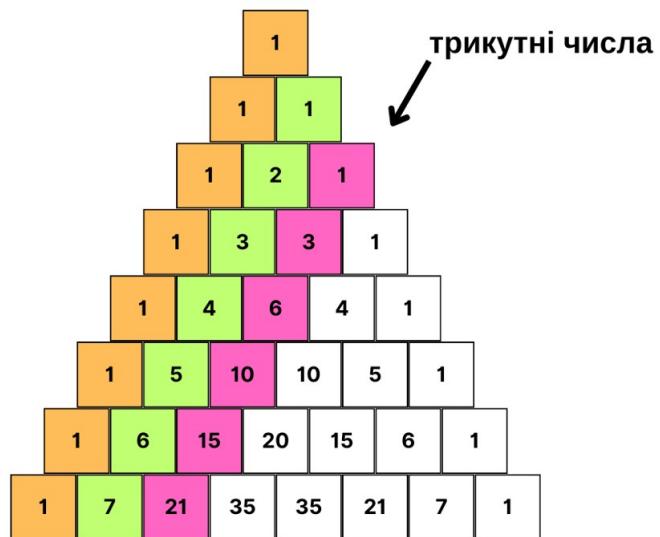


Теорема про чотири кольори (фарби). Будь-яку карту без анклавів можна розфарбувати в 4 кольори, щоб жодні сусідні області не мали одинакового кольору. Анклав — територія або частина однієї держави, оточена з усіх сторін територією іншої держави.

Які найпопулярніші алгоритми в комбінаториці?

Трикутник Паскаля

У математиці трикутник Паскаля — це трикутний масив біноміальних коефіцієнтів. Він названий на честь французького математика Блеза Паскаля, хоча інші математики вивчали його за 3 століття до нього в Китаї. Рядки трикутника Паскаля умовно перераховуються, починаючи з рядка $n = 0$ у верхній частині (0-й рядок). Записи в кожному рядку нумеруються зліва, починаючи з $k = 0$, і зазвичай розташовані в шаховому порядку відносно чисел у сусідніх рядках. Трикутник можна побудувати в такий спосіб: У рядку 0 (самий верхній рядок) є унікальний ненульовий запис 1. Кожен запис кожного наступного рядка створюється шляхом додавання числа вгорі та ліворуч із числом угорі та праворуч, обробляючи пусті записи як 0. Наприклад, початкове число в першому (або будь-якому іншому) рядку дорівнює 1 (сума 0 і 1), тоді як числа 1 і 3 в третьому рядку додаються, щоб отримати число 4 в четвертому рядку.



Трикутник Ян Хуея, по суті трикутник Паскаля, в якому цифри зображені з допомогою паличок, з'являється в математичній роботі Чжу Шицзе, датованій 1303 роком. Трикутник Паскаля був описаний Паскалем у праці “Трактат про арифметичний трикутник”, опублікованій в 1653 році. Цей трактат був дуже важливим, оскільки представляв одну з перших робіт з теорії ймовірності, але самі біноміальні коефіцієнти, які представляють трикутник Паскаля, були вже відомі в Європі. Близько 1150 року індійський математик Бхаскарачарья (Бхаскара, означає “вчитель”), також відомий як Бхаскара II, у книзі “Лілаваті”, частина 6, розділ 4, дав чіткий опис біноміальних коефіцієнтів.

У трикутнику Паскаля на перетині k -ї вертикальні та n -ї горизонтальні розміщене число $C(n,k)$.

$$C(n,k) = n! / ((n-k)! * k!);$$

$$C(n,0) = 1 \text{ (для всіх } n\text{);}$$

Лейбніц присвятив свою дисертацію комбінаториці та пізніше опублікував її під назвою “Про комбінаторне мистецтво” (написано в 1666 р.).

Комбінаторика — розділ математики, що вивчає дискретні об'єкти, множини (комбінації, перестановки, розміщення та перерахування елементів) і відношення на них (наприклад, частковий порядок). Термін “комбінаторика” ввів у математичний вжиток Лейбніц, але Паскаль почав розвивати комбінаторику ще до Лейбніца.

Кількість комбінацій без повторень

П'єр Ферма вже зновував формулу: кількість комбінацій без повторень від n до k дорівнює біноміальному коефіцієнту $C(n, k) = n!/(k!(n - k)!)$.

$\bar{A}(n,k)$ - означає кількість комбінацій, які допускають дублювання, довжиною k із набору з n елементів.

$\bar{A}(n, k) = \text{pow}(n, k)$.

$\bar{A}(3,2) = 3^2$, оскільки якщо у нас є три елементи x,y,z , ми можемо побудувати 3^2 комбінацій.

1. xx.
2. xy.
3. xz.
4. ux.
5. yy.
6. yz.
7. zx.
8. zy.
9. zz.

$A(n,k)$ - означає кількість комбінацій, без дублювання, довжиною k із набору з n елементів.

$A(n,k) = n!/(n-k)!$.

$A(2,2) = 2!/(2-2)! = 2/1 = 2$, оскільки якщо у нас є два елементи x,y , ми можемо побудувати дві комбінації без повторення.

1. xy.
2. yx.

$\bar{C}(n,k)$ - означає кількість спільногорозміщення, яке допускає дублювання та ігнорує порядок, довжиною k із набору з n елементів.

$\bar{C}(n,k) = (k+n-1)!/k!*(n-1)!$.

$\bar{C}(n,k) = (2+2-1)!/2!*(2-1)! = 6/2 = 3$, тому що ми можемо поєднати два елементи x,y з представленням і без упорядкування, тобто $xy = yx$, лише трьома способами.

1. xx.

2. uy.

3. xy або ux.

$C(n,k)$ - означає кількість спільного розміщення, яке забороняє дублікати та ігнорує порядок, довжиною k із набору з n елементів.

$$C(n,k) = n! / ((n-k)! * k!).$$

$$C(n,0) = 1 \text{ (константа).}$$

$C(2,2) = 2! / (2-2)! * 2! = 2/1 * 2 = 2/2 = 1$, оскільки ми можемо поєднати два елементи x, y без повторення та без упорядкування, тобто $xy = yx$, лише одним способом.

1. xy або yx.

Приклад комбінаторної задачі. Двоє дівчат зібрали 10 ромашок, 15 волошок і 14 тюльпанів. Скількома способами вони можуть поділитися цими квітами? Зрозуміло, що ромашки можна розділити одинадцятьма способами — перша може не брати жодної ромашки, взяти 1 ромашку, 2 ромашки, ..., всі 10 ромашок. Так само волошки можна розділити шістнадцятьма способами, а тюльпани — п'ятнадцятьма. Оскільки квіти кожного виду можна розділити незалежно від квіток іншого типу, то за правилом добутку отримаємо $11 * 16 * 15 = 2640$ способів поділу квітів. Звичайно, серед цих методів є й вкрай несправедливі, при яких, наприклад, хтось із дівчат взагалі не отримує квіти. Тому введемо обмеження й на те, що кожен з дітей повинен отримати не менше 3 квіток кожного виду. Тоді ромашки можна розділити лише п'ятьма способами: перша дівчина може взяти 3, 4, 5, 6 або 7 квіток. Так само волошки можна розділити на 10 способів, а тюльпани — на 9. У цьому випадку загальна кількість методів ділення становить $5 * 10 * 9 = 450$. Загалом, якщо є n_1 предметів одного виду, n_2 предметів іншого виду, ..., n_k предметів k -го виду, то їх можна розділити між двома людьми на $(n_1 + 1) * (n_2 + 1) * ... * (n_k + 1)$ способи. Зокрема, якщо всі предмети відрізняються один від одного і їх кількість дорівнює k , то $n_1 = n_2 = ... = n_k = 1$, а отже, існує 2^k способів поділу. Це випливає з того, що кожен предмет може бути або у першій, або у другій людини, що дає 2 варіанти для кожного предмета. Усього $2 * 2 * ... * 2$ (k рази) = 2^k .

Біном Ньютона

Ньютон описав формулу під назвою “біном Ньютона”, яка описує закономірність у трикутнику Паскаля і використовується в теорії ймовірностей та комбінаториці. Ісаак Ньютон відкрив близько 1665 р., а пізніше в 1676 р. без доказів сформулював загальну форму теореми (для будь-якого дійсного числа n), а доказ Джона Колсона (1680–1760) було опубліковано в 1736 р. Гаусс дав першу задовільний доказ збіжності таких рядів у 1812 році. Пізніше Абелль дав трактування, яке буде працювати для загальних комплексних чисел.

Біноміальна теорема

$$(a + x)^n = (a+x)(a+x)\dots(a+x); n \text{ разів.}$$

Розгорнемо дужки в правій частині цієї рівності й запишемо всі множники в тому порядку, в якому ми їх зустрічаємо. Наприклад, $(a+x)^2$ і $(a+x)^3$ запишемо у вигляді:

$$(a+x)^2 = (a+x)(a+x) = a^2 + 2ax + x^2 = aa + ax + xa + xx,$$

$$(a+x)^3 = (a+x)(a+x)(a+x) = a^3 + 3a^2x + 3ax^2 + x^3 = aaa + aax + axa + axx + xaa + xax + xxa + xxx;$$

Зараз ми даємо подібні умови. Члени, що містять однакову кількість літер x і a , будуть подібними. Знайдемо, скільки буде членів, до складу яких входять k букв x і, отже, $n - k$ букв a .

Ці члени є перестановками з повтореннями, тому їх кількість:

$$C(n,k) = n! / ((n-k)! * k!);$$

З цього випливає, що після скорочення таких виразів вираз $x^k a^{n-k}$ увійде з коефіцієнтом $C(n,k)$.

Отже, ми довели рівність, яку називають біномом Ньютона, а саме:

$$(a + x)^n = C(n,0)a^n + C(1,n)a^{n-1}x + \dots + C(k, n)a^{n-k}x^k + \dots + C(n,n)x^n.$$

Факторіал:

$$1) 0! = 1;$$

$$2) 1! = 1;$$

$$3) (n)! = n!;$$

$$4) n! = ((n-1)!) * n;$$

Числа Каталана

Числа Каталана виникли в теорії комбінаторики та теорії чисел. Вони використовуються для розв'язання різних комбінаторних задач, таких як кількість шляхів у сітці чи кількість способів розкладання дужок.

Ці числа отримали свою назву на честь бельгійського математика Еугена Шарля Каталана, який вивчав їх в середині 19 століття. Вони виникають у різних галузях математики та мають різноманітні застосування, зокрема в комбінаториці та теорії графів.

Один із прикладів застосування чисел Каталана — це кількість можливих способів розміщення дужок у виразі. Наприклад, якщо у нас є вираз із 3 пар дужок, то кількість різних шляхів, якими можна розмістити ці дужки, визначається числом Каталана.

Для випадку з трьома парами дужок, можливі конфігурації можуть виглядати так:

1. ((()),
2. ()(),
3. ()(),
4. ()()0,
5. (00)

Кількість розміщення дужок відповідає числу Каталана, а саме третьому числу ($\text{Catalan}(3) = 5$).

Уявіть ситуацію, де ви маєте організувати захід чи обід із кількома парами людей. Щоб створити зручну атмосферу та сприяти спілкуванню, ви хочете розташувати цих людей у коло або за столом так, щоб жодні двоє з одного партнерства не сиділи поруч одне з одним.

Числа Каталана виникають тут, оскільки вони визначають кількість можливих способів розташування цих пар у колі або за столом без об'єднання представників одного партнерства.

Наприклад, якщо у вас є 3 пари, то існує 5 різних способів розташування їх так, щоб жодні двоє з одного партнерства не сиділи поруч. Це дозволяє створити комфортну та приемну атмосферу для учасників події.

Ось перші кілька чисел Каталана:

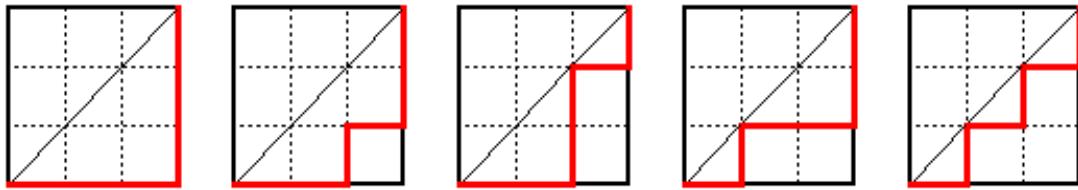
$$\begin{aligned} C(0) &= 1 \\ C(1) &= 1 \\ C(2) &= 2 \\ C(3) &= 5 \\ C(4) &= 14 \\ C(5) &= 42 \\ C(6) &= 132 \\ C(7) &= 429 \\ C(8) &= 1430 \\ C(9) &= 4862 \end{aligned}$$

Далі можна обчислити числа Каталана за допомогою рекурсивної формулі або за допомогою інших методів, таких як динамічне програмування. Ці числа виникають в різних комбінаторних задачах і мають широкий спектр застосувань в математиці та інших галузях.

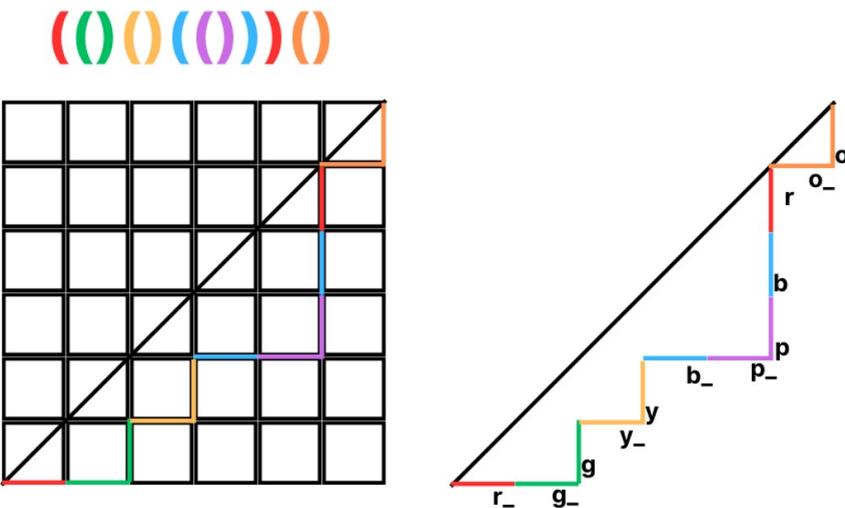
Ось формула для чисел Каталана:

$$C_n = 1 / (n + 1) * ((2n)! / (n! * n!));$$

Монотонні шляхи у квадраті – маршрути з лівого нижнього кута квадрата у правий верхній, які йдуть лініями сітки вгору або вправо і не заходять вище діагоналі. На малюнку такі шляхи для квадрата 3x3, кількість $Catalan(3) = 5$.



Дуже легко побудувати відповідність між послідовностями дужок та монотонними шляхами у квадраті.



Читаючи послідовність дужок зліва направо, будемо будувати шлях, почавши з лівого нижнього кута, — для кожної дужки намалюємо горизонтальний відрізок, для дужки, що закривається, — вертикальний.

Через те, що в послідовності було рівне число дужок, що відкриваються і закриваються, то шлях в результаті закінчиться в правому верхньому кутку, а той факт, що кожна дужка, що відкривається, стоять раніше відповідної її дужки (адже послідовність - правильна) гарантує нам, що шлях не перейде в верхню половину квадрата.

Числа Стірлінга

Числа Стірлінга — це клас комбінаторних чисел, які виникли у комбінаториці. Їх назва пов'язана з ім'ям шотландського математика Джеймса Стірлінга, який вивчав ці числа у 18 столітті.

Числа Стірлінга другого роду, позначені як $S(n,k)$, представляють собою кількість можливих способів розбити множину n елементів на k непорожніх підмножин. Іншими словами, ці числа вказують, на скільки способів можна розподілити n різних елементів у k підмножин, що неперетинаються.

Одна з основних ідей у формулюванні цих чисел полягає в тому, що вони дозволяють визначити кількість можливих способів поділу множини таким чином, що жодне з підмножин не містить порожніх елементів, але може містити більше одного елемента.

Наприклад, якщо ми маємо множину з трьох елементів $\{1,2,3\}$, то числа Стірлінга другого роду для $n=3$ та $k=2$ визначають кількість можливих способів розбити цю множину на дві непорожні підмножини. Зокрема, якщо обрано число $S(3,2)=3$, то можливі комбінації будуть:

1. $\{1,2\}, \{3\}$,
2. $\{1,3\}, \{2\}$,
3. $\{2,3\}, \{1\}$

Отже, існує три різних способи розбити множину з трьох елементів на дві непорожні підмножини.

Числа Фібоначчі

Числа Фібоначчі — це послідовність чисел, де кожне число є сумаю двох попередніх. Таким чином, послідовність починається з двох перших чисел, які зазвичай дорівнюють 0 і 1. Формально, числа Фібоначчі можна визначити наступним чином:

$$\begin{aligned} F(0) &= 0, \\ F(1) &= 1, \\ F(n) &= F(n-1) + F(n-2) \text{ для } n > 1. \end{aligned}$$

Отже, третє число ($F(2)$) буде дорівнювати сумі першого ($F(1) = 1$) та другого ($F(0) = 0$), тобто 1. Четверте число ($F(3)$) буде дорівнювати сумі другого ($F(1) = 1$) та третього ($F(2) = 1$), тобто 2, і так далі.

Початкова частина послідовності чисел Фібоначчі виглядає наступним чином: 0, 1, 1, 2, 3, 5, 8, 13, 21, і так далі.

Ця послідовність зустрічається в різних галузях математики, науки та природних явищах. Вона використовується в програмуванні, алгоритмах та інших областях через свої унікальні властивості та рекурсивний характер.

Ідеальні числа (досконалі числа)

Ідеальні числа — це натуральні числа, які дорівнюють сумі своїх натуральних дільників, за винятком самого себе.

Приклади ідеальних чисел:

6 - дільники: 1, 2, 3; $(1+2+3=6)$.

28 - дільники: 1, 2, 4, 7, 14; $(1+2+4+7+14=28)$.

496 та 8128 також є ідеальними числами.

Ідеальні числа вивчаються в теорії чисел та мають цікаві властивості.

Які найпопулярніші алгоритми в теорії ймовірностей?

На основі картин 16 століття ми знаємо, що в той час вже були гральні карти та гральні кубики. Італійський математик Джироламо Кардано (1501 — 1576) у своїй автобіографії писав, що любить грати в азартні ігри для розваги, але потрібно це робити розумно, в компанії порядних людей. Важливо зауважити, що є цілком не контролювані ігри, а є ігри де не все залежить від випадку, наприклад, ігри покер та монополія в великий мірі покладаються на вміння гравця, хоча в них є елемент випадковості.

Теорема Баєса

Умовна (пов'язана) ймовірність.

Ймовірність події A, що монета зі сторонами S1 та S2 під час першого підкидання впаде стороною S1 і під час другого підкидання тією ж стороною S1 буде $\frac{1}{4}$, бо всього можливо 4 комбінації на основі двох підкидань, а саме: (S1, S1), (S1, S2), (S2, S1), (S2, S2). Таким чином, ймовірність події A, що позначається як $P(A)$ буде $\frac{1}{4}$.

Якщо подія B означає, що випала сторона S1, тоді, у разі настання події B, ймовірність повії A буде вже не $\frac{1}{4}$, а $\frac{1}{2}$, бо наступний кидок може дати тільки дві комбінації (S1, S1), або (S1, S2).

$P(A|B)$ - ймовірність настання події A, якщо настала подія B.

$P(A|B) = |A \cap B| : |B|$, якщо $A \cap B \neq \emptyset$.

$P(A|B) = P(A)$, якщо $A \cap B = \emptyset$.

Відповідно до теореми Баєса, ймовірність події $P(A|B)$ можна визначити як $P(A|B) = \frac{1}{2} = P(B|A) * P(A) / P(B) = 1 * (\frac{1}{4}) / (\frac{1}{2}) = \frac{1}{2}$;

Теорема Баєса: $P(A|B) = P(B|A) * P(A) / P(B)$;

Приклад використання теореми Баєса

Нехай є захворювання X, яким може хворіти 1% населення. Також у нас є тест, який правильно виявляє захворювання у 90% випадків, але також може давати хибно позитивні результати у 5% здорових людей.

Тепер, якщо ми хочемо знати ймовірність того, що людина справді хвора, якщо тест показав позитивний результат, ми можемо скористатися теоремою Баєса:

Позначимо події:

A: людина хвора на захворювання X;
B: тест показав позитивний результат;

Позитивний результат тесту в цьому випадку означає, що людина хвора.

Знаходимо априорну ймовірність:

$P(A) = 0.01$ (ймовірність того, що випадково обрана людина хвора);

Знаходимо ймовірність тесту, якщо людина справді хвора:

$P(B|A) = 0.90$ (правильний результат тесту для хворих);

Знаходимо ймовірність позитивного тесту, якщо людина здорована:

$P(B|\neg A) = 0.05$ (позитивний результат тесту для здорових);

Застосовуємо формулу Баєса:

$$P(A|B) = (P(B|A) * P(A)) / ((P(B|A) * P(A)) + (P(B|\neg A) * P(\neg A)));$$

$$P(A|B) = (0.90 * 0.01) / ((0.90 * 0.01) + (0.05 * 0.99));$$

Частина $(0.90 * 0.01)$ вказує на ймовірність того, що людина справді хвора і тест покаже позитивний результат.

Частина $(0.05 * 0.99)$ вказує на ймовірність отримати позитивний результат тесту, якщо людина здорована.

Тобто $(0.90 * 0.01) + (0.05 * 0.99)$ вказує на загальну ймовірність отримати позитивний результат тесту.

Ймовірність того, що піддослідний хворий:

$$P(A|B) \approx 0.153;$$

Таким чином, навіть якщо тест показав позитивний результат, існує тільки близько 15.3% шансів того, що людина справді хвора. Без проведення тесту, шанс (ймовірність), що людина дійсно хвора 1%. Позитивний тест, в нашому випадку, підвищує шанс більше ніж в 15 разів.

Метод Монте-Карло

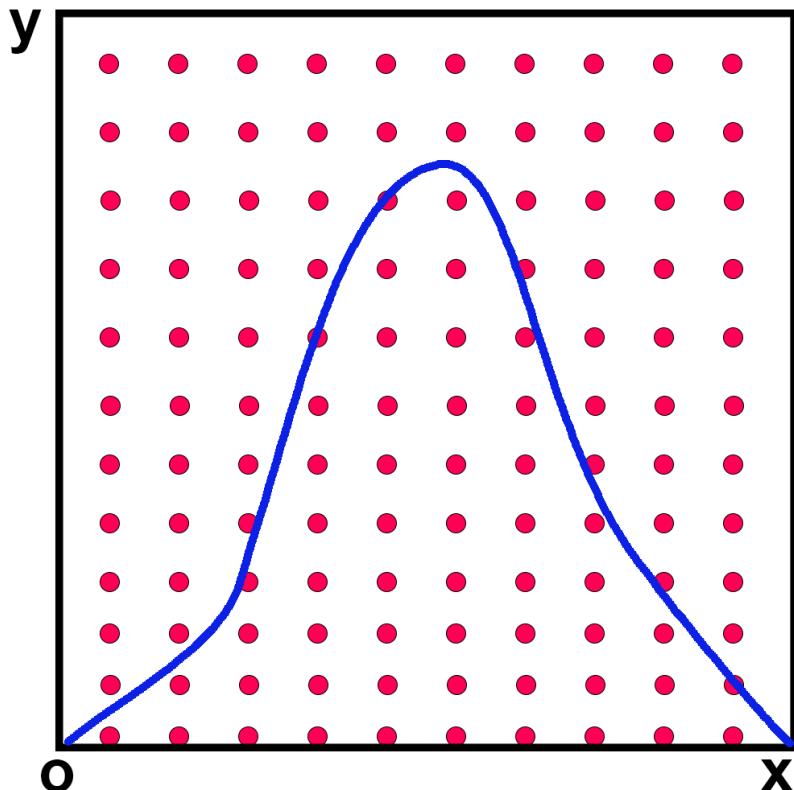
Метод Монте-Карло — це статистичний обчислювальний метод, який використовує випадкові числа для наближеного вирішення різних математичних, фізичних або обчислювальних проблем. Цей метод названий на честь казино Монте-Карло у Монако, де гравці використовують випадковий вибір чисел для азартних ігор. Метод Монте-Карло використовується для розв'язання проблем, які можуть бути важко або навіть неможливо вирішити аналітично.

Основна ідея методу Монте-Карло полягає в тому, щоб генерувати велику кількість випадкових чисел і використовувати їх для апроксимації результатів. Наприклад, метод Монте-Карло може бути використаний для обчислення наближеного значення числа Пі (π), обчислення інтегралів, моделювання фізичних процесів, аналізу ризиків у фінансах, та багато інших завдань.

Основний принцип методу полягає в тому, щоб використовувати велику кількість випадкових величин для апроксимації математичних функцій або розподілів і використовувати статистичні методи для обчислення потрібних значень.

Метод Монте-Карло дуже потужний і може бути застосований до широкого спектру завдань, але він також вимагає великої кількості обчислень і може бути витратним з точки зору обчислювальних ресурсів.

Геометричний алгоритм Монте-Карло для інтеграції



Для визначення площин під графіком функції можна використовувати наступний стохастичний алгоритм:

1. Обмежимо функцію прямокутником, площеу А якого можна легко обчислити. Таким чином, А буде максимальним значенням інтеграла цієї функції.
2. Рівномірно (рівноточково) "закинемо" певну кількість N точок у цей прямокутник.
3. Визначте кількість K точок, що потрапляють під графік функції.
4. Площа області S (тобто інтеграла), обмеженої функцією та осями координат, задається виразом: $S = A * (K / N)$. Чим більше точок N, тим точнішим буде обчислення інтеграла.

Метод Монте-Карло заснований на теорії ймовірності.

Парадокс Монті Холла

Парадокс Монті Холла — одна з відомих проблем теорії ймовірностей, вирішення якої, на перший погляд, суперечить здоровому глузду. Це завдання не є парадоксом у прямому сенсі слова, оскільки воно не містить протиріччя, його називають парадоксом, оскільки його вирішення може здатися несподіваним.

Найпоширеніше формулювання цього завдання виглядає наступним чином:

Уявіть, що ви стали учасником гри, в якій вам належить вибрати одну з трьох дверей. За одними дверима є приз, а за двома іншими дверима нічого. Ви обираєте одну з дверей, наприклад, номер 1, після цього модератор, який знає, де знаходиться приз, а де нічого, відкриває одну з дверей, що залишилися, наприклад, номер 3, за якою нічого немає. Потім він запитує вас: "Чи хотіли б ви змінити свій вибір і вибрати двері номер 2?" Чи зростуть ваші шанси виграти приз, якщо ви приймете пропозицію модератора та зміните свій вибір?

Кілька уточнень:

1. приз однаково ймовірно знаходиться за будь-якими з трьох дверей;
2. модератор знає, де знаходиться приз;
3. модератор у будь-якому випадку повинен відкрити двері без призу (але не обрані гравцем) і запропонувати гравцеві змінити свій вибір;

Для виграшної стратегії важливо: якщо ви змінюєте вибір дверей після дій модератора, то виграєте, якщо спочатку вибрали програшні двері. Це станеться з ймовірністю $\frac{2}{3}$, оскільки спочатку ви можете вибрати програшні двері 2 з 3 способів.

Але часто при розв'язанні цієї задачі гравці міркують приблизно так: ведучий в кінці завжди прибирає одні програшні двері, і тоді ймовірність виграну за двома невідкритими дверима стає рівною $\frac{1}{2}$, незалежно від початкового вибору. Але це неправда: хоча дійсно є дві можливості вибору, ці можливості не є однаково вірогідними. Це так, оскільки спочатку всі двері мали рівні шанси на перемогу, але потім мали різні шанси бути виключеними.

Для більшості людей такий висновок суперечить інтуїтивному сприйняттю ситуації, а через виникачу невідповідність між логічним висновком і відповідю, до якої схиляє інтуїтивна думка, завдання називають парадоксом Монті Холла.

Ймовірність виграти у цій грі при зміні рішення буде $\frac{2}{3}$, бо гравець програє тільки у випадку, коли його перший вибір був правильний, тобто $\frac{1}{3}$.

Парадокс дня народження

Парадокс дня народження — це твердження, що якщо дана група з 23 і більше людей, то ймовірність того, що хоча б у двох з них одинаковий день народження (день і місяць), перевищує 50%.

У групі з 23 і більше осіб ймовірність збігу днів народження (число і місяць) принаймні для двох осіб перевищує 50%. Наприклад, якщо в класі 23 або більше учнів, імовірніше, що кілька однокласників будуть мати дні народження в один день, ніж кожен матиме свій унікальний день народження.

Для 57 і більше людей ймовірність такого збігу перевищує 99%, хоча досягає 100%, за принципом Діріхле, лише тоді, коли в групі не менше 367 осіб (рівно на 1 більше, ніж кількість днів у високосному році).

Це твердження може здатися незрозумілим, оскільки ймовірність збігу днів народження двох людей з будь-яким днем у році ($1/365 = 0,27\%$), помножена на кількість людей у групі (23), дає лише $(1/365) * 23 = 6,3\%$. Це міркування невірне, оскільки кількість можливих пар $(23 * 22 / 2 = 253)$ значно перевищує кількість людей у групі ($253 > 23$). Таким чином, твердження не є парадоксом у строго науковому розумінні: в ньому немає логічного протиріччя, а парадокс полягає лише в відмінностях між інтуїтивним сприйняттям ситуації людиною та результатами математичних обчислень.

Потрібно визначити ймовірність того, що в групі з n людей хоча б двоє з них мають однакові дні народження.

Нехай дні народження розподілені рівномірно, тобто припустимо, що:

у році 365 днів (високосних років немає);

у групі немає людей, які явно народилися в один день (наприклад, близнюки);

народжуваність не залежить від дня тижня, сезону чи інших факторів.

Давайте випадково візьмемо одну людину з групи та згадаємо її день народження. Тоді беремо другу людину навмання, і ймовірність того, що його день народження не збігається з днем народження першої особи, дорівнює $1 - 1/365$. Потім візьміть третю особу; тоді як ймовірність того, що його день народження не збігається з днями народження перших двох, становить $1 - 2/365$. Сперечаючись за аналогією, дійдемо до останньої людини, для якої ймовірність дня народження не збігається з усіма попередніми буде дорівнює $1 - (n - 1) / 365$.

Припустимо у нас в році 6 днів. Ймовірність народження людини однакова для будь-якого з шести днів. Тоді ймовірність того, що кожен з шести людей народиться єдиним у свій день, тобто їхні дні народження будуть різні, можна визначити по формулі:

$$6/6 * 5/6 * 4/6 * 3/6 = (6*5*4*3)/6^6 = 0,2777..;$$

Таким чином, ймовірність, що хоча б у двох з них буде день народження в один день, більше 0,7 з 1, тобто більше ніж 70 відсотків.

Петербурзький парадокс

Санкт-Петербурзький парадокс — математична задача, що ілюструє розбіжність математичного очікування виграшу з його реальною оцінкою гравцями.

Санкт-Петербурзький парадокс полягає в тому, що очікуваний грошовий виграш в грі нескінчений, проте більшість людей ухиляється від участі в ній. Чому так відбувається?

Санкт-Петербурзький парадокс був вперше опублікований Даніелем Бернуллі у 1738 році в "Коментарях Санкт-Петербурзької Академії".

Формулювання парадоксу:

Нехай казино проводить таку гру: вступаючи в гру, гравець платить деяку суму, а потім підкидає монету (імовірність кожного результату — 50 %), поки не випаде орел. При випаданні орла гра закінчується, а гравець отримує виграш, розрахований за наступними правилами: якщо орел випав при першому підкиданні, гравець отримує 2^0 , при другому підкиданні — 2^1 і так далі: при n -ному підкиданні — 2^{n-1} . Іншими словами, виграш зростає від підкидання до підкидання вдвічі, пробігаючи по ступенях двійки — 1, 2, 4, 8, 16, 32 і так далі.

Питання: Який вступний внесок повинно взяти казино з гравця, щоб не залишитися в програші?

Відповідь: Нескінченно великий.

Математичне сподівання суми, яку повинно виплатити казино:

$$\mu = \frac{1}{2} + 2/4 + \dots + 2^{n-1}/2^n + \dots = \infty;$$

Ніколя Бернуллі запропонував ідею вирішення парадоксу. Він припустив, що люди будуть нехтувати малоймовірними подіями. Оскільки в петербурзькій лотереї лише малоймовірні події приносять високі виграші, які ведуть до нескінченної очікуваної вартості, це може вирішити парадокс. Ідея ймовірнісного зважування знову виникла набагато пізніше в роботі Даніеля Канемана та Амоса Тверскі над теорією перспектив.

Математичне сподівання

Математичне сподівання, також відоме як середнє значення, є однією з основних характеристик статистичних розподілів і числових послідовностей в математиці та статистиці. Воно вказує на середню арифметичну величину в множині чисел або результатів випробувань. Математичне сподівання визначається за допомогою наступної формули:

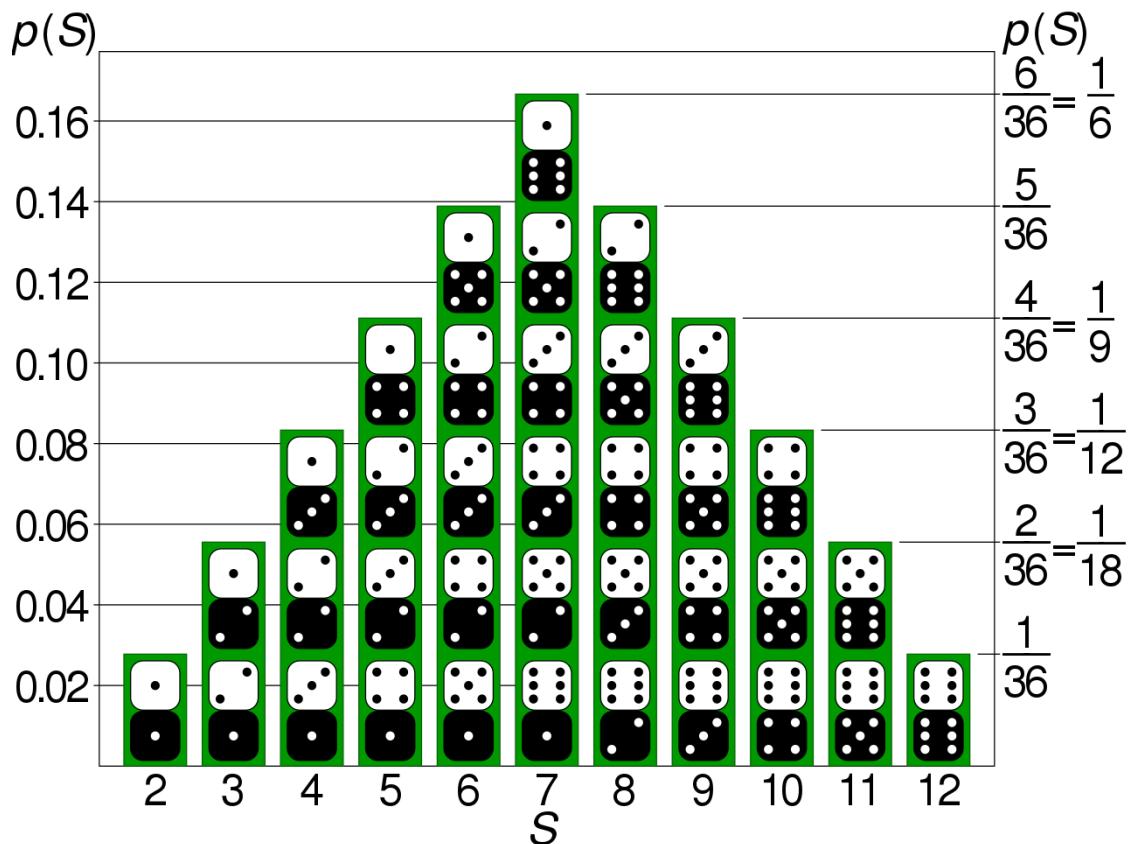
$$\text{Математичне сподівання } (\mu) = (x_1 * p_1) + (x_2 * p_2) + \dots + (x_n * p_n)$$

де x_1, x_2, \dots, x_n - це можливі значення змінної, і p_1, p_2, \dots, p_n - ймовірності відповідних значень. Зазвичай величина x_i множиться на ймовірність його виникнення і всі ці значення сумуються разом, щоб отримати математичне сподівання.

Математичне сподівання зображає центральну тенденцію в розподілі даних. Воно може бути корисним для прогнозування середнього результату в експериментах або для порівняння різних розподілів. Наприклад, математичне очікування для рівномірного розподілу на відрізку $[a, b]$ дорівнює $(a + b) / 2$, тобто середньому значенню між a і b .

Гюйгенс вже знав поняття математичного сподівання. У своєму трактаті про ймовірності Гюйгенс написав, перефразуючи його, наступне: припустимо, що в одному кулаку у людини 3 монети, а в іншому — 7 монет. Вам потрібно обрати один з кулаків і, в залежності від того, скільки буде монет, ви отримаєте стільки ж. Яку середню кількість монет ви отримуєте в такій грі. Відповідь 5. Відповідь визначається за формулою $(3 + 7) / 2$, де 2 — кількість доданків. У цьому випадку математичне сподівання буде $(3 + 7) / 2$, тобто 5. Таким чином, граючи в цю гру протягом тривалого часу, людина матиме в середньому $5 * n$ монет, де n — кількість ігор.

Дискретний розподіл ймовірностей для суми двох гральних кубиків

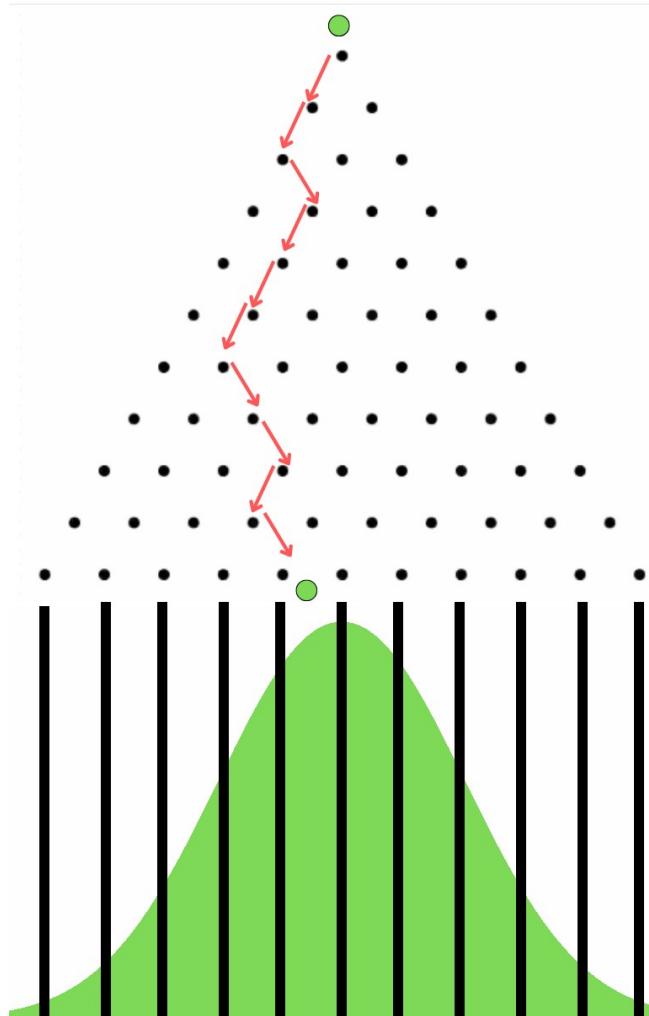


Закон великих чисел — це фундаментальний математичний принцип, який описує поведінку середнього значення великої кількості випадкових подій. Цей закон стверджує, що якщо велика кількість незалежних, однаково розподілених випадкових подій відбувається багато разів, то середнє арифметичне цих подій зближується до математичного сподівання або очікуваного значення цих подій. Іншими словами, чим більше випробувань ми проводимо, тим більше ймовірність того, що середнє значення співпаде з очікуваним значенням. Ключові ідеї та формулювання цього закону з'явилися в роботах Якоба Бернуллі, який виклав основні принципи закону у своєму творі "Мистецтво припущення", опублікованому у 1713 році.

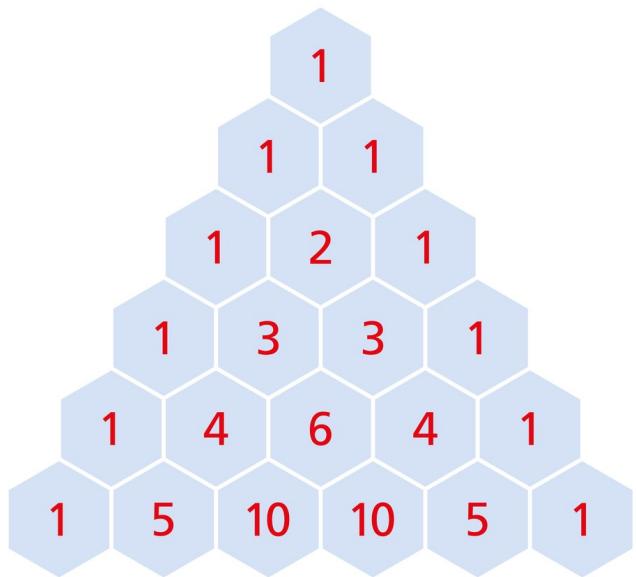
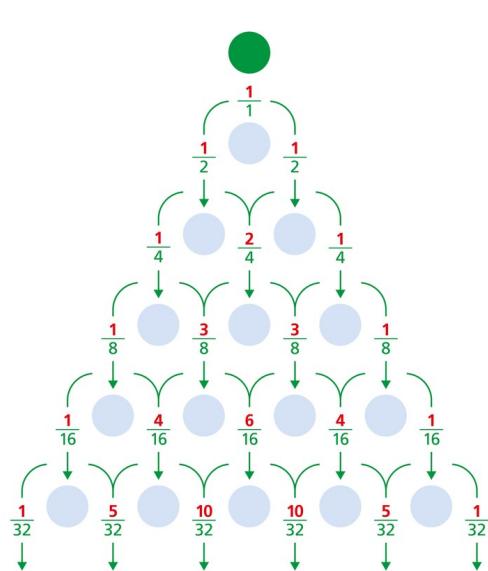
Дошка Гальтона

Дошка Гальтона — пристрій, винайдений англійським вченим Френсісом Гальтоном (перший примірник був виготовлений в 1873 році, потім пристрій був описаний Гальтоном у книзі “Природне успадкування”, виданій в 1889 році) і призначений для демонстрації центральної граничної теореми. Дошка Гальтона являє собою коробку з прозорою передньою стінкою. Стрижні (цвяхи) вбиваються в задню стінку в шаховому порядку, утворюючи трикутник.

Зверху в ящик через лійку (вихід з якої розташований рівно посередині між лівою і правою стінками) закидаються кульки. В ідеалі при зіткненні зі стрижнем кожен раз м'яч з однаковою ймовірністю рухатиметься або вправо, або вліво. Нижня частина ящика поділена перегородками (кількість яких дорівнює кількості стрижнів у нижньому ряду), в результаті чого кульки, скочуючись на дно ящика, утворюють стовпчики, які вище, чим більше вони до середини дошки (при досить великій кількості куль зовнішній вигляд стовпчиків наближається до кривої нормального розподілу).



Якщо намалювати трикутник Паскаля на задній стінці, то можна побачити, скількома способами можна дістатися до кожного зі стрижнів (чим ближче стрижень до центру, тим більше шляхів веде до нього).



Псевдовипадкові числа

Випадкові числа — це числа, які генеруються без будь-якого очевидного порядку чи системи. Їх використовують у багатьох галузях, включаючи математику, статистику, комп'ютерні науки та ігри. Вони можуть бути важливі, наприклад, при моделюванні випадкових подій або створенні криптографічних ключів. Тобто, вони не мають певного порядку чи правила, за якими вони генеруються, і виглядають, ніби вони вибираються випадковим чином.

Псевдовипадкові числа — це числа, які виглядають як випадкові, але насправді генеруються за допомогою детермінованого процесу чи алгоритму. Вони не є справжньо випадковими, оскільки, якщо ви знаєте початкові умови та алгоритм, ви можете передбачити, яке буде наступне число.

Генератори псевдовипадкових чисел зазвичай використовують алгоритми, які створюють послідовність чисел, що здається випадковою. Але насправді, якщо ти знаєш початкове значення, ти можеш передбачити всю послідовність.

Один з простих методів — це лінійний конгруентний метод.

Є формула:

$$X_{n+1} = (a * X_n + c) \bmod m$$

Де:

X_n - поточне число,

a - множник,

c - приріст,

m - модуль (максимальне значення числа).

Цей метод досить простий, але не надійний у криптографічному сенсі через те, що початкове значення може бути визначене. Для складніших випадків використовують складніші алгоритми, такі як Вихор Мерсенна (Mersenne Twister).

Вихор Мерсенна — генератор псевдовипадкових чисел (ГПВЧ), розроблений 1997 року японськими вченими Макото Мацумото і Такудзі Нісімурою. Вихор Мерсенна ґрунтуються на властивостях простих чисел Мерсенна (звідси й назва) і забезпечує швидке генерування високоякісних за критерієм випадковості псевдовипадкових чисел.

Існування чогось справді випадкового в природі є темою досліджень. Проте на практиці фізики називають "випадковими" події, які складно передбачити, бо вони мають складні правила, залежить від багатьох факторів, умов. Теорія хаосу почала розвиватися з кінця 19 століття, після того, як були відкриті процеси, в яких мінімальні зміни на початку, приводять до зовсім інших результатів. Для такої чутливості системи навіть ввели термін "Ефект Метелика", який взяли з книги американського письменника Рея Бредбери. Ефект метелика виникає коли незначний вплив на систему може мати великі та непередбачувані ефекти де-небудь в іншому місці та в інший час.

В математиці все простіше, тому що математики зазвичай самі задають ймовірності деяких подій, на основі певної логіки та працюючи з ідеальними об'єктами (ідеально симетрична монета, кубик тощо). Наприклад, математик може сказати, що ймовірність в грі "орел і решка" буде $1/2$, хоча реальна монета може падати частіше на якусь певну сторону, бо вона можливо не симетрична (не збалансована). Розподіл ймовірностей все ж можна виявити емпірично, на основі багатьох експериментів.

Французький математик П'єр Лаплас оптимістично вважав, що може передбачити майбутній стан системи, знаючи її теперішній стан (параметри). Але Анрі Пуанкарє показав, що навіть для "задачі трьох тіл" важко точно щось передбачити через чутливість системи. Лаплас писав: "Геометр у його формулах тепер охоплює всю сукупність Сонячної системи та її послідовні зміни" (П'єр Лаплас, "Система світу", Париж, 1795). Як виявилося, все значно складніше. В системі з великою чутливістю до початкових параметрів, приблизні розрахунки втрачають цінність, бо будуть кардинальні розбіжності в результатах. (Приклад такої чутливої системи - подвійний маятник.)

Генерувати випадкові числа можна створюючи хеш суму кadrів відео на якому кипить вода. Бурління води можна назвати випадковим процесом, бо його важко передбачити. В електроніці для генерації випадкових чисел використовують вакуумні лампа й діоди, які можуть створювати тепловий шум.

Розподіл Пуассона

Розподіл Пуассона — розподіл дискретного типу випадкової величини, що є числом подій, що відбулися за фіксований час, за умови, що ці події відбуваються з деякою фіксованою середньою інтенсивністю і незалежно одна від одної.

Ось приклади задач, які можна розв'язати з допомогою формули Пуассона:

У великому офісному будинку працює техпідтримка, яка отримує в середньому 6 телефонних дзвінків за годину. Вам цікаво спрогнозувати, яка ймовірність рівно трьох дзвінків протягом наступної години.

Для цього використаємо формулу Пуассона:

$$P(k) = (e^{-\lambda} * \lambda^k) / k!;$$

У нашому випадку $k = 3$, $\lambda = 6$ тому:

$$P(3) = (3^{-6} * 6^3) / 3! = 0.09;$$

Причому $P(3) < P(6)$ при середній частоті $\lambda = 6$.

Код мовою JavaScript

```
function poissonDistribution(L, k){  
    return (Math.pow(L, k) * Math.pow(Math.E, -L)) / factorial(k);  
}  
  
function factorial(n){  
    if(n<1) return 1;  
    const arr = Array.from(Array(n+1).keys());  
    arr.shift();  
    return arr.reduce((accum, next)=>{  
        return accum * next;  
    });  
}  
  
console.log(poissonDistribution(50, 6));
```

Термін L^k представляє кількість способів, якими можуть відбуватися k подій, які можуть відбуватися в даному інтервалі, де L — середня частота подій.

Термін $-e^{-L}$ є експоненціальним коефіцієнтом розпаду. Він відображає ймовірність спостереження нульових подій на малому інтервалі. Зі збільшенням L цей термін зменшується, що вказує на меншу ймовірність спостереження за подіями.

Факторіал $k!$ представляє кількість способів, якими можна впорядкувати k різних подій.

$A(n,k)$ - означає кількість комбінацій, без дублювання, довжиною k із набору з n елементів.

$$A(n,k) = n!/(n-k)!.$$

$A(n,k)$ - означає кількість комбінацій, які допускають дублювання, довжиною k із набору з n елементів.

$$A(n, k) = \text{pow}(n, k).$$

Що таке упередження виживання?

Упередження виживання (Survivorship bias) — це вид систематичного спотворення, яке виникає, коли в аналізі враховуються тільки "виживші" елементи чи події, ігноруючи ті, які не вижили або не взяли участі. Це може привести до некоректних висновків або оцінок, оскільки враховуються лише ті дані, які залишилися "вижившими".

У контексті фінансів або інвестицій, це може означати спотворення результатів через врахування тільки тих компаній чи активів, які вижили або показали успіх, і ігнорування тих, які зазнали невдачі.

Наприклад, якщо дивитися на успішних підприємців і вивчати їхні історії успіху, ігноруючи тих, хто зазнав невдачі, можна неправильно вважати, що певні рішення чи стратегії завжди призводять до успіху.

Для уникнення упередження виживання важливо розглядати і аналізувати всі дані, включаючи ті, що можуть представляти ті, хто не вижив чи не взяв участі в дослідженні.

Важливо враховувати не тільки дані, які зустрічаються часто, а й дані які зустрічаються рідко.

Під час Другої світової війни статистик Абрахам Вальд врахував упередження виживання у своїх розрахунках мінімізації втрат бомбардувальників від ворожого вогню. Дослідники Центру військово-морського аналізу провели дослідження пошкоджень літаків, які повернулись з завдань, та порекомендували додати броні у місця, пошкодження яких було найчастіше і найбільше. Вальд зазначив, що дослідження врахувало лише літаки, які повернулися з завдання — збиті бомбардувальники не були присутні для оцінки їх пошкоджень. Таким чином, дірки у літаках, що повернулися, були ділянками, попадання в які насправді дозволяє бомбардувальнику повернутися на базу. Натомість він запропонував ВМС додати броні на ділянки, які були неушкоджені на вцілілих літаках, оскільки при попаданні у ці ділянки, літак буде збитий.

Аксіоми теорії ймовірності

З допомогою цих аксіом можна довести Нерівність Чебишова та Закон великих чисел (Бернуллі). Стандартні аксіоми теорії ймовірності (на основі теорії множин) розроблені російським математиком Андрієм Колмогоровим у 1933 році.

Аксіоми теорії ймовірності:

1. $\exists S (S \neq \emptyset)$.

Існує множина S елементарних подій (результатів). (Скажімо, ми кидаємо монету і дивимося, з якого боку вона впала. У випадку з монетою S складається лише з двох елементів: перша сторона монети та друга сторона.)

2. $\exists F (F = \text{Pow}(S))$.

Існує множина F усіх підмножин S , це означає, що F є булеаном (множина всіх підмножин) S . Якщо $|S| = x$, тоді $|\text{Pow}(S)| = |F| = \text{pow}(2, |S|)$.

$\text{Pow}(S) = F$ та $\cup F = S$.

3. $\exists P (\forall x \in P (\exists y, z ((y \in F \wedge z \in [0, 1]) \wedge (x = \langle y, z \rangle)))$ або $\exists P (P : F \rightarrow R)$.

Існує функція P , домен (область визначення) якої заданий F , а кодомен (область значень) $\in L = \{x \in R : x \geq 0 \wedge x \leq 1\}$;

Уточнення:

$a \leq b$ означає $a < b$ або $a = b$;

$[0, 1]$ - множина (відрізок) усіх дійсних чисел від 0 до 1, включаючи 0 і 1.

R - множина дійсних чисел.

$\langle y, z \rangle$ - впорядкована пара.

4. $\forall x \in F (((x = S) \rightarrow (P(x) = 1)) \wedge (((S \supset x) \wedge !(x = S)) \rightarrow (P(S) > P(x))))$.

Імовірність усіх подій дорівнює 1.

5. $P(A \cap B) = P(A) + P(B)$, де $A, B \in F \wedge A \cap B = \emptyset$.

$\forall A, B \in F ((A \cap B = \emptyset) \rightarrow (P(A \cup B) = P(A) + P(B)))$.

$P(A \cup B)$ - ймовірність події, яка є принаймні в одному з двох незалежних наборів подій A і B , дорівнює сумі її ймовірностей ($P(A) + P(B)$).

Якщо всі події з S є рівніймовірними, тоді $\forall A \in F (P(A) = |A|/|S|)$.

Множину $\{S, F, P\}$ будемо називати полем ймовірності.

Наприклад, набір $\{S, F, P\}$ є

$S = \{A, B\}$.

$F = \{\emptyset, \{A\}, \{B\}, \{A, B\}\}$.

$P = \{\langle \emptyset, x_1 \rangle, \langle \{A\}, x_2 \rangle, \langle \{B\}, x_3 \rangle, \langle \{A, B\}, x_4 \rangle\}$, де $x_4 = x_3 + x_2 + x_1 = 1$.

$P(A \cap B) = |A \cap B|/|F|$.

$P(A \cap B)$ - ймовірність події, яка задовольняє дві умови для A і B одночасно.

$P(A \cap B) = 0$, якщо $A \cap B = \emptyset$.

Умовна ймовірність

$P(A|B)$ - ймовірність появи події A , якщо настало подія B .

$P(A|B) = |A \cap B|/|B|$, якщо $A \cap B \neq \emptyset$.

$P(A|B) = 0$, якщо $A \cap B = \emptyset$.

Дві події A і B ми називаємо незалежними якщо $P(A|B) = P(A)$ або $P(A \cap B) = 0$.

Один відсоток (1%) додатного числа a дорівнює одній сотій числа, тобто $a / 100$. Таким чином, p відсотків ($p\%$) від a дорівнює $a(p / 100)$. Наприклад: 10% від 55 дорівнює $55 * (10/100)$, тобто 5,5. Коли число a збільшується на p%, ми отримуємо число $a * (1 + (p / 100))$. Коли число a зменшується на p%, ми отримуємо $a * (1 - (p / 100))$.

Які найпопулярніші алгоритми та поняття в математичному аналізі?

Похідна функція

Похідна функція від дійсної змінної

Нехай f — дійсна функція, визначена на відрізку $[a, b]$. Взявши довільне число $x \in [a, b]$, визначимо:

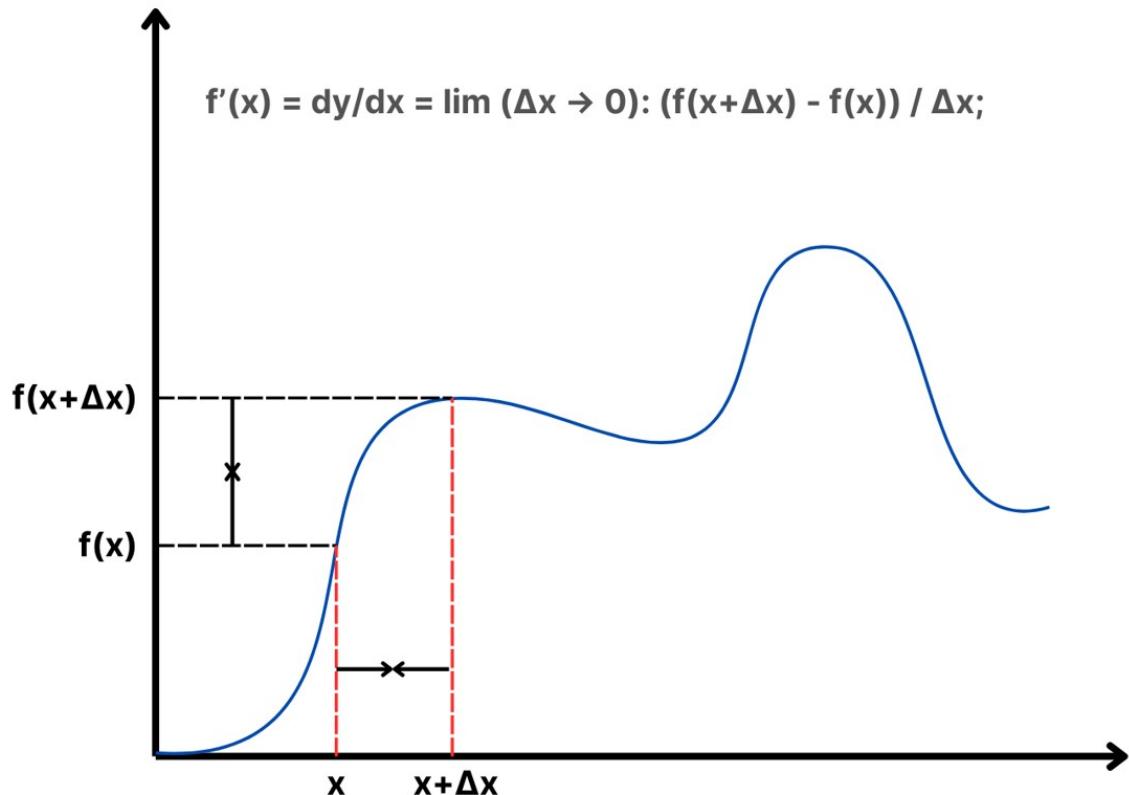
$$f'(x) = \lim_{t \rightarrow x} ((f(t) - f(x)) / (t - x)), \text{ де } a < t < b, t \neq x.$$

Також:

$$f(t) - f(x) = dy;$$

$$t - x = dx;$$

$$f'(x) = dy/dx = \lim_{t \rightarrow x} (dy/dx);$$



f' ми назвали похідну функції f .

Якщо f' визначено в точці x , ми будемо називати цю функцію f диференційованою в точці x .

З геометричної точки зору похідна функція вказує нахил кривої, точніше швидкість відхилення кривої.

Похідні функції вищих порядків.

Якщо функція f має похідну f' на деякому відрізку і якщо f' також диференційована, то ми будемо позначати похідну f' як f'' , а похідну f'' буде позначати f''' і так далі.

Якщо приріст Δy функції $f(x)$, що відповідає приросту незалежної змінної Δx (тобто $\Delta y = f(x+\Delta x) - f(x)$), можна представити у вигляді: $\Delta y = (f'(x) * \Delta x) + g(\Delta x)$, то цю функцію називають диференційованою в точці x .

Та частина приросту функції, яка лінійно залежить від приросту незалежної змінної Δx , тобто $f'(x) * \Delta x$, називається диференціалом функції. (Де $f'(x)$ — похідна функція).

Для того, щоб функція була диференційованою в певній точці, вона повинна бути неперервною в цій точці й мати ліміт послідовності в цій точці, а крім того буде визначена для всіх значень. Просто кажучи, щоб функція була диференційованою вона повинна бути гладкою (плавною). Саме тому сигмоїда диференціюється і близька до функції Хевісайда, яка має розрив і не диференціюється в нулі, її (сигмоїду) використовують як альтернативу функції Хевісайда.

Функція Хевісайда — це функція дійсної змінної значення якої рівне 0 для від'ємних значень аргументу і рівне 1 для додатних значень аргументу.

Наприклад функція $|x|$ не диференціюється, бо має прямий кут в 0.

Гладка функція або неперервно-диференційовна функція — це функція, що має неперервну похідну на всій області визначення.

У 1806 році Андре-Марі Ампер спробував аналітично довести, що кожна “довільна” неперервна функція всюди диференційована, крім “виняткових та ізольованих” значень аргументу. У цьому випадку прийнято як очевидну можливість поділу інтервалу зміни аргументу на частини, в яких функція буде монотонною. У першій половині 19 століття були зроблені спроби довести гіпотезу Ампера для ширшого класу, а саме, для всіх неперервних функцій. У 1872 році німецький математик Карл Веєрштрас вказав функцію контрприклад і представив суворий доказ її недиференційованості. Функція Веєрштраса є прикладом дійсної функції, яка скрізь неперервна, але ніде не диференційована. Це приклад фрактальної кривої. Функція Веєрштраса історично відігравала роль патологічної функції, ставши першим опублікованим прикладом (1872 р.), спеціально придуманим, щоб заперечити уявлення про те, що кожна безперервна функція диференційована, за винятком набору ізольованих точок.

Одним з основних понять механіки є поняття матеріальної точки. Під цією назвою розуміють тіло, розмірами й формою якого можна знехтувати при описі його руху. Положення матеріальної точки в просторі визначається її радіус-вектором r , компоненти якого збігаються з її декартовими координатами x, y, z . Похідна r від часу t , вона $v = dr/dt$, називається швидкістю, а друга похідна $a = d^2r/dt^2$ — прискоренням.

Де

$$r = f(t);$$

$$dr/dt = f'(t_0) = \lim_{(t \rightarrow t_0)} (f(t) - f(t_0))/t - t_0;$$

$$d^2r/dt^2 = f''(t_0) = \lim_{(t \rightarrow t_0)} (f'(t) - f'(t_0))/t - t_0;$$

v будемо позначати як **r'**.

a будемо позначати як **r''**.

У нотації Лейбніца, якщо функція y залежить від u , а u залежить від x , тоді **ланцюгове правило** формулюється так: $\frac{dy}{dx} = \left(\frac{dy}{du}\right) * \left(\frac{du}{dx}\right)$. Якщо $y = f(u)$ і $u = g(x)$, то похідна y по x обчислюється як: $y' = f'(g(x)) * g'(x)$, де f' та g' — це похідні відповідно функцій f і g .

Теорема Ролля вперше ця теорема була доведена Огюстеном-Луї Коші в 1823 році як наслідок доведення теореми про середнє значення. Обмежена форма теореми Ролля для поліномів була доведена Мішелем Роллем у 1691 році без використання математичного аналізу.

Відкритий інтервал: $(a, b) = \{x \in \mathbb{R} \mid x > a \wedge x < b\}$;

Закритий інтервал (відрізок): $[a, b] = \{x \in \mathbb{R} \mid x \geq a \wedge x \leq b\}$;

Теорема Ролля

Якщо для функції $y = f(x)$ вірно:

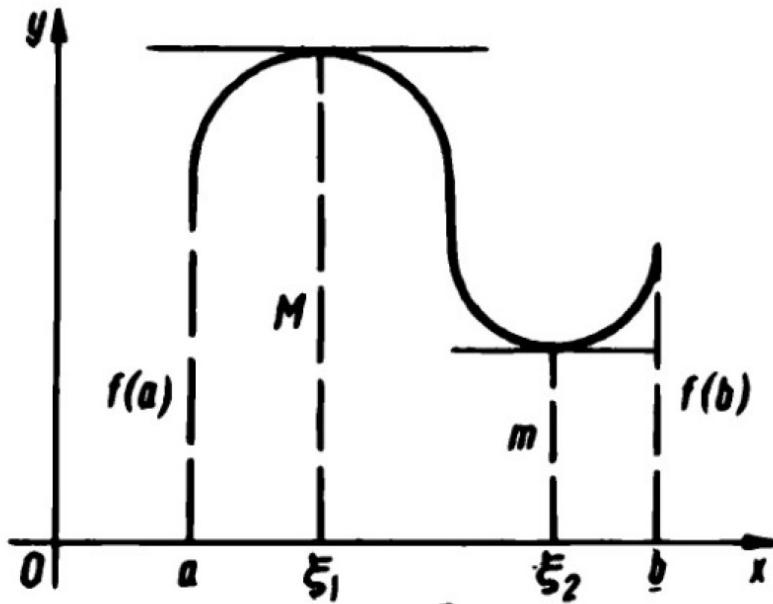
1. визначений і неперервний на відрізку $[a, b]$.
 2. має похідну в кожній точці інтервалу (a, b) .
 3. на кінцях відрізка приймає рівні значення $f(a) = f(b)$, тоді всередині відрізка $[a, b]$ є хоча б одна точка ξ , похідна в якій $f'(\xi)$ дорівнює нулю.
- З геометричної точки зору це означає, що всередині відрізка є точки ξ , в яких дотична до кривої в цих точках паралельна осі Ox .

Доказ. Оскільки $f(x)$ неперервна на відрізку $[a, b]$, то на цьому відрізку $f(x)$ приймає як найбільше значення M , так і найменше значення m .

Можливі наступні випадки.

Випадок 1. $M = m$, тоді $f(x) = \text{const}$, $a \leq x \leq b$ і $f'(x) = 0$.

Випадок 2. $M \neq m$.



Розглянемо точку ξ_1 , в якій функція має найбільше значення M , і покажіть, що похідна в цій точці дорівнює нулю.

Нехай $f'(\xi_1) \neq 0$. Зауважимо, що ξ_1 — внутрішня точка $[a, b]$, оскільки $f(a) = f(b)$ і $M \neq m$.

Потім:

а) $f'(\xi_1) > 0$. Відповідно, в околиці точки ξ_1 є точка x така, що $f(x) > f(\xi_1)$, тобто $f(x) > M$.

Ми прийшли до протиріччя, тому припущення не відповідає дійсності.

б) $f'(\xi_1) < 0$. Відповідно, існує точка x така, що $f(x) > f(\xi_1)$, тобто $f(x) > M$. Ми також прийшли до протиріччя, тому припущення не відповідає дійсності. Отже, $f'(\xi_1) = 0$. Теорема доведена.

Оператор набла

Оператор набла (∇) — це векторний диференціальний оператор у тривимірному просторі. У тривимірному просторі він записується як:

$$\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z),$$

де $(\partial/\partial x)$, $(\partial/\partial y)$, та $(\partial/\partial z)$ - часткові похідні за відповідними змінними x , y , та z .

Оператор набла може використовуватися для обчислення градієнту скалярної функції та дивергенції векторного поля.

Градієнт скалярної функції f записується як:

$$\nabla f = (\partial f/\partial x, \partial f/\partial y, \partial f/\partial z).$$

Оператор набла застосовується у лінійній алгебрі, рівняннях Максвелла (електрика) та нейронних мережах.

У нейронних мережах оператор набла використовується для обчислення градієнту функції втрати щодо параметрів мережі під час процесу навчання. Дозволяючи визначити, як швидко змінюються значення функції втрати при зміні кожного параметра, градієнт допомагає зорієнтувати процес оновлення параметрів таким чином, щоб мінімізувати цю функцію втрати.

Метод зворотного поширення помилки (backpropagation) є основним методом навчання нейронних мереж, і оператор набла грає важливу роль у цьому процесі.

Розглянемо кожен крок детальніше:

1. Прямий прохід (Forward pass):

- На початку прямого проходу вхідні дані подаються на вхід мережі.
- Вхідні дані проходять через кожен шар мережі, в якому здійснюються обчислення за допомогою ваг і функцій активування.
- На кожному шарі обчислюється вихідна активування, яка передається до наступного шару.

2. Обчислення втрат (Loss computation):

- Після того, як вихідна активування досягне останнього (вихідного) шару мережі, порівнюється здійснений прогноз зі справжніми мітками.
- На цьому етапі обчислюється величина втрат, яка відображає різницю між прогнозованими та справжніми значеннями.

3. Зворотній прохід (Backward pass):

- Після обчислення втрат починається зворотний прохід.
- Оператор набла використовується для обчислення градієнту функції втрати щодо параметрів мережі. Для цього застосовуються правила ланцюга та часткові похідні.
- Градієнт передається назад через мережу, обчислюючи градієнт для кожного шару.

4. Оновлення параметрів (Parameter update):

- Після обчислення градієнту параметри мережі оновлюються з метою мінімізації значення функції втрати.
- Для цього використовуються методи оптимізації, такі як стохастичний градієнтний спуск.
- Параметри оновлюються в напрямку, протилежному градієнту, з урахуванням швидкості навчання.

Оператор набла дозволяє систематично та ефективно обчислювати градієнт функції втрати щодо параметрів мережі, що є ключовим для навчання нейронних мереж. Це дозволяє мережі адаптуватися до вхідних даних та покращувати її прогностичні можливості під час процесу навчання.

Метод скінчених різниць

Різницева машина Беббіджа називається так тому, що в її основі лежить метод скінчених різниць, який був відомий Ньютону й описаний Ейлером в книзі “Диференціальнечислення” (1755).

Суть методу полягає в наступному: припустимо, що у нас є якась математична функція $f(x) = x^2$, визначена на натуральних числах.

Ми можемо перевірити значення цієї функції шляхом ітерації:

$$f(1) = 1;$$

$$f(2) = 4;$$

$$f(3) = 9;$$

$$f(4) = 16;$$

$$f(5) = 25;$$

Визначимо першу різницю (Різниця першого порядку).

$$df(x) = f(x) - f(x - 1);$$

$$3 = df(2) = f(2) - f(1);$$

$$5 = df(3) = f(3) - f(2);$$

$$7 = df(4) = f(4) - f(3);$$

$$9 = df(5) = f(5) - f(4);$$

Визначимо другу різницю.

$$ddf(x) = df(x) - df(x - 1);$$

$$2 = ddf(3) = 5 - 3;$$

$$2 = 7 - 5;$$

$$2 = 9 - 7;$$

Коли різниця того чи іншого порядку дає однакові значення, ми можемо зупинитися і далі використовувати ці значення.

Зауважимо, що для визначення значень 1, 4, 9, 16, 25 функції $f(x)$ довелося виконати операцію піднесення до степеня, тобто множення, яка є досить складною (реалізувати її в машинах), але знаючи скінченні різниці, ми можемо обчислити наступні значення функції без збільшення ступеня, а шляхом екстраполяції.

$$f(6) = f(5) + (f(5) - f(4)) + ((f(5) - f(4)) - (f(4) - f(3)));$$

$$f(6) = 25 + 9 + 2 = 36;$$

$$f(7) = 36 + (36 - 25) + ((36 - 25) - 9);$$

$$f(7) = 36 + 11 + 2 = 49.$$

Таким чином, ми замінюємо складніші операції на простіші. Оскільки всі комп’ютери, зрештою, дискретні, тобто вони не можуть працювати з нескінченними числами, а завжди працюють зі своїми наближеннями, тієї чи іншої точності, то цей трюк зі скінченими різницями можна зробити для будь-якої функції, тобто навіть для дійсних, замінюючи натуральні числа дійсними з тим чи іншим мінімальним значенням (кроком).

Візьмемо, наприклад, одне з рівнянь у натуральних числах (1,2,3,4,...), яке цікавило Беббіджа, а саме:
 $x^2 + x + 41$, $f(x) = x^2 + x + 41$;

$$f(x) = x^2 + x + 41;$$

$$df(x) = f(x) - f(x - 1);$$

$$ddf(x) = df(x) - df(x - 1);$$

x	$f(x)$	$df(x)$	$ddf(x)$
1	43		
2	47	4	
3	53	6	2
4	61	8	2
5	71	10	2
6	83	12	2
7	97	14	2
8	113	16	2
9	131	18	2

$$f(10) = f(9) + df(9) + ddf(9) = 151;$$

$$f(10) = 131 + 18 + 2 = 151;$$

Метод Ейлера

Метод Ейлера — чисельний метод розв'язування систем звичайних диференціальних рівнянь. Він заснований на апроксимації інтегральної кривої з допомогою так званої полілінії (ламаної) Ейлера.

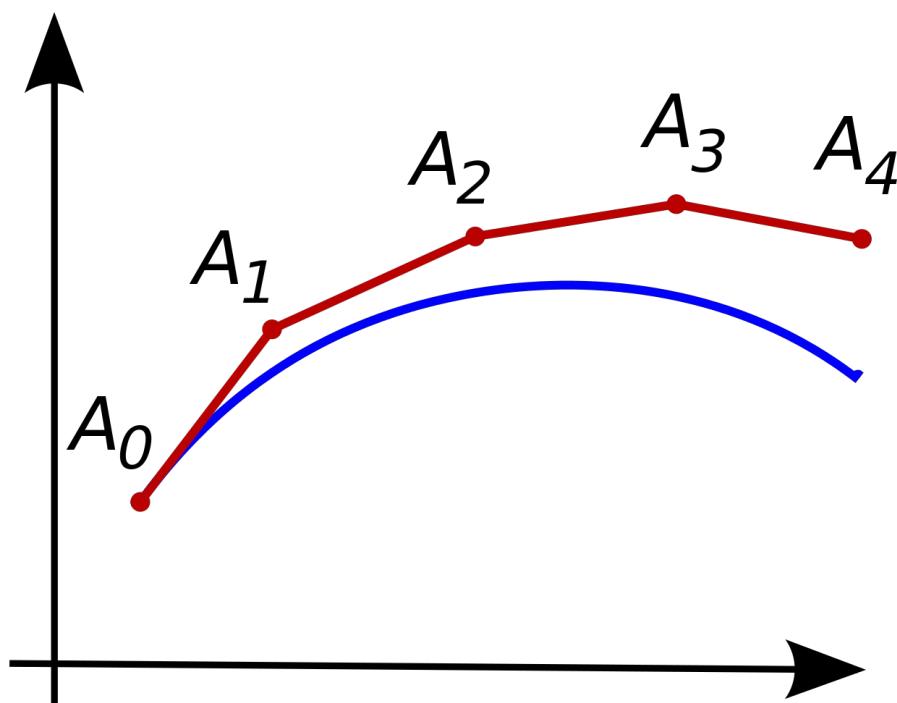
Для простоти викладу обмежимося розв'язанням задачі Коші, записаної у вигляді:

$$y' = f(x, y(x)), \quad a \leq x \leq b; \quad (\text{де } y' = dy/dx)$$

з однією невідомою функцією y та початковою умовою:

$$y(a) = 0.$$

Ми хочемо знайти повний інтеграл від dy / dx , що означає, що коли величині x надано певне значення, скажімо, $x = p$, інша змінна y повинна отримати певне значення $y = q$. Спочатку ми підходимо до цього питання так: будемо шукати значення величини y , коли величині x буде надано значення, яке не сильно відрізняється від a , тобто будемо шукати y , встановивши $x = a + \omega$. Але оскільки ω є надзвичайно маленькою частинкою, значення y також буде надзвичайно мало відрізнятися від 0, тому поки x змінюється лише з a на $a + \omega$, число $f(x, y(x))$ можна вважати постійним (тобто 0). Нехай для $x = a$ і $y = 0$ маємо $f(x, y(x))$, то під час цієї надзвичайно малої зміни ми матимемо $dy / dx = g$, інтегруючи, $y = 0 + g(x - a)$, додавши, звичайно, такий розмір константи, що при $x = a$ отримуємо $y = 0$. Тепер підставимо $x = a + \omega$, тоді $y = 0 + g(\omega)$. Таким же чином, оскільки з початково заданих значень $x = a$ і $y = 0$ ми знайшли наступні значення $x = a + \omega$ і $y = 0 + g(\omega)$, від цих значень можна послідовно рухатися далі через надзвичайно малі інтервали, поки, нарешті, ми не досягнемо значень, як завгодно віддалених від вихідних.



Ліміт (межа) послідовності

В 5 столітті до нашої ери жив філософ Зенон Елейський. Місто Елея знаходилось на території Апеннінського півострова, тобто на території сучасної Італії, в ньому проживали греки. Вважається, що саме там народився філософ Зенон, учень Парменіда. Він відомий насамперед тим, що сформулював, як мінімум, три цікаві парадокси.

Парадокси Зенона, вони ж апорії Зенона.

Парадокс під назвою Дихотомія, тобто ділення надвое.

Бігуну, щоб пробігти сто метрів, спочатку потрібно пробігти 50 метрів, але щоб пробігти 50 метрів, йому треба пробігти 25 метрів, і так до нескінченості. Висновок: Бігун ніколи не пробіжить сто метрів і навіть не зійде з місця.

Парадокс під назвою Ахіллес і черепаха.

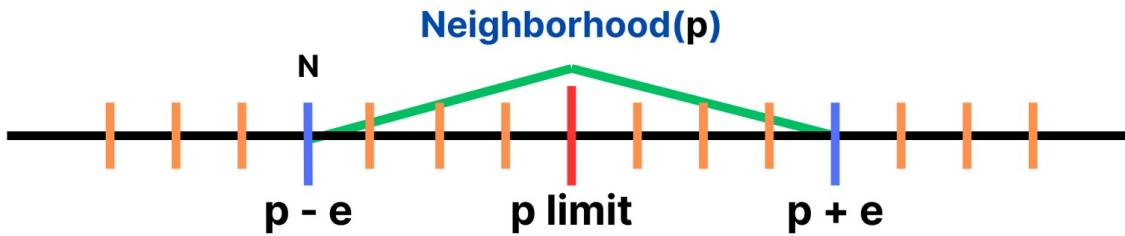
Ахіллес біжить в десять разів швидше ніж черепаха, але черепаха стартує на 10 метрів попереду від Ахіллеса. Чи зможе Ахіллес обігнати черепаху?

Припустимо Ахіллес пробіг 10 метрів, черепаха пройшла за цей час 1 метр. Тобто вона на 1 метр попереду Ахіллеса. Далі, Ахіллес пробіг один метр, черепаха пройшла 10 сантиметрів, і так до нескінченості. Висновок: Ахіллес ніколи не обгонить черепаху.

При детальному вивчені парадоксів Зенона виявилось, що вони не парадокси, а псевдопарадокси. Парадокс — це вірний розсуд, що веде до хибних наслідків. Висновки парадоксів Зенона не слідують з їх формулування. Однак, потрібно дати правильні відповіді на проблеми Зенона. Дати чітку відповідь на проблеми Зенона вчені змогли в 18 столітті, коли був розроблений аналіз нескінченно малих і розроблене поняття границі (ліміту) послідовності. Якщо число ділити до нескінченості, врешті-решт частка буде 0. Тобто в парадоксі про Ахіллеса і черепаху, Ахіллес врешті-решт стане в один ряд з черепахою, а потім обгонить її.

Запис $[\lim(n \rightarrow \infty) x_n = p] := [\exists p \in X (\forall \epsilon > 0 (\exists N (\forall n > N (d(x_n, p) < \epsilon)))]$ означає:

Лімітом (межею) послідовності $\{x_n\}$ є число p , тобто: існує число p , таке що, для будь-якого числа ϵ більше нуля, існує число N і для всіх членів послідовності більше чим N , буде вірно $d(x_n, p) < \epsilon$, тобто відстань (дистанція) між x_n та p менше ϵ .



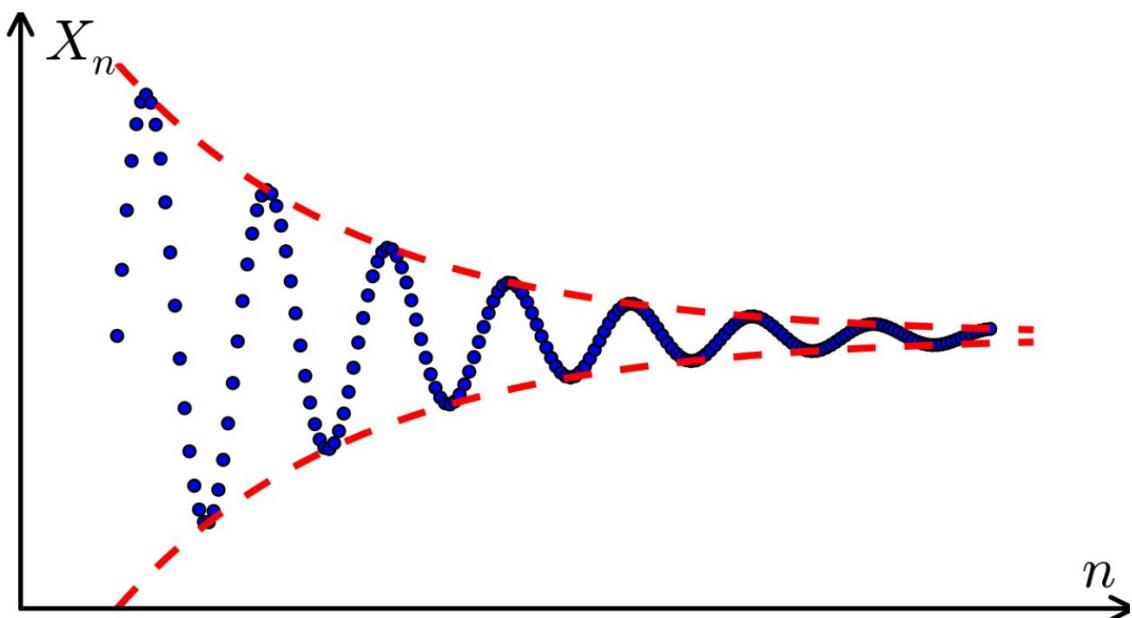
$\lim(n \rightarrow \infty) 1/n = 0$, де n – натуральне число 1,2,3,4,...

$\lim(x \rightarrow \infty) (\sum(0, n = x) 1/n) = 2$, де x – натуральне число 1,2,3,4,...

$(\sum(0, n = \infty) 1/n) = 2 = 1/1 + 1/2 + 1/3 + 1/4 + \dots$ - послідовність Зенона.

$\{x_n\}$ - означає певну впорядковану множину з елементами $\langle x_1, x_2, \dots, x_n \rangle$.

Послідовність Коші — це послідовність, елементи якої стають довільно близькими один до одного в міру просування послідовності. Точніше, якщо врахувати будь-яку невелику додатну відстань, усі елементи послідовності, окрім кінцевої кількості, менші за задану відстань один від одного.



Послідовність точок $\{p_n\}$ в метричному просторі називається збіжною, якщо вона задовольняє критерію Коші: для будь-якого $\epsilon > 0$ існує натуральне число N таке, що $d(p_n, p_m) < \epsilon$ для всіх $n, m > N$.

Критерій Коші запропонував французький математик Огюстен Луї Коші у своєму "Курсі аналізу" 1821 року.

Відповідно до критерію Коші, ряд Гранді не сходиться, тому він не має кінцевого значення.

Широко відомий ряд Гранді, названий на честь математика і католицькогоченця італійця Гвідо Гранді (1671 – 1742), який розглянув його в 1703 році.

Ряд Гранді:

$$\sum_{n=0}^{\infty} (-1)^n = 1 - 1 + 1 - 1 + 1 - 1 + \dots = ?$$

Сам Гранді не визначив значення для своєї серії, Огюстен Коші також утримався від присвоєння значення цьому ряду. Бернард Больцано (1781 — 1848) приписує цій серії 0 у своїй книзі “Парадокси нескінченості” (опублікованій у 1851 році, через три роки після смерті Больцано), але він ставить дужки: $(1-1) + (1-1) + (1-1) + \dots$, що не зовсім правомірно, оскільки природне розташування дужок у цьому рядку виглядає наступним чином: $((((1-1)+1)-1)+1)-1+\dots$, звідси значення цього ряду може бути 0 або 1, якщо він скінчений, але для нескінченного ряду відповідь складніше. Ми будемо вважати, що ця послідовність не збіжна і не має сенсу, якщо вона нескінчена, так само як не має значення послідовність:

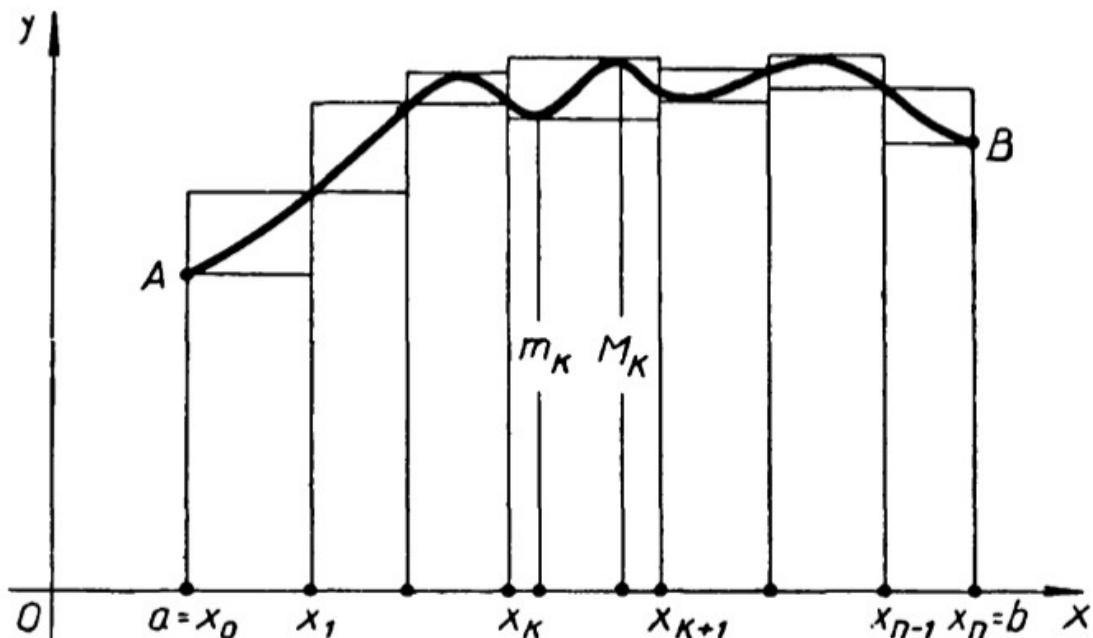
$$\sum_{n=0}^{\infty} n+1 = 1+2+3+4+5+6+7+8+9+10+11+12+13+\dots,$$

оскільки найбільшого числа не існує, бо якщо таке число існує, наприклад, позначимо його як m , то $m+1$ буде більшим за нього. Таким чином, ми методом від протиріччя довели, що найбільшого числа не існує.

Середня швидкість v_c визначається за формулою s/t , де s пройдена відстань (шлях), t - час на проходження відстані s . Швидкість тіла в даній точці траєкторії (у певний момент часу) — миттєва швидкість, — буде тим точніше означена наступним відношенням, чим менші значення Δs та Δt . В границі, коли t прямує до нуля, отримується вираз для модуля миттєвої швидкості: $v = \lim (t \rightarrow 0) (\Delta s / \Delta t) = ds/dt$;

Інтеграл

Нам потрібно обчислити площеу А фігури аABb з неправильною формою.



Розділимо відрізок $[a, b]$ на n частин точками

$$a = x_0 \leq x_1 \leq x_2 \leq \dots \leq x_{n-1} \leq x_n \leq b.$$

Позначимо через M_i найвищу і через m_i найнижчу точку (значення) функції $f(x)$ на відрізку $[x_k, x_{k+1}]$.

Сума $s = \sum_{k=0}^{n-1} m_k dx_k$, де $dx_k = x_{k+1} - x_k$, - це площа ступінчастої фігури всередині головної фігури аABb, яка побудована з усіх прямокутників з базовими лініями $[x_k, x_{k+1}]$ та висотою m_k .

Сума $S = \sum_{k=0}^{n-1} M_k dx_k$, де $dx_k = x_{k+1} - x_k$, - це площа ступінчастої фігури поза головною фігурою аABb, яка побудована з усіма прямокутниками з базовими лініями $[x_k, x_{k+1}]$ і висотою M_k .

$$A = \lim(dx \rightarrow 0) \sum_{k=0}^{n-1} m_k dx_k = \lim(dx \rightarrow 0) \sum_{k=0}^{n-1} M_k dx_k.$$

Тобто

$$A = \lim(dx \rightarrow 0) s = \lim(dx \rightarrow 0) S.$$

Таким чином ми можемо визначити суму:

$$S_s = \sum_{k=0}^{n-1} f(y_k) dx_k, \text{ де } y_k \text{ довільна точка на відрізку } [x_k, x_{k+1}].$$

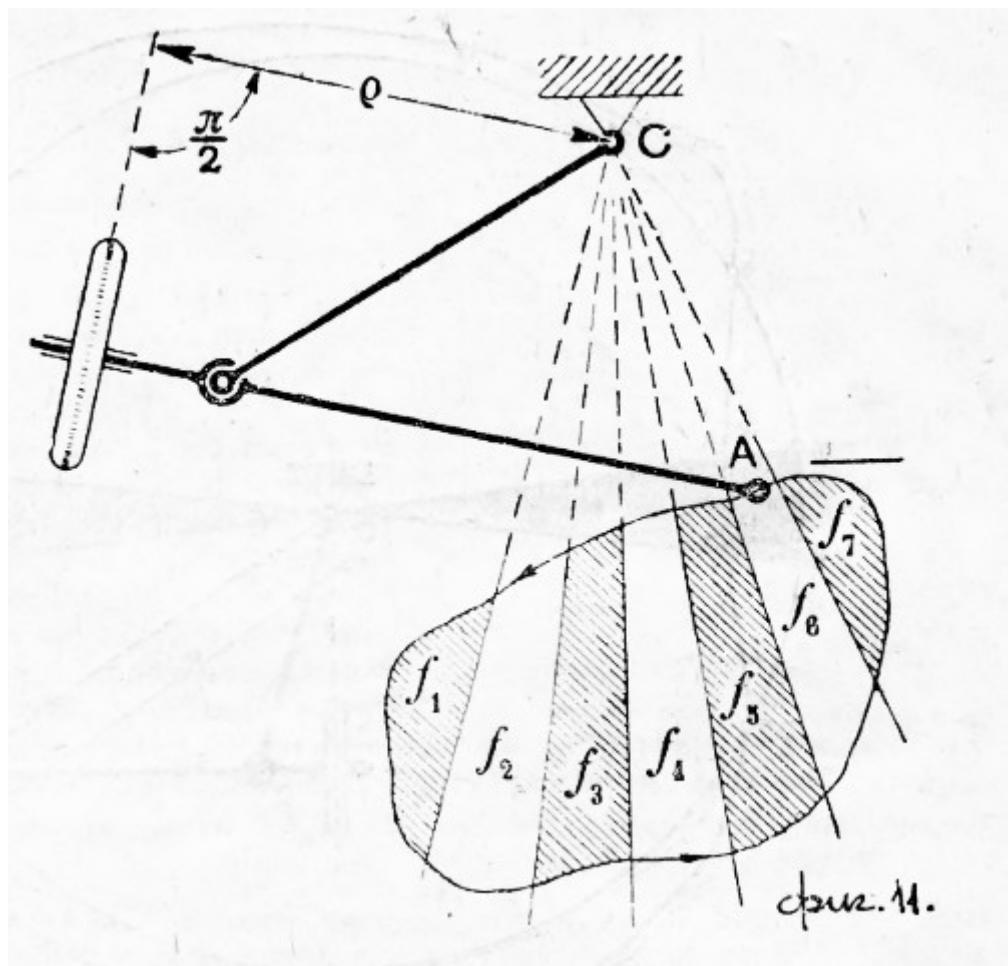
Позначимо S_s як інтеграл $\int_a^b f(x) dx$.

$$\int_a^b f(x) dx = \sum_{k=0}^{n-1} f(y_k) dx_k.$$

Планіметр

Планіметр (механічний інтегратор) — прилад, що використовується для механічного визначення площ (інтегрування) замкнтих контурів, нанесених на рівну поверхню. Масово використовувався планіметр Амслера-Кораді. Існує кілька видів планіметрів, але всі вони працюють подібним чином. Швейцарський математик Якоб Амслер-Лаффон побудував перший сучасний планіметр у 1854 році, ідея була введена Йоганном Мартіном Германом у 1814 році. Багато розробок слідували за знаменитим планіметром Амслера, включаючи електронні версії. Найперший планіметр, винайдений у 1814 році Йоганном Мартіном Германом у Баварії, використовував конусне колесо як інтегруючий механізм для вимірювання площи. Той самий принцип був реалізований у високоточних приладах з до 18 інтеграторами, ініційованими Ванневаром Бушем для оцінки диференціальних рівнянь у вищій фізиці.

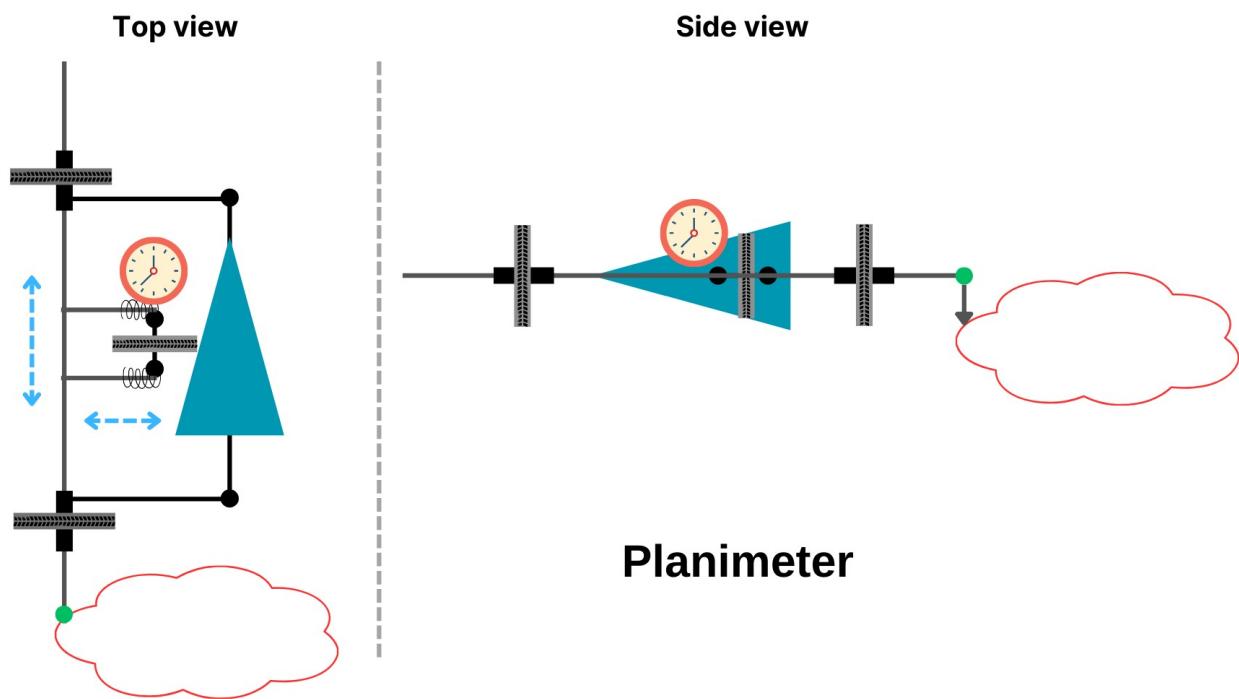
Принцип дії приладу Амслера-Кораді заснований на вимірюванні довжин дуг, описаних на поверхні, спеціальним валиком з дуже малим плямою контакту. Ролик закріплений на одному з шарнірних важелів найпростішого пантографічного механізму. Відоме положення ролика відносно ланок механізму дає можливість апроксимувати вимірюваний контур прямокутником з відомою довжиною сторін і площею, що дорівнює площі вимірюваного контуру внаслідок прокочування роликом при кожному конкретному моменті часу вздовж дуги зі строго визначеним радіусом, коли вимірюваний контур обходить вимірювальний штифт.



Планіметр – це прилад, який використовується для визначення площин ділянок різної форми. Найпоширенішим приладом на початку 20 століття був планіметр Амслера-Кораді.

Планіметр Амслера-Кораді складається з двох плечей АВ і ВС, з'єднаних кульовим шарніром в точці В. У точці С плече ВС може обертатися навколо осі, закріпленої на площині креслення. На продовженні важеля АВ розміщено каретку К, в якій закріплений ролик D так, що його вісь MN була строго паралельна лінії ВА. По контуру фігури, яку потрібно вимірюти, рухають кінець А. Ролик обертається навколо осі MN або ковзає по ній, вимірюючи площину фігури F.

Планіметр Йоганна Мартіна Германа.



Як видно на малюнку, голка може рухатися точно вздовж осей x та y . Коли голка рухається вздовж осі x , обертається конус, який повертає шестерню лічильника в ту чи іншу сторону.

Число е

Якоб Бернуллі у 1690 році вивчав число е.

Припустимо ми отримуємо сто відсотків прибутку в кінці року в залежності від суми яку ми вклали в банк, чи бізнес.

Прибуток = дохід — витрати.

Зважаючи на те, що процентна ставка залежить від суми грошей яку ми вклали, ми могли б взяти прибуток за пів року i , додавши його до початкової суми, знову вкласти його в банк, таким чином збільшуючи кінцевий прибуток. Якщо ми будемо так робити n разів на рік, відсотки для кожного інтервалу становитимуть $100\%/n$, а значення на кінець року становитиме $(1 + 1/n)^n$. Бернуллі помітив, що ця послідовність наближається до межі (сили інтересу). Щотижневе нарахування ($n = 52$) дає 2,692597 долара..., а щоденне ($n = 365$) дає 2,714567 долара... (приблизно на два центи більше).

Обмеженням у міру збільшення n є число, яке отримало назву е. Тобто при безперервному нарахуванні вартість рахунку досягне \$2,7182818...

$$e = \lim_{x \rightarrow \infty} (1 + (1/x))^x = \sum_{n=0}^{\infty} 1/n! \approx 2,718.$$

Загалом, рахунок, який починається від 1 долара і пропонує річну відсоткову ставку R , через t років принесе e^{Rt} долари з безперервним нарахуванням.

(Зверніть увагу, що R є десятковим еквівалентом процента ставки, вираженої у відсотках, тому для 5% відсотка $R = 5/100 = 0,05$.)

Літеру е для позначення цього числа почав використовувати Ейлер. Відома формула Ейлера. Відповідно, е зазвичай називають числом Ейлера.

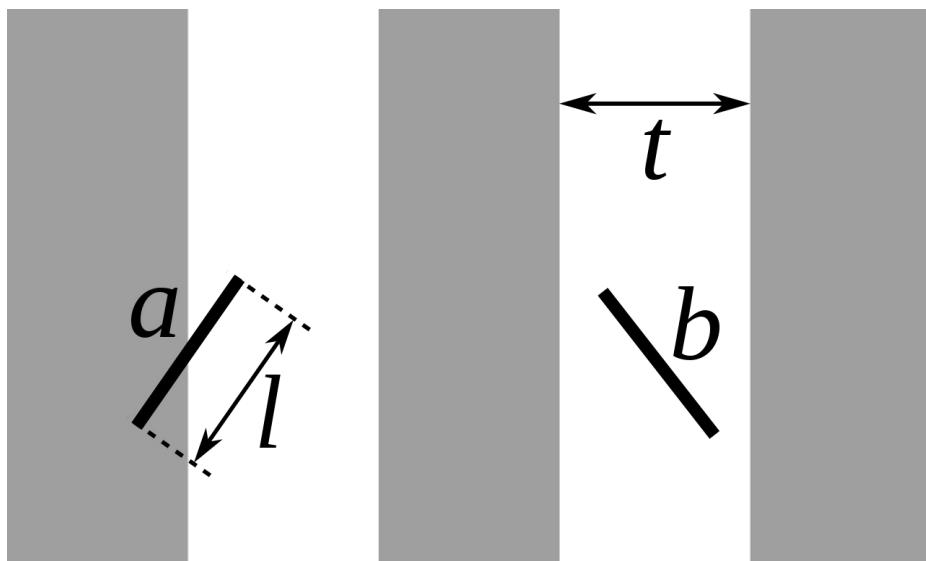
Число е трансцендентне. Вперше це було доведено у 1873 році Шарлем Ермітом. Трансцендентні числа — це числа, які не задовольняють жодне алгебраїчне рівняння з раціональними коефіцієнтами.

Джон фон Нейман разом зі Станіславом Уламом був популяризатором методів Монте-Карло, тобто методів вирішення різноманітних задач математичного аналізу на основі теорії ймовірності. Назва методу Монте-Карло походить від регіону Монте-Карло (Монако), в якому була популярна гра в рулетку.

Рулетка — це гра в казино, названа на честь французького слова, що означає маленьке колесо. Рулетку можна використовувати для генерування випадкових чисел.

Прикладом ймовірнісного методу типу Монте-Карло може бути задача Бюффона.

Задача Бюффона використовується для статистичного обчислення числа Пі. Її запропонував французький вчений Жорж-Луї Леклерк, граф де Бюффон (1707 – 1788) у 1777 році. Виявилося, що ця задача дозволила визначити число Пі ймовірнісними методами.



У математиці проблема з голкою Бюффона — це питання, яке вперше поставив у 18 столітті Жорж-Луї Леклер, граф де Бюффон:

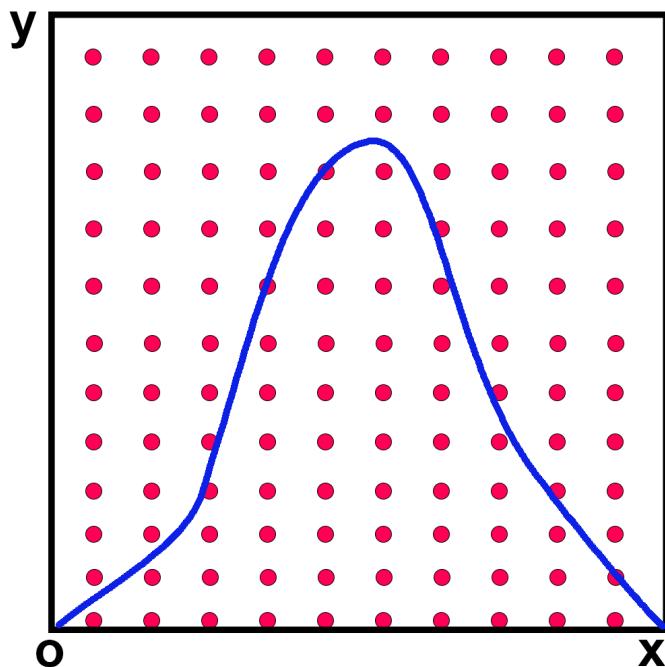
Припустимо, що у нас є підлога з паралельних смужок дерева, кожна однакової ширини, і ми опускаємо голку на підлогу. Яка ймовірність того, що голка буде лежати поперек лінії між двома смужками?

Голка Бюффона була найпершою проблемою геометричної ймовірності, яку треба було розв'язати; її можна вирішити з допомогою інтегральної геометрії. Рішення для шуканої ймовірності p , якщо довжина голки l не перевищує ширину t смужок, є

$$p = (2/\pi)(l/t).$$

Це можна використати для розробки методу Монте-Карло для апроксимації числа π , хоча це не було початковою мотивацією для запитання де Бюффона.

Геометричний алгоритм Монте-Карло для інтеграції



Для визначення площин під графіком функції можна використовувати наступний стохастичний алгоритм:

1. Обмежимо функцію прямокутником, площа А якого можна легко обчислити. Таким чином, А буде максимальним значенням інтеграла цієї функції.
2. Рівномірно (рівномовірно) “закинемо” певну кількість N точок у цей прямокутник.
3. Визначте кількість K точок, що потрапляють під графік функції.
4. Площа області S (тобто інтеграла), обмеженої функцією та осями координат, задається виразом: $S = A * (K / N)$. Чим більше точок N, тим точнішим буде обчислення інтеграла.

Метод Монте-Карло заснований на теорії ймовірності.

Перетворення Фур'є

"Жозеф Фур'є казав, що у математиці немає знаків для вираження нечітких понять"
(Фелікс Кляйн, "Лекції з розвитку математики")

Ряд Фур'є — це розкладання періодичної функції в суму тригонометричних функцій.

Перетворення Фур'є є розширенням ряду Фур'є, яке в найзагальнішому вигляді вводить використання комплексних функцій. Прикладом застосування перетворення Фур'є може бути визначення складових тонів у звуковій хвилі. Перетворення Фур'є дозволяє перейти від представлення функції $f(x)$ в просторі "час-амплітуда" до її представлення в просторі "частота-амплітуда". Перетворення названо на честь французького математика Жана Батиста Жозефа Фур'є, який ввів поняття в 1822 році.

Перетворення Фур'є:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi\xi x} dx.$$

У функцію $f'(\xi)$ як аргумент вписують частоту й отримують комплексне число, дійсна частина якого вказує на амплітуду частоти, а уявна на фазу. Якщо амплітуда для певної частоти дорівнює нулю, то це означає, що в початковій функції не міститься складова з цієї частоти. Якщо амплітуда для певної частоти додатна, то це означає, що в початковій функції $f(x)$ міститься складова з цієї частоти, а фаза вказує на фазу цієї складової. Якщо амплітуда для певної частоти від'ємна, то це означає, що в початковій функції $f(x)$ також міститься складова з цієї частоти, але з протилежною фазою.

Отже, аналізуючи амплітуди та фази компонентів в перетворенні Фур'є, можна встановити, які частоти присутні в початковій функції, а які — ні. Таким чином, можна використовувати цю інформацію для аналізу складу функції $f(x)$ та її спектра.

$e^{-i2\pi\xi x} = \cos(2\pi\xi x) - i * \sin(2\pi\xi x)$ — формула Ейлера.

ξ — частота в герцах;

$f'(\xi)$ — перетворення Фур'є, функція, яка отримує частоту (дійсне число) й повертає комплексне число; $f(x)$ — дійсна функція, частота якої аналізується і яка повертає амплітуду коливань відносно осі часу x . 2π — периметр кола з діаметром 1 в радіанах.

i — число Ейлера.

Комплексне число можна представити, як впорядковану множину з двох дійсних чисел: $\langle a, b \rangle$, причому комплексне число, яке має нульову другу компоненту, будемо вважати дійсним числом, наприклад, $\langle 2, 0 \rangle = 2$. Два комплексні числа рівні, якщо значення й порядок їхніх компонентів рівні. Додавання комплексних чисел: $\langle 2, 3 \rangle + \langle 4, 5 \rangle = \langle 2+4, 3+5 \rangle$. Множення комплексних чисел: $\langle 2, 3 \rangle * \langle 4, 5 \rangle = \langle 2*4-3*5, 2*5+3*4 \rangle$.

Уявну одиницю можна задати комплексним числом:

$$i = \langle 0, 1 \rangle;$$

$$i^2 = -1;$$

Доказ:

$$\langle 0, 1 \rangle * \langle 0, 1 \rangle = \langle 0*0-1*1, 0*1+1*0 \rangle = \langle -1, 0 \rangle = -1;$$

Алгоритм мовою TypeScript

```
function discreteFourierTransform(signal: number[]): { re: number, im: number }[] {
    const N = signal.length;
    const result: { re: number, im: number }[] = [];

    for (let k = 0; k < N; k++) {
        let re = 0;
        let im = 0;
        for (let n = 0; n < N; n++) {
            const theta = (2 * Math.PI * k * n) / N;
            re += signal[n] * Math.cos(theta);
            im -= signal[n] * Math.sin(theta);
        }
        result.push({ re, im });
    }

    return result;
}

// Приклад використання:
const signal = [1, 2, 3, 4];
const dftResult = discreteFourierTransform(signal);
console.log(dftResult);
```

Функція `discreteFourierTransform` обчислює дискретне перетворення Фур'є (ДПФ) вхідного сигналу та повертає масив об'єктів, де кожен об'єкт має дві властивості: `re` (дійсне) та `im` (уявне).

Ось що представляє вихід функції:

`re` (дійсний): дійсна частина комплексного числа, що відповідає певній частотній складовій у ДПФ. Він представляє амплітуду або силу цієї частотної складової у вихідному сигналі.

`im` (уявний): уявна частина комплексного числа, що відповідає певній частотній складовій у ДПФ. Він представляє інформацію про фазу цієї частотної складової вихідного сигналу.

ДПФ розкладає сигнал на суму синусоїдальних компонентів (синусоїда та косинусоїда) на різних частотах. Значення `re` та `im` для кожної частотної складової надають інформацію про те, яка частина конкретної частотної складової існує у вихідному сигналі та її фазу відносно вихідного сигналу.

Шукайте значення `re` та `im` з найбільшою амплітудою, оскільки це вказує на наявність сильних частотних компонентів у вхідному сигналі.

Теорема відліків

Реальні сигнали скінченні в часі і, звичайно, мають у часовій характеристиці розриви, відповідно до яких спектр нескінчений. У такому випадку повне відновлення сигналу неможливе і з теореми відліків випливають 2 наслідки:

Будь-який аналоговий сигнал можна відновити з якою завгодно точністю за його дискретними відліками, взятими з частотою $f > 2\Omega$, де Ω — максимальна частота, якою обмежений спектр реального сигналу.

Якщо максимальна частота в сигналі перевищує половину частоти дискретизації, то способу відновити сигнал з дискретного в аналоговий без спотворення не існує.

Частота ноти

"Призначення музики полягає у задоволенні слуху" – писав Леонард Ейлер у книзі "Нова теорія музики" (1739).

Точно відомо, що Піфагор та його послідовники почали вивчати музику з точки зору математики. Вони почали орієнтуватись не тільки на слух, але й на математичні співвідношення, для виявлення гарної музики. Кажуть (зокрема Нікомах Гераський), саме Піфагор виявив, що струна удвічі коротша ніж перша є співзвучною з першою. Якщо струна А вдвічі коротша ніж струна Б, тоді вона буде співзвучна зі струною Б, але її тон буде на октаву вищий, тобто в ній буде вдвічі більша частота коливань. Піфагор, як кажуть, також виявив, що струна, довжина якої становить $2/3$ від довжини першої, буде співзвучна з першою.

Тобто, якщо взяти дві однакові струни, а потім від однієї відрізати третину довжини, струни все одно будуть співзвучні. Інтервал між нотою першої струни та другої називається квінта, якщо співвідношення їх довжин буде $2/3$. Будуючи послідовний ряд квінт ви отримуєте 7 ступеневий діатонічний стрій. Відкриття цих математичних співвідношень між струнами свідчило про те, що існують певні закони слухового сприйняття. Пізніше, це породило уявлення про мозок, як апарат, що обробляє певні коливання і сприймає їх як гарні, тобто консонанси, а інші, як не гарні, тобто дисонанси, або навіть шум. На основі піфагорійського строю в 16 столітті був створений рівномірно-темперований стрій, що був точнішим ніж стрій Піфагора, адже ділив октаву на 12 звуків, а не на 7, наприклад, нота До другої октави була віддалена на дванадцять півтонів від ноти До першої октави. Людина чує звук з частотами від 16 Гц до 20 кГц.

Ми знаємо, що нота Ля першої октави має частоту коливань 440 герців, а нота Ля другої октави частоту коливань 880 герців. Співвідношення струн при цьому $1/2$. Тобто, та сама струна, що дає частоту 440 герців, буде давати частоту 880, якщо буде вдвічі коротша. Частота 440 герців означає 440 коливань за секунду.

Щоб поділити відстань між нотою Ля першої та другої октави на 12 частин нам потрібно знайти число, яке в степені 12 буде давати 2. Тобто, якщо будь-яку базову частоту помножити на це число 12 разів, це буде еквівалентно, якщо помножити цю частоту на два.

У дванадцятитоновому рівномірно-темперованому строї, який ділить октаву на 12 рівних частин, ширина півтону, тобто співвідношення частот інтервалу між двома сусідніми нотами, становить дванадцятий корінь з двох:

$$\text{root}(2, 12) = 2^{1/12} = 1.059\dots$$

Частоту ноти у рівномірно-темперованому строї можна розрахувати за формулою:

$$f(i) = f_0 * 2^{I/12},$$

де I - кількість півтонів до потрібної ноти від базової ноти з частотою f_0 .

Наприклад, якщо взяти за f_0 частоту 440 герців, тобто ноту А першої октави (5-й лад нижньої тонкої струни шестиструнної гітари), то нота С другої октави, яка на три півтони вища від ноти Ля, обчислюється за формулою:

$$440 * 2^{3/12} = 523.25\dots$$

Трактат про музичні сопілки Сіма Цяня, китайського історика ранньої династії Хань, описує 12-тонову музичну гамму. Ця гамма приблизно відповідає сучасній хроматичній гамі. Її можна отримати так: Позначимо довжину струни першого (головного) звуку, як 1.

Припустимо, що струна другого звуку дорівнює 0,95 довжини першого (1), тоді довжина струни (або трубки) третього звуку буде $(1 * 0,95 * 0,95)$, для четвертого $(1 * 0,95 * 0,95 * 0,95)$. Сам Сіма Цянь буде свою настройку, починаючи з трубки довжиною $2/3$ від основної та перемножує її та дроби, що випливають з неї, на $4/3$ або $2/3$. (Таким чином маємо довжини: 1, $2/3$, $8/9$, $16/27$, $64/81$, $128/243$ і так далі).

Які найпопулярніші алгоритми в криптографії?

Шифр Віженера

Шифр Цезаря називають на честь Юлія Цезаря, який, згідно з “Життям дванадцяти цезарів” Светонія (75 — 160), використав його зі зсувом 3, щоб захищати військові повідомлення. У шифрі Цезаря кожна літера алфавіту зміщена на кілька позицій; наприклад, у шифрі Цезаря при зсуві +3 А перетвориться на D, В стане Е і так далі. Шифр Віженера складається з послідовності кількох шифрів Цезаря з різними значеннями зміщення. Для шифрування може використовуватися таблиця алфавітів, яка називається таблиця Віженера. Що стосується латинського алфавіту, то таблиця Віженера складається з рядків із 26 символів, кожен наступний рядок зміщений на кілька позицій. Таким чином, в таблиці отримано 26 різних шифрів Цезаря. На кожному етапі шифрування використовуються різні алфавіти, підібрані залежно від характеру ключового слова.

Наприклад, припустимо, що вихідний код виглядає так:

WEAREATHEADQUARTERS;

Людина, яка надсилає повідомлення, записує ключове слово, наприклад, "STORM", циклічно, доки його довжина не буде відповідати довжині оригінального тексту:

STORMSTORMSTORMSTOM;

Перший символ оригінального тексту ("W") шифрується літерою S, яка є першим символом ключа. Перший символ зашифрованого тексту ("O") знаходиться на перетині рядка S і стовпця W в таблиці Віженера. Аналогічно, другий ключовий символ використовується для другого символу вихідного тексту; тобто другий символ зашифрованого тексту ("X") виходить на перетині рядка T і стовпця E. Решта вихідного тексту шифрується аналогічним чином.

Оригінальний текст: WEAREATHEADQUARTERS,

Ключ: STORMSTORMSTORMSTOM,

Зашифрований текст: OXOIQSMVVMVJIRDLXFE.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	
M	M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Шифр Віженера було дуже важко зламати до створення комп'ютерів. У наш час з допомогою спеціальних програм, перебором, його можна швидко зламати на комп'ютері.

Метод Казіскі — метод криптоаналізу поліалфавітних шифрів, таких як шифр Віженера.

У 1863 році німецький вчений Фрідріх Вільгельм Казіскі опублікував свою 95-сторінкову працю “Тайнопис та мистецтво дешифрування”. Це була книга про атаки на шифри, створені з допомогою поліалфавітної заміни. У цій книзі Казіскі описує своє відкриття у криптоаналізі, а саме, алгоритм, відомий усім як “Метод Казіскі”. Ідея методу заснована на тому, що ключі є періодичними, а в природній мові існують найпоширеніші буквосполучення: біграми й триграми. Це наводить на думку, що набори символів, що повторюються, в шифротексті — повторення популярних біграмм і триграмм вихідного тексту. Якщо підрядок, що повторюється, у відкритому тексті зашифровується одним і тим же підрядком у ключовому слові, тоді шифрований текст містить підрядок, що повторюється, а відстань між двома входженнями кратно довжині ключового слова.

В книзі “Зломники кодів” (1967) американський історик Девід Кан розповідає історію криптографії, зокрема, згадує про решітку Кардано, шифр Цезаря, шифр Віженера, праці Тритемія по криптографії та стеганографії.

Шифр Роберта Етьєна (Книжковий шифр).

З давніх часів книги так чи інакше поділяли на глави, але найпопулярнішим у світі є поділ Біблії на книги, глави та вірші, наприклад, Мт. 5:18. Стандартний поділ Біблії на розділи ввів у 13 столітті англійський католицький єпископ і кардинал Стівен Ленгтон, а стандартний поділ Біблії на вірші ввів у 1551 році французький книгодрукар Роберт Етьєн (відомий як Робертус Стефанус). Ця техніка розділення може бути використана в криптографії для передачі повідомлень. Замість того, щоб передати повідомлення (лист) у тій чи іншій формі, ми передаємо місця (координати) у книзі, те чи інше слово, букву. Таким чином, попередньо домовившись про те, яку книгу використовувати, ми можемо бути впевнені в безпеці наших повідомлень, якщо ця книга недоступна для тих, хто хоче вкрасти інформацію. Наприклад, ми можемо передати код:

“T2Л1С3Л5С3Л1С4Т1Л1С7Т4Л8С4Т1Л131Л232Л13632Л432Л532”, що означає: “ви не знаєте що написано”. Т – код книги (Послання Павла до Солунян, переклад Івана Хоменка), Т1 – перший розділ Т. Л – (рядок, лінія). Л7 – сьомий рядок. С – слово. С2 – друге слово. З – знак, літера. З3 – третій знак. Цей код абсолютно безпечний лише в тому випадку, якщо зловмисник не знає, яку книгу шукає, і ви постійно змінюєте розташування символів, наприклад, ви можете взяти букву А для одного слова з різних місць у книзі, тим самим не роблячи зрозумілим, що це ті самі букви. Ale варто пам'ятати, що в книжковому шифрі Етьєна немає остаточної інформації, він містить лише координати.

Надійність книжкового шифру Етьєна залежить від того, наскільки добре вибрана книга та як ефективно змінюються розташування символів. Якщо ви використовуєте важкодоступну книгу і регулярно змінюєте правила шифру, цей метод може бути досить безпечним. Проте слабкість може виникнути, якщо книга виявиться відомою або якщо шаблони змін стають передбачуваними для атакуючого.

Шифр Бекона — це метод стеганографічного кодування повідомлень, розроблений Френсісом Беконом. Щоб закодувати повідомлення, кожна літера відкритого тексту замінюється групою з п'яти літер, серед яких “а” або “б”. Наприклад, А - аaaaa (двійкова система 00000). Б — ааааб (двійкова система 00001) і так далі. Відповідно, Бекон один з перших почав використовувати бінарний код (білітеральну абетку).

Приклад. Нам потрібно надіслати секретну фразу: “Я Є Ф Б”. (Означає: Я є Френсіс Бекон).

Ми перекладаємо наші літери у двійковий код.

Я — абааа.

Є — аaaaa.

Ф — абабб.

Б — аабаб.

“Я Є Ф Б” кодується як “абааа аaaaa абабб аабаб”.

Тепер сховаємо нашу фразу в якийсь текст.

Лист:

“В природі живуть: **птах**, **барс**, **качка**, **саламандра**, **рак**, **жаба**, **бобер**, **лама**, **бабуїн**”.

Як бачите, наша фраза поширина по всьому тексту. Вам просто потрібно записати послідовно всі літери “а” та “б”, що зустрічаються в тексті та поділити їх на рядки по 5 символів.

Шифр Вернама

Шифр Вернама — симетрична система шифрування. Його основна ідея проста, ми додаємо двійкову інформацію з двійковим ключем, що дорівнює довжині інформації, з допомогою оператора ”виключне або” (також виключна диз'юнкція). Шифр названий на честь американського телеграфіста Гілберта Вернама, який в 1917 винайшов, а в 1919 запатентував систему автоматичного шифрування телеграфних повідомлень.

Операція $(a + b) \% 2$ також називається ”виключне або” (позначається як \vee). Операція $A \vee B \in \{0, 1\}$ істинною, тільки якщо $1 \vee 0 = 1$, або $0 \vee 1 = 1$, інакше вона хибна.

Щоб отримати зашифрований текст, двійковий відкритий текст об'єднується з допомогою операції ”виключне або” з двійковим секретним ключем. Так, наприклад, при використанні ключа (11101) для літери ”A” (11000), ми отримуємо зашифроване повідомлення (00101): $(11000) \oplus (11101) = (00101)$. Знаючи, що для отриманого повідомлення ми маємо ключ (11101), легко отримати вихідне повідомлення тією ж операцією: $(00101) \oplus (11101) = (11000)$. Для абсолютної криптографічної міцності ключ повинен мати три критичні властивості:

1. Мати випадковий рівномірний розподіл.
2. Відповідати розміру вказаного тексту.
3. Використовуватись тільки один раз.

Приховувати потрібно не тільки алгоритм шифрування й спосіб передачі даних, а й час виконання шифрування. Частотний аналіз даних (наприклад, ймовірність зустрічі певного символу у шифротексті) може дати підказку про використаний алгоритм шифрування й вхідні дані. Також аналіз часу шифрування й дешифрування в залежності від вхідних даних може натякнути на використовуваний алгоритм, оскільки часова складність всіх популярних алгоритмів шифрування відома.

Подання цілих чисел у двійковому вигляді

$$7 = 0111,$$

$$6 = 0110,$$

$$5 = 0101,$$

$$4 = 0100,$$

$$3 = 0011,$$

$$2 = 0010,$$

$$1 = 0001,$$

$$0 = 0000,$$

$$-1 = 1111,$$

$$-2 = 1110,$$

$$-3 = 1101,$$

$$-4 = 1100,$$

$$-5 = 1011,$$

$$-6 = 1010,$$

$$-7 = 1001,$$

$$-8 = 1000,$$

$$3 + 2 = 0011 + 0010,$$

$$0011 + 0010 = 0101,$$

$$0101 = 5,$$

$$-3 + -2 = 1101 + 1110,$$

$$1101 + 1110 = 1011,$$

$$1011 = -5.$$

Алгоритм Гаффмана — це алгоритм, призначений для стиснення повідомлень без втрат, після стиснення повідомлення ми отримуємо код Гаффмана. Алгоритм був розроблений аспірантом Массачусетського технологічного інституту Девідом Гаффманом під час написання ним курсової роботи та надрукований в статті 1952 року “Метод побудови кодів з мінімальною надлишковістю”.

Алгоритм заснований на наступному принципі:

1. Повідомлення надано певною мовою (скажімо, англійською).
2. Кожна літера вихідної літери зазвичай зберігається в цифровому файлі як двійкове число, двійковий код.
3. Може статися, що в тексті листа часто зустрічаються букви з найбільшою довжиною двійкового коду. (Кожна літера має власний специфічний двійковий код. Код 000 вважається меншим за 001, а 001 меншим за 011 і 111).
4. Необхідно проаналізувати повідомлення і розрахувати ймовірності появи в ньому тих чи інших літер, тобто з якою ймовірністю зустрічається та чи інша буква. (Власне, достатньо порахувати кількість кожної букви).
5. Літери, які найчастіше зустрічаються в листі, мають бути закодовані найкоротшими двійковими кодами, найрідкісніші — найдовшими.
6. Таким чином ми оптимізуємо (стиснемо) повідомлення, але текст повинен буде декодувати кінцевий адресат, для цього йому потрібно знати, з якою частотою той чи інший символ зустрічається в нашому вихідному повідомленні.

Стеганографія (від грец. “таємне письмо”) — спосіб передачі або зберігання інформації з урахуванням збереження в таємниці самого факту такої передачі (зберігання). Цей термін ввів у 1499 році абат Йоганнес Тритемій (1462 – 1516) у трактаті “Стеганографія”. На відміну від криптографії, яка приховує зміст секретного повідомлення, стеганографія приховує сам факт його існування. Стеганографія зазвичай використовується в поєднанні з методами криптографії, тим самим доповнюючи її. Поліграфія — криптографічна праця Йоганна Тритемія, опублікована в 1518 році, присвячена мистецтву стеганографії. Стеганографія і Поліграфія — це лише один твір, представлений у двох частинах: перша метафізична і цілком теоретична, другий більш практичний і використовується для кодування повідомлень.

Одним з найпоширеніших методів класичної стеганографії є використання симпатичного чорнила. Симпатичні (невидимі) чорнила — чорнила, записи яких спочатку невидимі й стають видимими лише за певних умов (нагрівання, спеціальне освітлення, хімічний проявник тощо). Як правило, процес запису здійснюється наступним чином: перший шар — це важливий запис невидимими чорнилом, другий шар — неважливий запис видимим чорнилом. Приховане письмо можна виявити, знаючи тип невидимого чорнила.

Частотний аналіз та атака по часу

Частотний аналіз:

Цей метод базується на вивченні частоти вживання символів у шифртексті. У багатьох мовах деякі символи вживаються частіше за інші. Наприклад, в англійській мові буква "e" є однією з найчастіше вживаних. Аналізуючи частоту вживання символів у шифртексті, можна намагатися визначити, які букви чи символи відповідають конкретним літерам чи символам відкритого тексту. Це особливо ефективно для monoалфавітних шифрів, де кожен символ відповідає одному символу в алфавіті.

Аналіз по часу виконання:

Цей метод використовує час виконання операцій або алгоритмів для знаходження слабких сторін шифру. Наприклад, атаки на час виконання можуть виявити різницю у часі, який вимагається для обробки правильного і неправильного ключа. Якщо атакуючий може визначити, що деякі операції виконуються швидше чи повільніше залежно від правильного ключа, це може допомогти відкрити шифр.

Наприклад, якщо в алгоритмі є умова, яка перевіряє певний біт ключа, і час виконання змінюється в залежності від того, чи цей біт правильний, то атакуючий може виміряти цей час і визначити біти ключа.

Атака по часу (timing attack) — це атака стороннім каналом, в якій нападник загрожує крипtosистемі, аналізуючи час потрібний для виконання криптографічних алгоритмів.

Атака по часу дієва в багатьох випадках:

Атаку по часу можна застосувати до будь-якого алгоритму, якому властиві залежні від даних варіації. Деякі операції, на кшталт множення, можуть мати різний час виконання залежно від аргументів.

Виявлення секретів через часову інформацію може бути відчутно легшим ніж через аналіз відомої пари відкритого тексту та шифртексту. Іноді часова інформація поєднується з криптоаналізом для покращення показника витоку інформації.

RSA

RSA (Rivest–Shamir–Adleman) — це криптосистема з відкритим ключем, яка широко використовується для безпечної передачі даних. Акронім RSA походить від прізвищ Рона Рівеста, Аді Шаміра та Леонарда Адлемана, які публічно описали алгоритм у 1977 році.

RSA — це криптосистема з відкритим ключем. Він заснований на складності задачі розкладання числа на множники.

Алгоритм RSA:

p, q — великі прості числа.

$n = p * q;$

$\phi(n) = (p-1)(q-1)$ — функція Ейлера числа n ;

e, d — натуральні числа такі, що e, d взаємно прості з $\phi(n)$ і $e * d = 1 \pmod{\phi(n)}$.

Перший абонент А генерує свій відкритий ключ $K_{pub} = \{n, e\}$ і секретний ключ $K_{priv} = \{n, d\}$.

Відкритий ключ А повідомляється всім абонентам. Будь-який інший абонент В може зашифрувати секретне повідомлення m для А, використовуючи відкритий ключ А. Отримавши зашифрований текст c , абонент А розшифрує його з допомогою свого секретного ключа.

Шифрування: $c = E(m) = m^e \pmod{n}$.

Дешифрування: $m = D(c) = c^d \pmod{n}$.

Доказ коректності алгоритму шифрування RSA

Теорема. Функції $E(m)$ і $D(c)$ визначають взаємно обернені перестановки множини чисел Z .

Доказ. $E(D(M)) = D(E(M)) = M^{ed} \pmod{n}$ для будь-якого M з Z . Ми знаємо, що e і d є взаємно оберненими за модулем $\phi(n)$, тобто

$$ed = 1 + k(p-1)(q-1)$$

для деякого цілого числа k . Якщо $M \neq 0 \pmod{p}$, то за Малою теоремою Ферма маємо

$$M^{ed} = M(M^{p-1})^{k(q-1)} = M * 1^{k(q-1)} = M \pmod{p}.$$

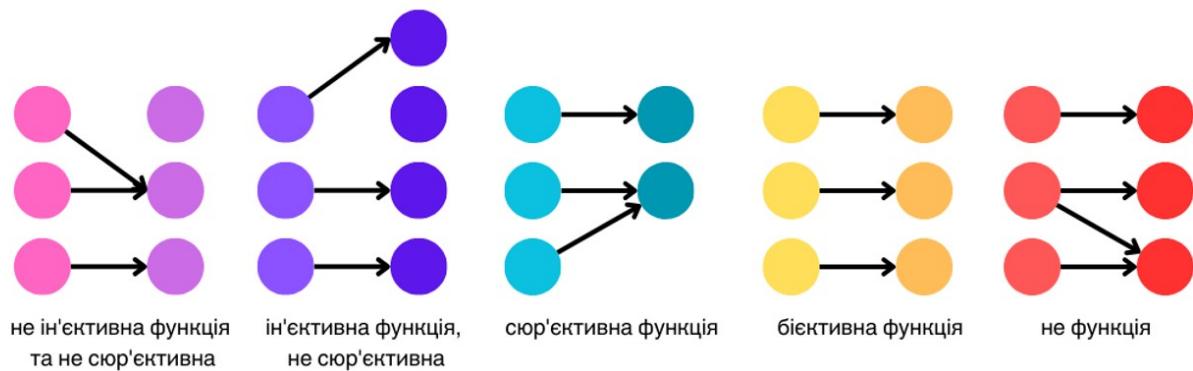
З тих же причин $M^{ed} = M \pmod{q}$, і тому $M^{ed} = M \pmod{n}$ для будь-якого M .

Які найпопулярніші алгоритми й теореми в теорії множин?

Кантор довів декілька відомих теорем.

У теорії множин теорема Шредера–Бернштейна стверджує, що коли між множинами A і B існують ін'єктивні функції $f: A \rightarrow B$ і $g: B \rightarrow A$, тоді також існує бієктивна функція $h: A \rightarrow B$.

З точки зору потужності двох множин, це класично випливає, що коли $|A| \leq |B|$ та $|B| \leq |A|$, тоді $|A| = |B|$; тобто A і B мають однакову потужність. Це корисна функція для впорядкування кардинальних чисел. Теорема названа на честь Фелікса Бернштейна (1878-1956) та Ернста Шредера (1841-1902). Вона також відома як теорема Кантора–Бернштейна.



Остання діаграма не зображає функцію, бо в ній існує елемент, якому відповідають два елементи з іншої множини.

Георг Кантор — автор двох “діагональних аргументів”, перший з яких свідчить про зліченність множини раціональних чисел, а другий — про незліченність множини дійсних чисел. Великий внесок у теорію множин зробив Георг Кантор, він є автором поняття алефів і ординалів, тобто спеціальних чисел, що позначають розмір нескінченості, хоча саме поняття “розмір нескінченості” спочатку шокувало вчених.

Діагональний аргумент Кантора про зліченість множини раціональних чисел

Легко довести, що множина раціональних чисел зліченна. Для цього достатньо навести алгоритм, який встановлює біекцію між наборами раціональних і натуральних чисел.

Прикладом такої конструкції є наступний простий алгоритм. Складається нескінчена таблиця звичайних дробів, у кожному i-му рядку в кожному j-му стовпці якої є дріб i/j . Для визначеності передбачається, що рядки та стовпці цієї таблиці нумеруються, починаючи з одиниці. Комірки таблиці позначаються (i, j) , де i — номер рядка таблиці, в якій розташована клітинка, а j — номер стовпця. Отриману таблицю обходить “змійка”. У процесі такого обходу кожне нове раціональне число пов’язується з наступним натуральним числом. Тобто, дробу $1/1$ приписується число 1, дробу $1/2$ — число 2, дробу $2/1$ — число 3 і так далі.

Множина раціональних чисел (\mathbb{Q}) є зліченою множиною, тому що ми можемо виконати її відповідність один до одного з натуральними числами наступним чином:

$1/1$	$1/2 \rightarrow 1/3$	$1/4 \rightarrow 1/5$	$1/6 \rightarrow 1/7$	$1/8 \rightarrow \dots$
$2/1$	$2/2$	$2/3$	$2/4$	$2/5$
$3/1$	$3/2$	$3/3$	$3/4$	$3/5$
$4/1$	$4/2$	$4/3$	$4/4$	$4/5$
$5/1$	$5/2$	$5/3$	$5/4$	$5/5$
$6/1$	$6/2$	$6/3$	$6/4$	$6/5$
$7/1$	$7/2$	$7/3$	$7/4$	$7/5$
$8/1$	$8/2$	$8/3$	$8/4$	$8/5$
\vdots	\vdots	\vdots	\vdots	\vdots

$$QN = \{ <1:1, 1>, <2:1, 2>, <1:2, 3>, <1:3, 4>, <2:2, 5>, <3:1, 6>, <4:1, 7>, <3:2, 8>, \dots \}$$

Теорема Кантора про потужність

Теорема Кантора про потужність. Нехай функція f — відображення з множини A в її паверсет $\text{Pow}(A)$. Тоді $f: A \rightarrow \text{Pow}(A)$ не є сюр'єктивним. Як наслідок, $|A| < |\text{Pow}(A)|$.

(Паверсет $\text{Pow}(A)$ — множина всіх підмножин множини A).

Доведення теореми Кантора (від протиріччя)

Розглянемо множину $B = \{x \in A \mid x \notin f(x)\}$.

Припустимо навпаки, що f є сюр'єктивним.

Тоді існує $\xi \in A$ таке, що $f(\xi) = B$.

Але за побудовою, якщо $\xi \in B \rightarrow \xi \notin f(\xi)$, або якщо $\xi \notin B \rightarrow \xi \in f(\xi)$.

Це протиріччя. Таким чином, f не може бути сюр'єктивним.

З іншого боку, $g: A \rightarrow \text{Pow}(A)$, визначений як $x \mapsto \{x\}$, є ін'єктивним відображенням.

Отже, ми повинні мати $|A| < |\text{Pow}(A)|$.

Пояснення

Якщо ми маємо скінченну множину $A = \{1, 2\}$ і $\text{Pow}(A)$, це за визначенням буде $\{\{\}, \{1\}, \{2\}, \{1, 2\}\}$ і припустимо, що $|A| = n$ і $P(A) = n$, отже $|A| = |P(A)|$, ми повинні мати можливість визначити функцію f , яка встановить A як домен (область аргументів) і встановить $B = \text{Pow}(A)$ як кодомен (область значень) ($f: A \rightarrow \text{Pow}(A)$) і для кожного елемента з A з допомогою f буде визначено тільки один елемент з B , до речі, що $a \neq b$, тоді $f(a) \neq f(b)$ (ін'єкція) і всі елементи з $P(A)$ були охоплені f (сюр'єкція). Але ми, очевидно, не можемо цього зробити, тому наше припущення хибне.

Якщо ми визначимо множину $B = \{x \in A \mid x \notin f(x)\}$, ми отримуємо множину всіх елементів x з A , які не входять до $f(x)$.

Наприклад,

$$x = 1, f(x) = f(1) = \{\},$$

$$x = 2, f(x) = f(2) = \{1\},$$

...

ми отримаємо:

$$B = \{1, 2, \dots\}.$$

Запитання $B \in P(A)$? Це не може, тому що якщо воно існує, то деякий x в A , що $f(x) = B$, і ми маємо нове запитання, якщо $x \in B$?

Якщо x знаходиться в B , то він не виконує умови $x \notin f(x)$, а якщо ні, то має бути в B . Ми отримуємо самопосилання (автореференція), яке дає протиріччя і, таким чином, наше припущення, що $|A| = |P(A)|$ неправильно.

За “Аксіомою потужності множини” можна зробити висновок, що не тільки для кінцевої, а й для нескінченної множини $|A| < |\text{Pow}(A)|$.

Чому, якщо $|A| = n$, тоді $|\text{Pow}(A)| = \text{pow}(2, n) = 2^n$?

Як ми знаємо, $\text{Pow}(A)$ — це множина всіх підмножин A .

Очевидно, що потужність максимальної підмножини в $\text{Pow}(A)$ буде $|A|$ і менше для інших.

Бо якщо елемент \max з $\text{Pow}(A)$ матиме кардинальне число більше, ніж A , це означає, що ми отримуємо нові елементи нізвідки.

Отже, ми можемо відобразити всю множину з $P(A)$ як двійкове число з довжиною $|A|$ і помістити одиницю в місце, де існує елемент.

Наприклад,

$A = \{1,2,3\}$, тоді $Pow(A) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$.

000 - $\{\}$

111 - $\{1,2,3\}$

100 - $\{1\}$

010 - $\{2\}$

001 - $\{3\}$

110 - $\{1,2\}$

101 - $\{1,3\}$

011 - $\{2,3\}$

Таким чином, за формулою комбінаторики маємо $|Pow(A)| = pow(2, n)$, де $2 = |\{0,1\}|$ і $n = |A|$.

Діагональний аргумент Кантора

Діагональний аргумент Кантора. Нехай у нас є нескінченна впорядкована множина A нескінченних впорядкованих множин.

Під порядком ми маємо на увазі впорядковану пару або порядок n -ї $\langle \rangle$, оскільки ми можемо покласти у відповідність натуральне число для кожного елемента з множини A .

Кожна множина з множини A містить елементи з деякої кінцевої множини E (наприклад, $E = \{0,1,2,3,4,5,6,7,8,9\}$ або $E = \{0,1\}$).

Ми позначимо кожну множину із множини A як x_1, x_2, x_3, \dots за порядком.

Ми позначимо кожен елемент у поточному x_i як $x_{i,1}, x_{i,2}, \dots x_{i,n}$ за його порядком, тобто $x_{1,1}, x_{1,2}, \dots$

Нехай у нас є впорядкована множина B , яка містить рівно один елемент з усіх x з A за порядком A , тобто $y_1 \in B$ і $y_1 \in x_{1,1} \in A$, $y_2 \in B$ і $y_2 \in x_{2,2} \in A$, $y_3 \in B$ і $y_3 \in x_{3,3} \in A$.

Ми маємо

Множина A

1: $x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, \dots$

2: $x_{2,1}, x_{2,2}, x_{2,3}, x_{2,4}, \dots$

3: $x_{3,1}, x_{3,2}, x_{3,3}, x_{3,4}, \dots$

4: $x_{4,1}, x_{4,2}, x_{4,3}, x_{4,4}, \dots$

.

.

.

і виберемо елемент для встановлення (діагональним) способом:

1: $(x_{1,1}), x_{1,2}, x_{1,3}, x_{1,4}, \dots$

2: $x_{2,1}, (x_{2,2}), x_{2,3}, x_{2,4}, \dots$

3: $x_{3,1}, x_{3,2}, (x_{3,3}), x_{3,4}, \dots$

4: $x_{4,1}, x_{4,2}, x_{4,3}, (x_{4,4}), \dots$

.

.

.

Ми маємо множину $B = [x_{1,1}, x_{2,2}, x_{3,3}, x_{4,4}, \dots]$.

Тепер визначимо функцію f , що $f(z)$, де $z \in B$, $f(z) \neq z$ і $f(z) \in E$.
Функція f поміщає для кожного елемента в B інший елемент з E .

Визначаємо множину $C = \{f(x_{1,1}), f(x_{2,2}), f(x_{3,3}), f(x_{4,4}), \dots\}$

Таким чином, A не містить C , оскільки він відрізняється від усіх упорядкованих множин в A принаймні в одній позиції.

(Якщо $x_{i,j} = x_{1,1}$ тоді C відрізняється від нього в положенні 1, якщо $x_{i,j} = x_{2,2}$ потім у позиції 2).

$s_1 = 0 0 0 0 0 0 0 0 0 0 0 \dots$
$s_2 = 1 1 1 1 1 1 1 1 1 1 1 \dots$
$s_3 = 0 1 0 1 0 1 0 1 0 1 0 \dots$
$s_4 = 1 0 1 0 1 0 1 0 1 0 1 \dots$
$s_5 = 1 1 0 1 0 1 1 0 1 0 1 \dots$
$s_6 = 0 0 1 1 0 1 1 0 1 1 0 \dots$
$s_7 = 1 0 0 0 1 0 0 1 0 0 \dots$
$s_8 = 0 0 1 1 0 0 1 1 0 0 \dots$
$s_9 = 1 1 0 0 1 1 0 0 1 1 0 \dots$
$s_{10} = 1 1 0 1 1 1 0 0 1 0 1 \dots$
$s_{11} = 1 1 0 1 0 1 0 0 1 0 0 \dots$
$\vdots \quad \vdots \ddots$

$$s = 1 0 1 1 1 0 1 0 0 1 1 \dots$$

Отже, ми завжди можемо показати новий елемент, якого немає в множині A , але якщо ми припустимо, що A є нескінченною множиною і позначається натуральними числами, це означає, що існує деяка множина, яку не можна позначити натуральними числами, оскільки ця множина більша за множину натуральних чисел.

Таким чином, використовуючи теорему Кантора та діагональний аргумент Кантора, ми можемо зробити висновок, що числом множини дійсних чисел є с або континуум.

Континуум або с, що є $a(1)$.
 $c = a(1) = \text{pow}(2, a(0))$.

Нескінчена множина X є зліченою, якщо ми можемо визначити відповідність один до одного (біекцію) від множини X до множини N натуральних чисел, інакше це незліченна множина.

Усі скінченні множини зліченні.

Ми бачимо, що нескінченна множина Z цілих чисел є зліченою, оскільки ми можемо визначити відповідність (бієктивну функцію) між нею та натуральними числами.

$$Z = \{1, -1, 2, -2, 3, -3, \dots\}$$

Для додатних чисел 1, 2, 3, ... із Z визначаємо парні числа $\langle 2, 1 \rangle, \langle 4, 2 \rangle, \langle 6, 3 \rangle, \dots$

Для від'ємних чисел -1, -2, -3, ... із Z визначаємо непарні числа $\langle 1, -1 \rangle, \langle 3, -2 \rangle, \langle 5, -3 \rangle, \dots$

Як ми знаємо, множина Парна = $\{x \in N: \exists y \in N (x = 2y)\}$ всіх парних чисел і множина Непарна = $\{x \in N: \neg(\exists y \in N (x = 2y))\}$ всіх непарних чисел мають таке ж кардинальне число, що й набір N усіх натуральних чисел.

$$|\text{Even}| = |\text{Odd}| = |N|.$$

$$\langle 1, 1, 2 \rangle, \langle 2, 3, 4 \rangle, \dots, \langle n, n+1, 2n \rangle.$$

Континуум гіпотеза — це припущення, висунуте в 1877 році Георгом Кантором, що будь-яка нескінчена підмножина континууму або злічена, або безперервна. Іншими словами, гіпотеза передбачає, що потужність континууму є найменшою, що перевищує потужність зліченої множини, і немає проміжних потужностей між зліченою множиною та континуумом. Гіпотеза континууму була першою з двадцяти трьох математичних задач, про які Давид Гільберт доповів на II Міжнародному конгресі математиків у Парижі в 1900 році. Тому гіпотеза континууму також відома як перша проблема Гільберта. Перші спроби довести це твердження з допомогою наївної теорії множин не увінчалися успіхом. У 1940 році Курт Гьодель довів, що заперечення гіпотези континууму не можна довести в системі аксіом Цермело-Френкеля з аксіомою вибору, а в 1963 році Пол Коен, використовуючи свій метод форсування, довів, що гіпотеза континууму також не доводиться в системі аксіом Цермело-Френкеля з аксіомою вибору. Просто гіпотеза континууму звучить так: чи існує між нескінченністю першого порядку, наприклад, між нескінченністю натуральних чисел, і між нескінченністю другого порядку, наприклад, між нескінченністю дійсних чисел, якась нескінченність, більше за першу і менша за другу? Відповідно до гіпотези (континууму) Георга Кантора – ні. Дійсні числа — це всі числа, необхідні для опису довжини прямої, тобто всі числа, що лежать на прямій.

Математик Ріхард Дедекінд (1831 — 1916) підтримав розробки Кантора, який вступив з ним у листування, і також розвивав теорію множин, але удар по системі Кантора, Дедекінда і Фреге, які також розвивали арифметику на основі теорії множин, зробили парадокси, які були відкриті в теорії множин, як-от парадокс Буралі-Форті, парадокс Кантора, парадокс Рассела. Сформулюємо ці парадокси усно.

Парадокс Буралі-Форті (1897): задана множина (набір) усіх порядкових чисел (ординалів), який також має власний порядковий номер. Чи входить цей номер у саму множину? Відповідаючи на це питання, отримуємо автореференцію, нескінченну рекурсію, одним словом, суперечності в будь-якому випадку. Приклад, ординали (порядкові числа): 0 - {}, 1 - {{}}, 2 - {{}, {{}}}. Множина всіх ординалів буде мати вигляд: {{}, {{}}, {{}, {{}}}, {{}, {{}}, {{}, {{}}}}, ...}, тобто сама буде ординалом, що приводить до абсурду, бо значить вона має містити саме себе.

Парадокс Кантора (1899): ми знаємо, що множина всіх підмножин множини завжди більша за початкову множину (відповідно до теореми Кантора). Тоді, якщо ми маємо набір усіх множин, позначених як U, то множина всіх підмножин U буде більшою за U, тобто міститиме нові множини, що суперечливо, оскільки U містить усі можливі множини за визначенням.

Парадокс Рассела (1901): Дано множину A, яка містить усі множини, які не містять себе як підмножини. Чи містить множина A сама себе? Відповідаючи на це питання, отримуємо автореференцію, у будь-якому випадку протиріччя.

Аксіома регулярності усуває ці парадокси.

Аксіома регулярності (аксіома фундування) — одна з аксіом теорії множин Цермело — Френкеля (ZF). Спочатку була сформульована фон Нейманом для теорії множин фон Неймана — Бернайса — Геделя (NBG) (в 1925).

Аксіома регулярності: "В будь-якій непорожній множині A є елемент B, що перетин A та B є порожньою множиною".

Наслідком Аксіоми регулярності є твердження, що не існує множини, яка є елементом самої себе.

Така суперечлива теорія множин, як вона фігурувала у Кантора, Дедекінда і Фреге, пізніше стала називатися "наївною теорією множин".

Математикам стало зрозуміло, що не кожне сформульоване речення визначає множину, їм потрібно

було обмежити довільні визначення, оскільки вони зустрічалися в парадоксах.

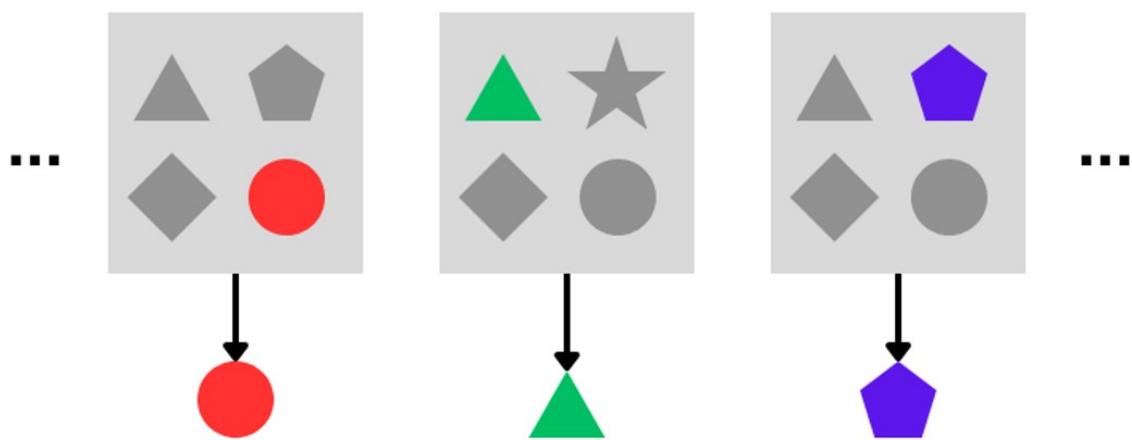
Першою успішною системою аксіом для наївної теорії множин стала система, опублікована в 1908 році німецьким математиком Ернстом Цермелю, який найбільш відомий своїми роботами з теорії множин і теорії ігор. Система аксіом Цермелю в 1921 році була доповнена ізраїльським математиком Абрахамом Френкелем (1891–1965) і детально ним описана у його книгах “Абстрактна теорія множин” та “Основи теорії множин”.

Другу успішну спробу позбутися парадоксів і поставити теорію множин на міцну формальну основу зробили англійські математики та філософи Альфред Норт Вайтгед і Берtran Рассел, який опублікував парадокс Рассела, хоча раніше він був відкритий Ернстом Цермелю (1871-1953).

Рассел та Вайтгед опублікували три томи з 1910 по 1913 рік під тією ж назвою “Математичні принципи”, в яких вони розробили систему аксіом для теорії множин, логіки предикатів та арифметики. Система, запропонована Вайтгедом і Расселом, усунула всі відомі на той час парадокси та, як писав Курт Гедель в 1931 році, створила враження, що будь-яке твердження математики можна довести з допомогою цієї системи.

Аксіома вибору. Деякий час серед математиків точилася суперечка про правомірність так званої аксіоми вибору. Аксіома вибору була сформульована та опублікована Ернстом Цермело в 1904 році. Аксіома вибору — це аксіома теорії множин. Як і фізики, математики мають деякі принципи. Аксіома вибору виражає інтуїтивний принцип, що елементи можна вибрати з кожного набору. Точніше, аксіома вибору говорить про те, що завжди можна сформулювати необхідну умову виділення тих чи інших елементів із множин. Припустимо, що у нас є множина в якій є певні елементи, чи впевнені ми, що для будь-якої множини елементів із цієї множини можна визначити умову їх виділення (фільтрації) з цієї множини? Багато математиків відповіли: так, це інтуїтивно зрозуміло і ввели аксіому вибору. Аксіома вибору не залежить від інших аксіом Цермело-Френкеля.

Аксіомою вибору є наступне твердження теорії множин: “Для будь-якого класу X непорожніх множин існує функція f , яка призначає кожній множині класу один з елементів цієї множини. Функцію f називають функцією вибору для даного класу”.



Ілюстрація аксіоми вибору, де кожен S_i та x_i представлені у вигляді коробки та кольорової фігури відповідно.

Для скінченної множини X аксіома вибору випливає з інших аксіом теорії множин. У цьому випадку це те саме, що сказати, якщо у нас є кілька коробок, кожна з яких містить одну ідентичну річ, то ми можемо вибрати рівно одну річ зожної коробки. Зрозуміло, що ми можемо це зробити: починаємо з першої коробки, вибираємо річ; переходимо до другої коробки, вибираємо річ; і так далі. Оскільки кількість коробок обмежена, то, діючи з нашою процедурою відбору, ми прийдемо до кінця.

Результатом буде явна функція вибору: функція, яка зіставляє першу коробку з першим елементом, який ми вибрали, другу коробку з другим елементом, і так далі. (Щоб отримати формальний доказ для всіх скінчених множин, скористайтесь принципом математичної індукції.)

У випадку нескінченної множини X іноді аксіому вибору також можна обійти. Наприклад, якщо елементи X є наборами натуральних чисел. Кожна непорожня множина натуральних чисел має найменший елемент, тому, визначивши нашу функцію вибору, ми можемо просто сказати, що кожна множина пов'язана з найменшим елементом множини. Це дозволяє нам вибирати елемент із кожного набору, тож ми можемо написати явний вираз, який говорить нам, яке значення приймає наша функція вибору. Якщо таким чином можна визначити функцію вибору, то аксіома вибору не потрібна.

Труднощі виникають, коли неможливо здійснити природний вибір елементів з кожного набору. Якщо ми не можемо зробити явний вибір, то чому ми впевнені, що такий вибір можна зробити в принципі? Наприклад, нехай X — множина непорожніх підмножин дійсних чисел. По-перше, ми можемо спробувати діяти так, ніби X скінчений. Якщо ми спробуємо вибирати елемент з кожного набору, то, оскільки X є нескінченим, наша процедура вибору ніколи не закінчиться, і, як наслідок, ми ніколи не отримаємо функцію вибору для всього X . Тому вона не працює. Далі ми можемо спробувати визначити найменший елемент з кожного набору. Але деякі підмножини дійсних чисел не містять найменшого елемента. Наприклад, takoю підмножиною є відкритий інтервал $(0, 1)$. Якщо x належить $(0, 1)$, то $x / 2$ також належить йому, і воно менше x . Тому вибирати найменший предмет теж не вийде. Причина, яка дозволяє вибирати найменший елемент з підмножини натуральних чисел, полягає в тому, що натуральні числа мають властивість бути повністю впорядкованими. Кожна підмножина натуральних чисел має унікальний найменший елемент завдяки природному впорядкуванню.

Неформально, в математиці, міра — це функція, що відображає множини на не від'ємні дійсні числа, при цьому, надмножини відображаються на більші числа, ніж підмножини.

У 1902 році французький математик Анрі Лебег у своїх лекціях сформулював теорію міри та гадав, що вона може бути застосована до довільної обмеженої множини. Але поява контрприкладів розвіяла ці сподівання. Побудова таких невимірних множин завжди спирається на аксіому вибору.

Множина Віталі — історично перший приклад множини, що не має міри Лебега (невимірна множина). Цей приклад опублікував 1905 року італійський математик Джузепе Віталі.

Парадокс Банаха-Тарського — теорема в теорії множин, яка стверджує, що тривимірна куля рівноскладена двом своїм копіям.

Парадокс (насправді теорема) Банаха-Тарського — це математична теорема, яка була сформульована і доведена у 1924 році польськими математиками Стефаном Банахом і Альфредом Тарським.

Зважаючи на її неправдоподібність, цю теорему часто використовують як аргумент проти прийняття аксіоми вибору, яка істотно використовується для побудови такого розбиття.

Припустимо, що ми маємо кулю в тривимірному просторі. За допомогою деякого математичного обчислення (перетворення), цю кулю можна розрізати на обмежену кількість окремих точок, а потім знову зібрати ці точки так, щоб утворилася дві одинакові кулі, кожна з яких ідентична по розмірах та формі з початковою кулею. Збирання двох куль проводиться без розтягування, а просто з допомогою трансляції (переміщення) та ротації (обертання). Для плоского кола аналогічна властивість неправильна.

Ця формально доведена теорема суперечить фізичній інтуїції, тому й називається парадоксом.

Стефан Банах (1892 — 1945) — польський математик, один із творців сучасного функціонального аналізу, один із двох засновників і безперечних лідерів Львівської математичної школи, професор університету Яна Казимира у Львові та Львівської Політехніки (з 1924). У 1922 році довід теорему Банаха про нерухому точку.

Деякі позначення теорії множин

Порожня множина - $\{\}$ або \emptyset .

Множина з одним елементом - $\{a\}$.

$A = \{a, b, c\}$.

$-A$ — обернена до множини A , яка означає множину всіх елементів, які існують у нашій області визначення, але яких немає в множині A .

$-A := \{x: \neg(x \in A)\}$.

$A \supset B$ означає, що B є власною підмножиною A , суворе включення.

$A \supset B := \forall x((x \in A \rightarrow x \in B) \wedge A \neq B)$.

$A \supset B = B \subset A$.

Об'єднання двох множин A і B

$A \cup B := \{x: x \in A \vee x \in B\}$.

Перетин двох множин A і B

$A \cap B := \{x: x \in A \wedge x \in B\}$.

Різниця A і B

$A - B$ (або $A \setminus B$) := $A \cap -B$.

Декартів добуток

$A \times B := \{(x, y): x \in A \wedge y \in B\}$.

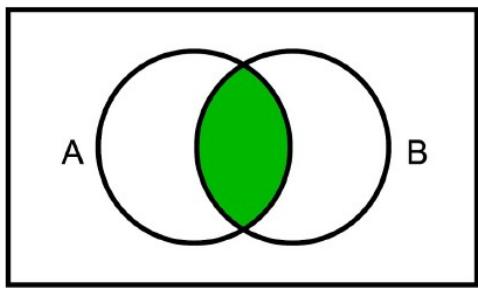
$A \times B \times C := (A \times B) \times C$.

$A \times B \times C \times D = ((A \times B) \times C) \times D$.

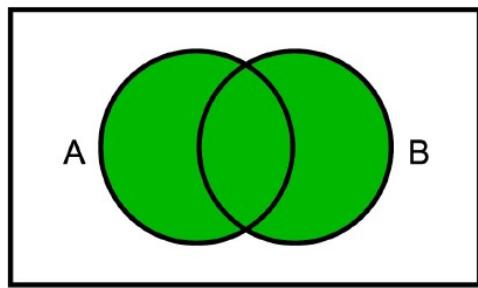
Якщо $A = \{1, 2\}$, $B = \{3, 4\}$, тоді $A \times B = \{(1, 3), (1, 4), (2, 3), (2, 4)\}$.

$A \times B \neq B \times A$.

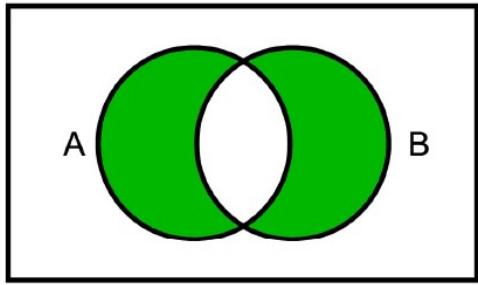
Діаграми Вена



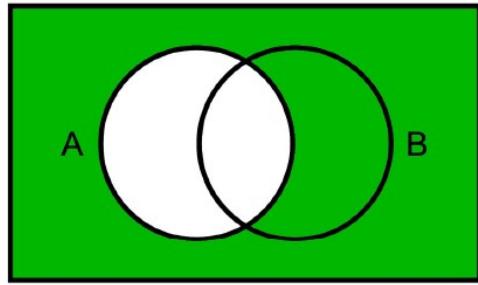
$A \cap B$



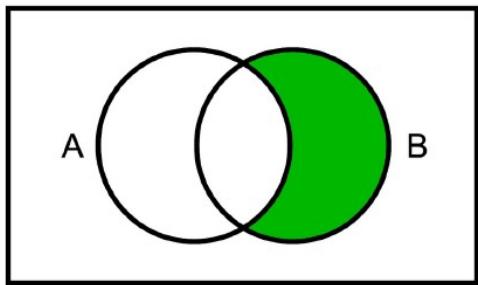
$A \cup B$



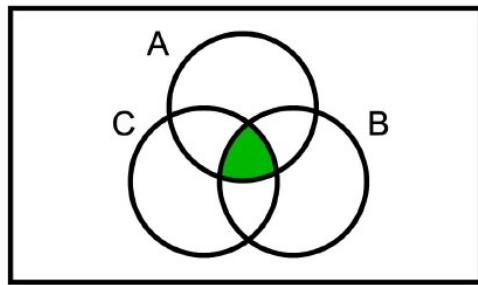
$-(A \cap B)$



$-A$



$-A \cap B$



$A \cap B \cap C$

Крива Пеано

Точки відрізка $A = [0, 1]$ можуть покрити всі точки відрізка $B = [0, 2]$. Наприклад, якщо кожній точці x відрізка A поставити у відповідність точку $2x$ відрізка B .

Крива Пеано, названа на честь Джузеппе Пеано (1858-1932), є загальною назвою так званої кривої заповнення простору.

Крива заповнення простору – крива, що проходить через усі точки простору, зокрема двовимірної площини.

Георг Кантор довів, що множина точок одиничного відрізка дорівнює множині точок одиничного квадрата, тобто кожна точка квадрата зі стороною 1 може бути пов’язана з точкою відрізка з довжиною 1, але не надав метод такого відображення точок. У 1890 році Джузеппе Пеано опублікував метод відображення всіх точок одиничного відрізка на одиничний квадрат (квадрат, сторона якого дорівнює одиниці), так що відображення є безперервним. У своїй статті “Крива, яка заповнює всю площину” (журнал “Математичні аннали”, 1890) Пеано не використовував малюнки. Через рік Давид Гільберт опублікував у тому самому журналі інший варіант побудови такого відображення. Стаття Гільберта була першою, до якої було включено малюнок, який пояснює техніку побудови вищезгаданої кривої. Крива Пеано (або крива Гільберта) є фракталом.

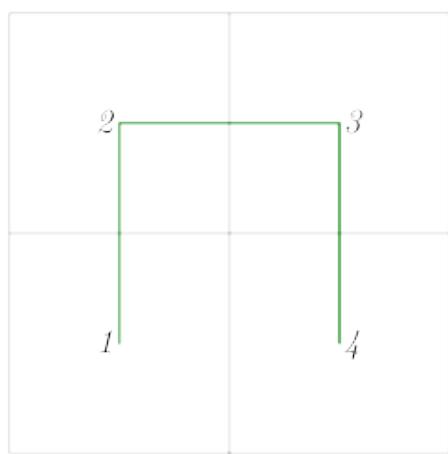


Fig. 1.

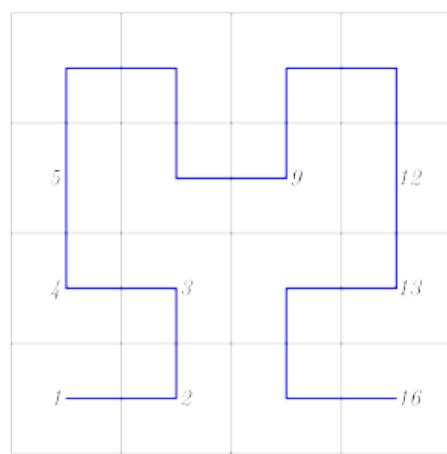


Fig. 2.

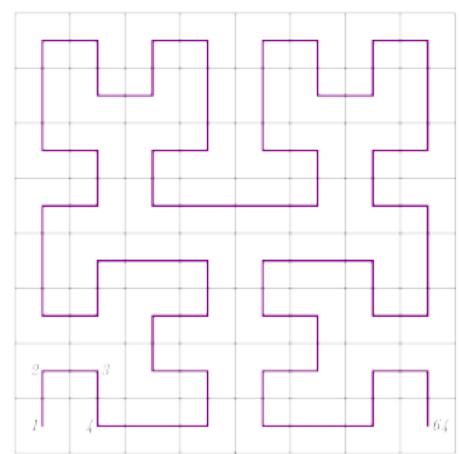
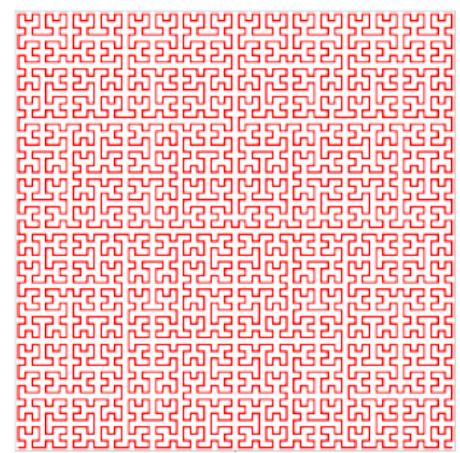
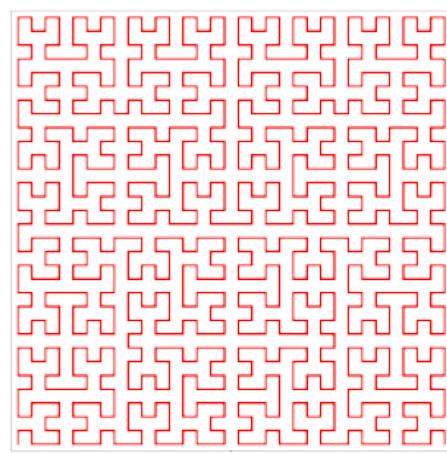
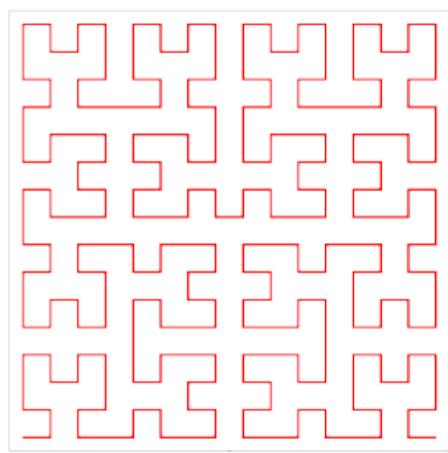
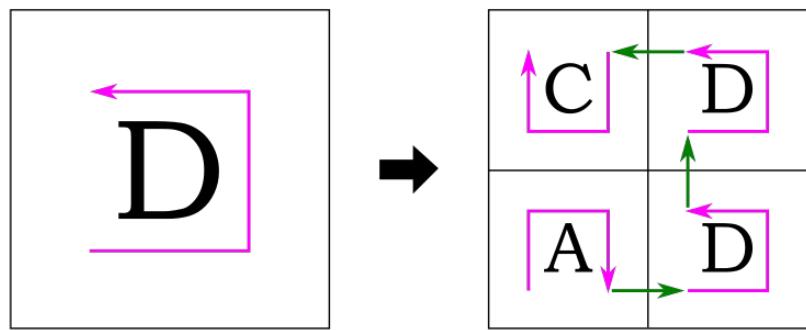
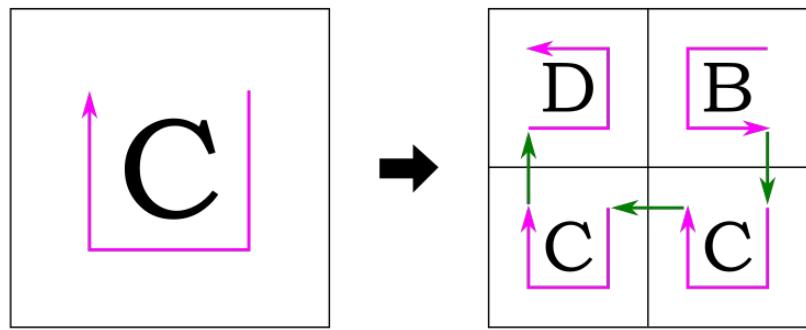
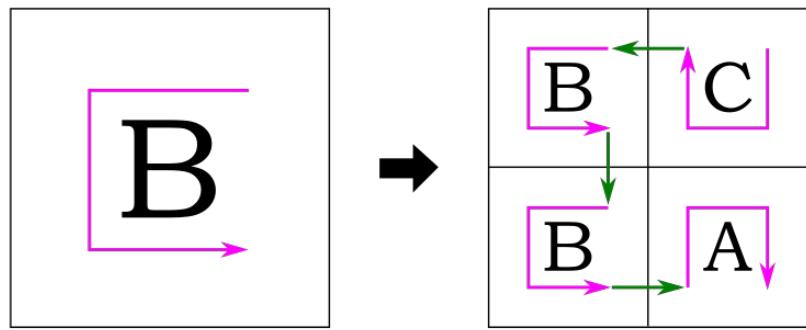
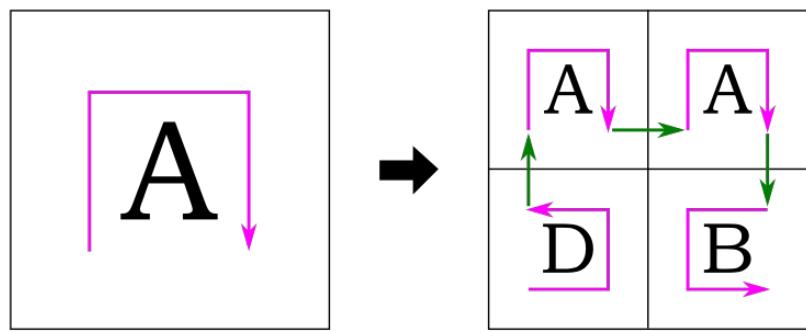


Fig. 3.



Шість ітерацій побудови кривої Пеано за Гільбертом. Ліцензія зображення [CC BY-SA 3.0](#).

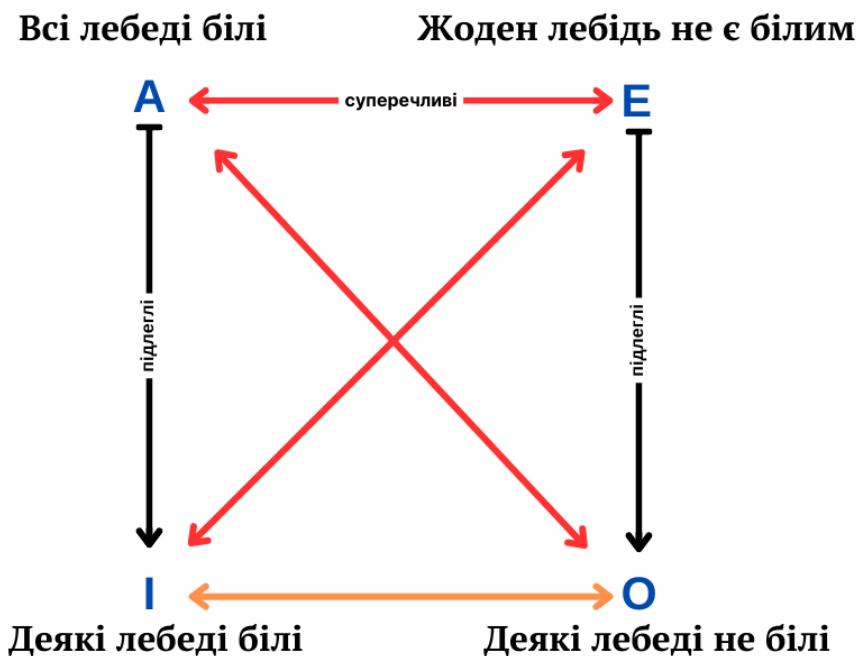


Алгоритм побудови кривої Гільберта (кривої Пеано).

Довжина кривої Пеано наближається до нескінченності, але її всю можна охопити скінченою площею та між двома її кінцями відстань завжди скінчена.

Які найпопулярніші теореми та принципи в логіці?

Логічний квадрат Боеція



Ось деякі поняття з логіки:

1. Абстракція: Абстракція — це процес спрощення складної реальності за допомогою моделювання класів об'єктів або концепцій. Це включає в себе ігнорування неважливих деталей і фокус на суттєвих характеристиках для створення спрощеного представлення.
2. Теорія моделей: Теорія моделей — це галузь математичної логіки, яка займається відношеннями між формальними мовами, їх інтерпретаціями та структурами, які вони представляють. Вона займається вивченням моделей формальних систем та їх властивостей.
- У теорії моделей математичної логіки модель - це структура, що задовольняє певну мову, або, іншими словами, це інтерпретація мови. Наприклад, якщо ми розглядаємо мову арифметики, модель може бути будь-яка структура, що задовольняє аксіоми арифметики, наприклад, множина натуральних чисел разом з операціями додавання та множення.
3. Математична індукція: Математична індукція — це метод доведення, який використовується для встановлення істинності нескінченної послідовності виразів. Зазвичай він включає в себе доведення базового випадку (наприклад, істинність виразу при $n=1$) та показ того, що якщо вираз істинний для певного значення n , то він також істинний для наступного значення ($n+1$). Це доводить істинність виразу для всіх натуральних чисел.

4. Дедукція (вивід): Дедукція — це процес виведення конкретних висновків з більш загальних припущенень або тверджень. Це фундаментальний метод мислення в логіці, часто використовується в математиці.

Наприклад, два менше ніж три ($2 < 3$), бо три, за визначенням, це двійка плюс одиниця, але аксіома каже, що число плюс одиниця більше ніж просто саме число ($x < x+1$).

Ще один приклад дедукції:

Всі справедливі люди добрі.

Всі добрі люди не роблять зла.

Отже, справедливі не роблять зла.

5. Modus Ponens: Modus ponens - це правильна логічна аргументація, яка стверджує, що коли у вас є умовне твердження (Якщо A, то B), і ви знаєте, що A є істинним, то ви можете зробити висновок, що B є істинним. Аргумент має такий вигляд: A вимагає B, A істинне, тому B істинне.

6. Modus Tollens: Modus tollens - це логічне правило, яке стверджує, що коли у вас є умовне твердження (Якщо A, то B), і ви знаєте, що B є хибним, то ви можете зробити висновок, що A є хибним. Аргумент має такий вигляд: A вимагає B, B хибне, тому A хибне.

7. Reductio ad Absurdum (Доведення до абсурду): Reductio ad absurdum - це метод доведення, при якому ви припускаєте протилежне тому, що ви хочете довести, і потім отримуєте суперечність. Ця суперечність показує, що ваше припущення було хибним, і, отже, доводить істинність початкового твердження.

8. Стрілка Пірса: Стрілка Пірса також відома як оператор NOR — була введена Чарлзом Сандерсом Пірсом у 1880—1881 р.р.. Для її позначення використовують символ \downarrow . Це двомісна логічна операція, яка є запереченням диз'юнкції; тому значення "істинно" одержується тільки тоді, коли обидва операнди мають значення "хибно". За допомогою стрілки Пірса (операції NOR) можна виразити будь-яку двомісну (бінарну) логічну операцію.

Стріла Пірса (ввів американський вчений Чарльз Пірс)

A	B	$A \downarrow B$
0	0	1
0	1	0
1	0	0
1	1	0

З допомогою стрілки Пірса можна визначити всі інші оператори.

$$A \downarrow A = \neg A;$$

$$(A \downarrow A) \downarrow (B \downarrow B) = A \wedge B;$$

$$(A \downarrow B) \downarrow (A \downarrow B) = A \vee B;$$

Три основні логічні операції: \vee (або), \wedge (і), \neg (не).

$A \rightarrow B$ те саме, що $\neg A \vee B$.

$A \leftrightarrow B$ те саме, що $(\neg A \wedge \neg B) \vee (A \wedge B)$.

$A \wedge B$ те саме, що $\neg(\neg A \vee \neg B)$;

9. Імплікація (\rightarrow): Імплікація в логіці — це логічний зв'язок між двома твердженнями, який виражає правило: “З правди слідує правда, а з брехні, що завгодно (правда або брехня)”.

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Логіці дають різні визначення, наприклад, логіка — це наука про докази, або логіка — це наука про закони мислення. Також, логіка — це наука, яка вивчає та аналізує твердження (висловлювання) з метою дослідити їх структуру, значення та наслідки. Кажуть, що логіка непереможна, адже її можна перемогти тільки іншою логікою.

“Перехід на особистості” — хибний метод в риториці, коли аргумент направлений не проти самої думки чи позиції, а проти особи, яка її висловлює. Замість того, щоб вести обговорення на рівні ідей чи доказів, людина, що використовує аргумент “Перехід на особистості”, намагається підірвати позицію противника, звертаючись до його особистих характеристик, недоліків або вад. Це може бути використано для відволікання від фактів або справжньої суті обговорення.

Приклади:

1. "Я не вірю тобі щодо цієї проблеми, ти ж завжди робиш помилки",
2. "Цей вчений не вартий уваги, він не має навіть докторського ступеня",
3. "Я не підтримую твою позицію, ти ж ще молодий і не маєш досвіду".

Вчені створили мови програмування, з допомогою яких можна точно та однозначно вказати машині, що потрібно виконати. Вчені також розробили булеву алгебру та логіку предикатів, які дозволяють робити чіткі твердження формальною мовою. Але булева алгебра й логіка предикатів не є мовами людського спілкування, тому що в людській мові є багато іншого, наприклад, імператив (наказові конструкції), причинно-наслідкові зв'язки, час (минулий, теперішній, майбутній). Можна з допомогою хитрих конструкцій намагатись формулювати твердження логікою предикатів про майбутній час, але це важче зробити для минулого часу, і взагалі не зручно. Були розроблені різні формальні системи, які дозволяють врахувати всі конструкції мови, зокрема, Модальна логіка (імперативна логіка, темпоральна логіка, деонтична логіка). Амір Пнуелі отримав премію Тюрінга за основоположну працю, що ввела поняття темпоральної логіки у комп'ютерні науки і видатний внесок у верифікацію програм та систем. Леслі Лемпорт також відомий своєю роботою з темпоральної логіки, де він представив часову логіку дій (TLA).

Теорема Гьоделя про повноту

Теорема Геделя про повноту — твердження про повноту класичного числення предикатів, доведене Куртом Геделем в 1930 році. Якщо предикатна формула істинна в будь-якій інтерпретації, то її можна вивести в численні предикатів.

У 1929 році Курт Гьодель довів теорему про повноту (логіки першого порядку), згідно з якою кожна тавтологія, тобто істинна формула для будь-якої інтерпретації, сформульована мовою логіки першого порядку (логіки предикатів), може бути виведена (доведена) у логіці першого порядку. У 1930 році Гьодель захистив дисертацію на тему “Про повноту логічного обчислення”.

Теорема Гьоделя про неповноту

Рассел та Вайтгед опублікували три томи з 1910 по 1913 рік під тією ж назвою “Математичні принципи”, в яких вони розробили систему аксіом для теорії множин, логіки предикатів та арифметики. Система, запропонована Вайтгедом і Расселом, усунула всі відомі на той час парадокси та, як писав Курт Гедель (Гьодель) в 1931 році, створила враження, що будь-яке твердження математики можна довести з допомогою цієї системи, доки він не довів протилежне.

Курт Гьодель (Гедель) у 1929 році у своїй дисертації довів, що логіка предикатів (першого порядку) є повною, тобто будь-яка тавтологія (формула, яка завжди вірна), сформульована її мовою, може бути виведена з аксіом логіки предикатів (першого порядку). Крім того, Курту Гьоделю, який знав про 2 проблему Гільберта щодо узгодженості арифметики та вивчав лінгвістичний парадокс брехуна (Брехун сказав, що він бреше. Він збрехав?), який ґрунтуються на порушенні правила автореференції (самопосилання), вдалося довести, що конструктивні методи Гільберта, засновані на рекурсивних функціях, непридатні для вирішення проблеми неузгодженості арифметики. Курт Гьодель довів, що кожна формальна теорія, в якій може бути розроблена теорія натуральних чисел, має правильне, але недоведене твердження, точніше, кожна така система є неповною, тобто не всі твердження, сформульовані її мовою, є доказовими з допомогою її аксіом, інакше, теорія є суперечливою, тобто можна буде вивести певне твердження, а також його заперечення, а це означає, що будь-яке можливе твердження може бути виведено, і, таким чином, теорія втрачає будь-який сенс. Гьодель довів свої відкриття в першій і другій теоремах про неповноту і прочитав їх без оголошення на науковому конгресі 1930 року в Кенігсберзі, що зробило його відомим математиком. Своїми теоремами Гьодель показав, що конструктивні методи, засновані на рекурсивно обчислюваних функціях, а отже, і на комп’ютері, не придатні для розв'язання проблеми узгодженості арифметики за методом Давида Гільберта.

Дві теореми про неповноту Гьоделя (Геделя) можна підсумувати таким чином: перша теорема Геделя про неповноту “Для будь-якого формалізму F , що задовольняє умови a) і b), що містить універсальний квантор і, крім того, омега-несуперечливий, можна вказати формулу без вільних змінних так, що ні вона сама по собі, ані її заперечення не будуть виведені в F ”.

a) Формалізм F містить терміни та формули рекурсивної арифметики. Кожна формула без змінних формули, виведена в рекурсивній арифметиці, виводиться в F , і кожен перехід від однієї формули від F до іншої, який можна виконати з допомогою рекурсивної арифметики, також виконується в F .

b) Існує біективна нумерація (один до одного) виразів формалізму F натуральними числами. Твердження “число m — це номер певної послідовності виразів із F , що є виведенням (доведенням) виразу з номером n ” можна визначити рекурсивною функцією $B(m, n)$ у формалізмі F .

Як показав американський математик Барклі Россер (1907 – 1989), припущення про омега-несуперечливість F можна виключити з формуллювання цієї теореми, тобто замість омега-

несуперечливості F ми, як і раніше, можемо говорити про несуперечливість F у звичайному розумінні цього слова, але для цього необхідно певним чином модифікувати доказ Гьоделя.

Формалізм F називається омега-несуперечливим (омега-послідовним), якщо в ньому завжди вірна наступна схема: $\forall x_1, \dots, x_n A(x_1, \dots, x_n) \rightarrow \neg(\neg \forall x_1, \dots, x_n A(x_1, \dots, x_n))$; Тобто, якщо формула з універсальним квантором істинна, то її заперечення (також з універсальним квантором) є хибним.

Формалізм F називається (просто) несуперечливим, якщо правильна наступна схема: $A(x_1, \dots, x_n) \rightarrow \neg(\neg A(x_1, \dots, x_n))$. Тобто якщо формула правильна, то її заперечення неправильне.

Теорема. Омега-несуперечлива система завжди просто несуперечлива (когерентна, послідовна). Однак несуперечлива система може бути омега-суперечливою, якщо виключити логічні квантори. Тобто, якщо істинність будь-якої формули без кванторів передбачає хибність її заперечення, то система все ж може бути омега-суперечливою, оскільки ми виключили квантори з умови.

Про доведення першої теореми Гьоделя про неповноту

Щоб зрозуміти доказ першої теореми Гьоделя про неповноту, спочатку опишемо основну ідею та принцип цього доведення.

--- Ми хочемо створити конструктивний алгоритм (або можна сказати програму), який брав би як входні дані рядки символів мови арифметики першого порядку, тобто елементарні арифметичні символи та символи логіки предикатів, і у відповідь давав результат про те, чи можна вивести цю формулу з аксіом арифметики (Пеано) з логікою (Рассела-Вайтгеда) чи ні. Виводимість у цьому випадку дорівнює доказовості формули, звідси її істинність.

--- Конструктивний алгоритм потрібно будувати не абияк, а з допомогою рекурсивних функцій, які, як довели С. Кліні та А. Черч, є обчислювальними функціями.

--- Рекурсивні функції — це дискретні функції над натуральними числами. Як вони працюватимуть з алфавітом арифметики першого порядку, що включає, крім чисел, логічні символи, символи операторів? Відповідь — нумерація по Гьоделю.

--- Курт Гьодель запропонував кодувати символи та рядки символів арифметики першого порядку, використовуючи запропонований ним метод нумерації по Гьоделю. Після кодування логічні символи перетворюються на рядки чисел, тому їх можна обробляти рекурсивними функціями.

--- Припустимо, що ми зібрали алгоритм із використанням рекурсивних функцій, щоб він міг визначити, чи можна вивести дану формулу (рядок символів, закодованих у вигляді чисел) чи ні. Оскільки теорія рекурсивних функцій досить елементарна, її можна розгорнути (розвинути) в межах арифметики першого порядку, тому алфавіту арифметики першого порядку достатньо, щоб довести або спростувати твердження про рекурсивні функції. (Арифметика першого порядку — це математична теорія, достатня для доведення теорем про натуральні числа, містить логіку предикатів і основні арифметичні оператори, а також функції, визначені з допомогою цих операторів). Ми отримуємо цікаву ситуацію, а саме, ми хочемо з допомогою рекурсивних функцій визначити, чи є та чи інша формула доказова в арифметиці першого порядку, в тому числі чи є доказові результати нашого конструктивного алгоритму на основі тих самих рекурсивних функцій, оскільки вона можна описати в арифметиці першого порядку. Не дивно, що це загрожує самопосиланням і парадоксами. Першим цю ситуацію помітив Курт Гьодель.

--- Припустимо, що ми створили алгоритми на комп'ютері, який отримує натуральне число як входні дані, що є не що інше, як закодований оператор мовою логіки предикатів, і повертає результат про те, чи можна це твердження довести чи ні. Уявіть, що АЛГО є справжньою обчислювальною машиною, ця машина приймає число як вход, позначимо його як АЛГО(число). Ми отримаємо протиріччя, якщо запустимо програму АЛГО(число), де число — це закодований набір символів, що по суті означає: “АЛГО(число) не можна довести”. Якщо реальний АЛГО(число) повертає “правда”, то АЛГО(число), який описаний у виразі, повинен повернати “брехня”, і навпаки. Ми отримуємо протиріччя, оскільки результати реальних алгоритмів і те, що в записі повинні збігатися.

Для побудови цього алгоритму Гьодель визначає приблизно 40 функцій. Ми їх не дамо, програмісти інтуїтивно розуміють, що це можливо, тобто запрограмувати такий алгоритм можна.

Схема доказу

Нумерація Гедолья:

Зіставимо символи нашої формальної мови із простими числами, включаючи одиницю.

1 - “0”,

3 - “f”, де f означає $f(1) = 2$. $f(f(1)) = 3$.

5 - “ \neg ”,

7 - “ \vee ”,

9 - “ \forall ”,

11 - “(“,

13 - “)”.

Імплікація $A \rightarrow B$ є скороченням виразу $\neg A \vee B$.

$x = y$ можна представити у вигляді $\forall a(a(x) \rightarrow a(y))$;

Припустимо, ви склали формулу $2 = 2$. використовуючи методи нашої системи, ви можете виразити її так:

$\forall a(a(ff(0)) \rightarrow a(ff(0)))$;

Ці символи будемо називати основними, вони займають всі числа до 13.

Далі в нашій формальній системі будуть змінні для натуральних чисел, назовемо їх змінними першого типу. Тобто x_1, y_1, z_1 .

Ми будемо використовувати змінні другого типу для формул, що складаються з базових символів і змінних першого типу. (Тобто для символьних рядків).

Ми будемо використовувати змінні другого типу для наборів формул, тобто наборів змінних другого типу.

Усім змінним для натуральних чисел ми будемо пов'язувати прості числа, починаючи з 13, оскільки до 13 вони вже зайняті основними символами.

До ланцюжків символів, тобто змінних другого типу, ми пов'язуємо прості числа, починаючи з 1, але другого степеня. До змінних третього типу ми пов'язуємо прості числа третього степеня.

Комбінації знаків виду $a(b)$, де b — ознака n -го, і a — ознака $(n+1)$ -го типу, ми називаемо елементарними формулами. Клас формул ми визначаємо як найменший клас, що містить усі елементарні формулі, а також, поряд з будь-якими a та b наступне: $\neg(a)$, $(a) \vee (b)$, $\forall x(a)$, де x — будь-який задана змінна.

Формулу $f(0)$ можна закодувати як $3^1 \cdot 11^2 \cdot 1^3 \cdot 13^4$. Де, “3-f, 11-(, 1-0, 13-)”;

Щоб відновити вихідну формулу за її числом Гьоделя, потрібно розкласти це число на прості множники. Розкладанням натурального числа на множники називають його розкладання в добуток простих множників. Існування та унікальність (до порядку множників) такого розкладання випливає з основної теореми арифметики.

Натуральне число n називається дільником цілого числа m , якщо для відповідного цілого k виконується рівність $m = n * k$. Просте число — це натуральне число $p \geq 2$, яке ділиться тільки на себе і на одиницю. Складене число — це число, яке має понад два дільники.

Як виконати розкладку цілого числа на прості множники?

1. Візьміть найменше просте число 2 і перевірте, чи ділиться вихідне число на 2 за критерієм подільності чи за звичайним діленням.

2. Якщо ділиться, то запишіть у праву колонку 2. Далі вихідне число поділіть на 2 і запишіть результат у ліву колонку під вихідним числом. Якщо не ділиться, то беремо наступне просте число, тобто 3.

4. Повторіть ці дії, працюючи з останнім числом у лівій колонці та з поточним простим числом. Розкладання закінчується, коли в лівій колонці записується число 1.

Приклад: $84/2 = 42$, $42/2 = 21$, $21/3 = 7$, $7/7 = 1$. Отже, $84 = 2 * 2 * 3 * 7$.

Визначення

$B(n)$ — означає: формула з номером Гьоделя n виводиться (доводиться). $B(n)$ повертає істину чи хибність, тобто 1 або 0.

$B^r(n)$ — ідентичний $B(n)$, але сформульований з допомогою консервативної теорії.

$G(x)$ — повертає число Гьоделя x ;

$G'(x)$ — обернено до $G(x)$, означає: повернути формулу числа x .

$S(x, y)$ — підстановка, повертає число Гьоделя формули x після підстановки y на місце її вільних змінних.

Доказ

$$q = G[\neg B(S(m, n))];$$

q — означає число Гьоделя “ $\neg B(S(m, n))$ ”.

$$p = S(q, q) = G[\neg B(S(q, q))];$$

$$\neg B(p) = \neg B(G[\neg B(p)]);$$

$$\neg B^r(p) \rightarrow \neg B(p);$$

Тоді

$$\neg B^r(p) \rightarrow \neg B(p) \rightarrow B(p);$$

$$B^r(p) \rightarrow B(p) \rightarrow \neg B(p);$$

Друга теорема про неповноту Гьоделя

У послідовному формалізмі F , що задовільняє умовам a) і b), не можна вивести формулу $(*)$, яка є формалізацією твердження про узгодженість F . (Умови a і b такі ж, як і в першій теоремі про неповноту).

$$(*) \exists x B(x) \rightarrow \neg \exists x B(e(x))$$

Число заперечення формули з номером n зображується в її залежності від n деякою рекурсивною функцією $e(n)$.

$$n = G[A];$$

$$e(n) = G[\neg A];$$

Теорема Тарського про невизначеність

Теорема Тарського про невизначеність — це теорема, доведена Альфредом Тарським у 1936 році, важливий обмежувальний результат математичної логіки. Теорема Тарського про невизначеність говорить: "Множина істинних формул арифметики першого порядку (тобто множина їх чисел для будь-якої фіксованої нумерації Гьоделя) не є арифметичною множиною. Іншими словами, поняття арифметичної істини не може бути виражено з допомогою самої арифметики".

Арифметична множина — це множина натуральних чисел S , яка може бути визначена з допомогою формули мовою арифметики першого порядку, тобто якщо існує формула $\phi(x)$ з однією вільною змінною x такою, що $\forall x (x \in S \leftrightarrow \phi(x))$.

Множина називається неарифметичною, якщо мовою арифметики першого порядку (арифметика Пеано) немає набору формул, для якої вона була б моделлю. Простіше кажучи, не існує остаточної формулі, яка б визначала набір усіх істинних формул в арифметиці.

Теорема Черча

Теорема Черча про нерозв'язність логіки предикатів:

Не існує універсального алгоритму (процедури) для визначення істинності чи хибності формули обчислення предикатів.

Доказ. У теоремі про зупинку машини Тюрінга ми показали, що можна створити таке твердження, яке не може бути розв'язане, таке твердження також можна сформувати в рамках логіки предикатів.

Теорема американського математика та філософа Алонзо Черча (1903–95) стверджує, що теореми обчислення предикатів не утворюють загального рекурсивного набору. Це означає, що не існує процедури прийняття рішення чи алгоритму для визначення того, чи є довільна формула обчислення предикатів першого порядку теоремою.

Які найпопулярніші алгоритми в алгебрі?

Теорема Абеля — Руффіні

Теорема Абеля — Руффіні стверджує, що загальне рівняння п'ятого та вищого степеня є нерозв'язним в радикалах (для коренів многочлена не існує формул, що використовують чотири арифметичні дії та корені довільного степеня).

Основна теорема алгебри доводить, що рівняння n-го степеня має n комплексних коренів, хоча над іншими полями коренів може і не існувати.

Загальну відповідь про наявність коренів многочлена над заданим полем та розв'язність над цим полем дає теорія Галуа.

Корінь рівняння — це значення невідомого, яке перетворює рівняння на правильну рівність.

Кубічне рівняння в загальній формі: $ax^3 + bx^2 + cx + d = 0$ (де a не дорівнює нуль).

Загальне квадратне рівняння: $ax^2 + bx + c = 0$, коли $a > 0$, має два корені:
 $x = (-b + \sqrt{b^2 - 4ac})/2a$, або $x = (-b - \sqrt{b^2 - 4ac})/2a$.

Вираз $b^2 - 4ac$, називається дискримінантом квадратного рівняння.

Вивід формули для квадратного рівняння:

$$ax^2 + bx + c = 0,$$

$$ax^2 + bx = -c,$$

$$x^2 + (b/a)x = -c/a = x(x + b/a),$$

$$(2x + b/a)^2 = (b/a)^2 + 4(-c/a),$$

$$2x + (b/a) = \pm \sqrt{(b/a)^2 + 4(-c/a)},$$

$$2x = -(b/a) \pm \sqrt{(b/a)^2 + 4(-c/a)},$$

$$x = \frac{-(b/a) \pm \sqrt{(b/a)^2 + 4(-c/a)}}{2},$$

$$x = \frac{(-b \pm \sqrt{b^2 - 4ac})}{2a}.$$

Формула Кардано

Розв'язування кубічних рівнянь за формулою Кардано.

Для кубічного рівняння вигляду:

$$ax^3 + bx^2 + cx + d = 0,$$

знаходять значення:

$$r_1 = b/a, r_2 = c/a, r_3 = d/a.$$

Далі знаходимо:

$$p = -(r_1^2/3) + r_2,$$

$$q = 2r_1^3/27 - (r_1r_2/3) + r_3.$$

Підставте отримані p і q у формулу Кардано:

$$y = \text{root}(-(q/2) + \sqrt{q^2/4 + p^3/27}, 3) + \text{root}(-(q/2) + \sqrt{q^2/4 + p^3/27}, 3);$$

В результаті знаходимо корені вихідного рівняння за формулою: $x = y - r_1/3$.

Для кубічного рівняння можливі два випадки: 1. Рівняння має один дійсний і два комплексних корені.
2. Рівняння має три дійсні корені.

Кардано зробив значний внесок у розвиток алгебри.

У 1545 році Джироламо Кардано в книзі "Велике мистецтво" для системи рівнянь:

$$x + y = 10,$$

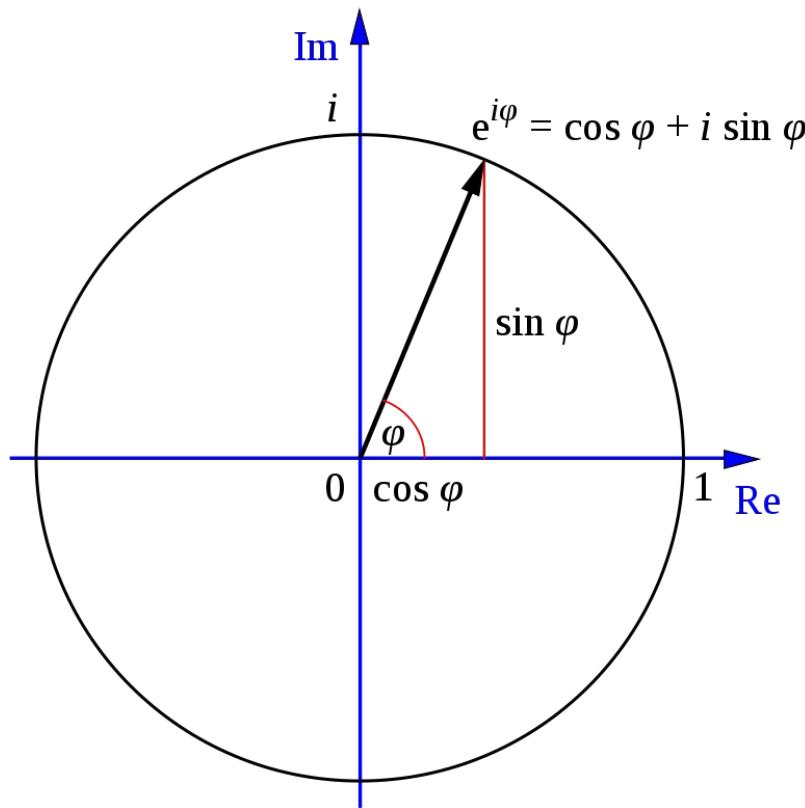
$$x * y = 40,$$

надав два корені: $5 + (\sqrt{-15})$ та $5 - (\sqrt{-15})$.

Немає кореня з від'ємного числа, тому Кардано ввів "уявні числа" (ще не комплексні числа).

В 18 століття в праці "Елементи алгебри" Леонард Ейлер писав: "Отже, коли потрібно витягти корінь з від'ємного числа, виникають труднощі, оскільки немає числа, квадрат якого був би від'ємною величиною. Припустимо, наприклад, що ми хочемо витягти корінь з -4 , тут ми вимагаємо такого числа, яке, помножене на себе, дало $6 - 4$: це число не є ні $+2$, ні -2 , тому що квадрат як $+2$, так і -2 дорівнює $+4$. Таким чином ми наводимо себе на ідею чисел, які за своєю природою неможливі й тому їх зазвичай називають уявними величинами, оскільки вони існують лише в уяві. Але попри це, ці числа постають перед розумом, тобто вони існують у нашій уяві, і ми ще маємо достатнє уявлення про них; оскільки ми знаємо, що під $\sqrt{-4}$ розуміється число, яке, помножене на себе, дає -4 . З цієї причини також ніщо не заважає нам використовувати ці уявні числа для розрахунків".

Комплексні числа у геометричному вигляді використовував Леонард Ейлер, наприклад, у 1748 році у своїй знаменитій книзі "Введення в аналіз нескінченних" він виводить так звану формулу Ейлера.



Формула Ейлера

$e^{i\pi} = -1$, де e – число Ейлера, i – уявна одиниця, π – Пі.

$$i^2 = -1.$$

$$\pi = 3,1415926535\dots$$

$$e = \lim(x \rightarrow \infty) (1 + (1/x))^x.$$

$$e = 2,7182818284\dots$$

$$e^n = \lim(x \rightarrow \infty) (1 + (n/x))^x.$$

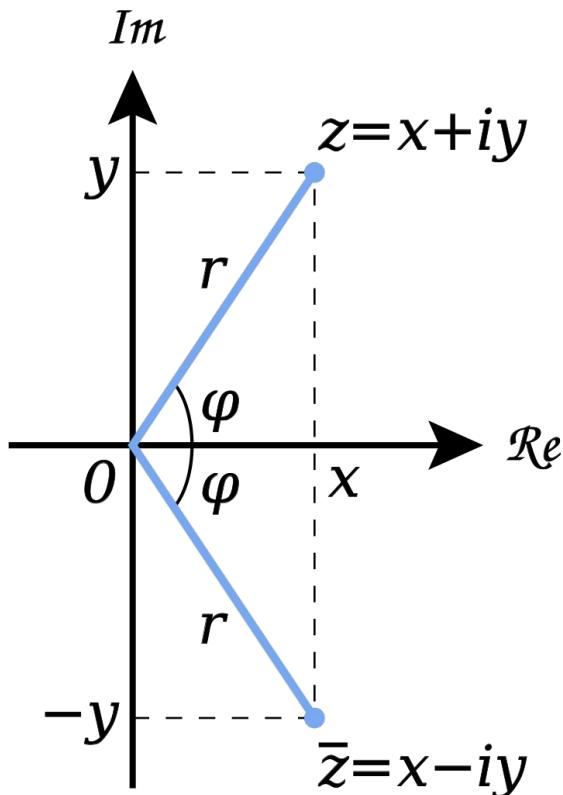
$$e^{i\varphi} = \lim(x \rightarrow \infty) (1 + (i\varphi/x))^x.$$

$$e^{i\pi} = \cos(\pi) + i * \sin(\pi) = -1.$$

Ейлер розробив цю формули ($e^{i\pi} = -1$) аналізуючи ряди Тейлора для числа e та синуса й косинуса.

Жан-Робер Арган (Арганд) (1768 – 1822) – французький математик. У 1806 році, керуючи книжковим магазином у Парижі, він опублікував ідею геометричної інтерпретації комплексних чисел (хоча її використовував вже Ейлер, але не зобразив), відому як діаграма Арганда. Жан-Робер Арган також відомий першим суворим доказом фундаментальної теореми алгебри (Основна теорема алгебри).

Ми можемо представити комплексне число $(a + bi)$ геометрично.



Запишемо комплексне число z у тригонометричній формі:

$$z = r(\cos(\varphi) + i \sin(\varphi)).$$

Тоді n корені з z визначаються за формулою Де Муавра (тригонометрична форма):

$$\text{cmplxroot}(z, n) = \text{root}(r, n) * (\cos((\varphi + 2\pi k)/n) + i \sin((\varphi + 2\pi k)/n)), k = 0, 1, \dots, n-1.$$

Де:

$z = x + iy$, (комплексне число),

$x = \text{Re}(z) \in \mathbb{R}$, (дійсна частина комплексного числа),

$y = \text{Im}(z) \in \mathbb{R}$, (уявна частина комплексного числа),

i — уявна одиниця, квадрат якої дорівнює -1 .

$r = |z| = \sqrt{x^2 + y^2}$, (абсолютне значення комплексного числа),

$\varphi = \arg(z) = \arctg(y/x)$, (аргумент комплексного числа).

Основна теорема алгебри

Поліном — це алгебраїчний вираз, який складається з суми або різниці декількох мономів. Зазвичай поліноми мають наступний вигляд:

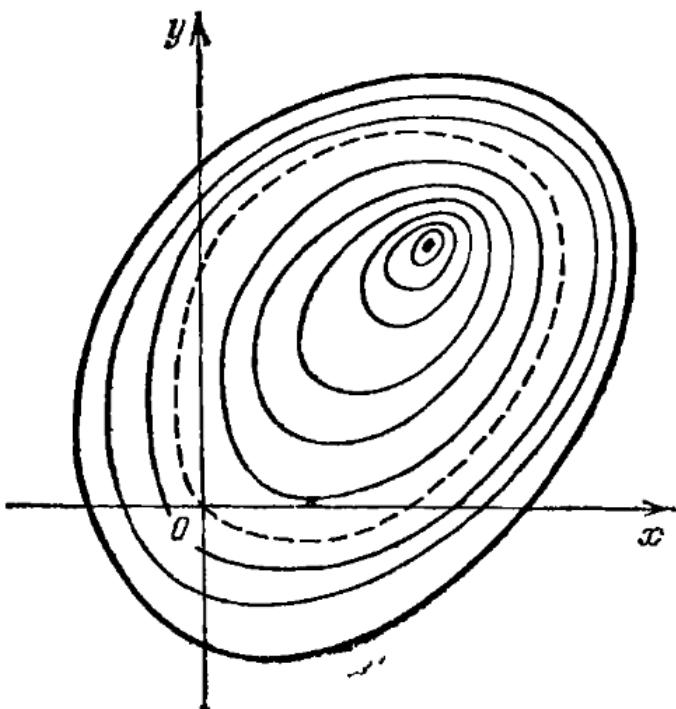
$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0.$$

Основна теорема алгебри

Будь-який поліном додатного степеня (тобто ≥ 1) з комплексними коефіцієнтами має принаймні один комплексний корінь.

Доведення (геометричне) від протиріччя.

Нехай поліном $f(z) = z^n + a_{n-1} z^{n-1} + \dots + a_0$ ступеня $n \geq 1$ не має комплексного кореня, тобто $f(z)$ для будь-якого z не дорівнює 0. Нехай z рухається рівномірно (у декартовій системі координат) по колу радіуса R з центром у нулі (точка О). Оскільки z — комплексне число, то переміщення по колу означає, що дві складові комплексного числа змінюються таким чином, що змінна z проходить через усі точки певного кола, кожна точка якого також задається двома числами. (Якщо число дійсне, то це комплексне число з другим компонентом 0, отже, в декартовій системі координат воно знаходитьться вздовж осі x , для $y = 0$.) Якщо цю точку z підставити в $f(z)$, то згідно з припущенням $f(z)$ не буде приймати значення нуль для всього проходження z по певному колу. У цьому випадку точка z^n буде рухатися по колу радіуса R^n з кутовою швидкістю (використовуємо поняття з фізики) в n разів більшою за кутову швидкість точки z . Це випливає із представлення z у вигляді: якщо $z = R(\cos(\phi) + \sin(\phi) i)$, то $z^n = R^n (\cos(n\phi) + \sin(n\phi) i)$, де ϕ — аргумент комплексного числа, то кут між віссю x і вектором від точки О до точки числа. Таким чином, коли z робить один повний оберт, точка z^n робить n обертів навколо нуля. Нехай $b(z) = a_{n-1} z^{n-1} + \dots + a_0$, тобто $b(z)$ частина вихідного полінома $f(z)$. Для великого R число $b(z)$ є незначним у порівнянні з z^n , тому переміщення точки $f(z) = z^n + b(z)$ буде відносно не відрізнити від руху точки z^n . Отже, точка $f(z)$ робить однакову кількість обертів навколо нуля, таку ж кількість z^n , тобто n обертів. Якщо R близький до нуля, то $f(z)$ близький до свого останнього члена a_0 (у нашому випадку — точки a_0). Коли z робить повний оберт навколо кола малого радіуса, $f(z)$ описує замкнуту криву поблизу a_0 . Оскільки a_0 не дорівнює 0 (інакше розв'язок тривіальний, просто корінь z в такому випадку дорівнює 0), виходить, що для великих R (радіусів) точка $f(z)$ робить 0 обертів навколо нуля. Нехай $r(R)$ позначає кількість обертів точки $f(z)$, коли змінна z проходить по всіх точках кола з радіусом R . Визначимо $r(R)$ як кількість обертів вектора, що проходить з точки О до точки $f(z)$, коли z робить повний оберт по колу радіусом R . Ми знаємо, що для достатньо великого R функція $r(R)$ прийматиме значення n , а для достатньо малого R функція $r(R)$ прийматиме значення 0, а $f(z)$ буде поблизу точки a_0 . Таким чином, для даних R і z поліном $f(z)$ повинен стати нульовим (пройти через початок координат О). Тому перше припущення про відсутність коренів не відповідає дійсності.



В 9 столітті жив відомий математик Мухаммад Аль-Хорезмі.

Мухаммад аль-Хорезмі (780 – 850) — перський математик і астроном, який жив на території сучасних Узбекистану та Іраку. Аль-Хорезмі вивчав праці грецьких та індійських математиків, спираючись на них, створив власні праці.

Відомою роботою Аль-Хорезмі є "Книга алгебри й алмукабали", яка присвячена рівнянням і від назви якої пішла назва науки "алгебра", що означає "ал-джабр" — завершення (доповнення). Слово "алмукабала" означає "протиставлення". Це тому що в рівняннях потрібно було доповнювати значення і записувати рівняння, тобто протиставляти значення.

"Книга алгебри та алмукабали" перекладена з арабської на латинь Робертом Честерським у 1145 році. Від латинізованого ім'я цього вченого (Аль-Хорезмі) походить слово "алгоритм", що означає певний набір дій для розв'язання конкретної задачі. Також завдяки його роботам до європейських мов потрапили слова шифр і цифра, які походять від арабського слова "сіфр", яке означає "нуль".

Аль-Хорезмі написав "Книгу про індійські числа". Арабський текст було втрачено, але його латинський переклад XII століття зберігся. Трактат розповідає про числа, які використовували індуси (індійці), і визначає дії над ними.

Слово "алгоритм" вживає Іоанн де Сакробоско (1195 – 1256) у своїй праці "Мистецтво нумерації" (надрукована в 1692). Цим словом Сакробоско називає науку, яку виклав Аль-Хорезмі у своєму трактаті про числа. У 1240 році французький монах Олександр із Вільдье (1175 — 1240) пише латинський текст під назвою "Кармен де Алгорісмо", на початку якого сказано, що "алгорісмус" означає мистецтво роботи з індійськими числами. Слово "алгорісмус" походить від латинізованої версії ім'я Аль-Хорезмі.

Французький математик Жан Лерон д'Аламбер у відомій енциклопедії 18 століття пише: "Алгоритм — арабський термін, який використовується деякими авторами, зокрема іспанцями, для позначення практики алгебри. Його також іноді використовують для арифметики, тобто операцій з числами. Алгоритм, відповідно до сили слова, насправді означає мистецтво обчислення з точністю та

легкістю". Вже з 19 століття слово алгоритм набирає значення чіткого набору кроків для отримання певного результату, зокрема результату обчислення. Англійський математик Чарльз Хаттон (1737 – 1823) в своєму математичному словнику (1795) писав: "Алгоритм — загальні правила обчислення в будь-якому мистецтві. Алгоритм також означає загальні правила для виконання операцій арифметики, або алгебри".

Зараз, алгоритм — це точний припис про порядок виконання певної системи операцій над вихідними даними для отримання бажаного результату.

Рівняння Аль-Хорезмі.

Для рівнянь такого типу $ax^2 + bx = c$ Аль-Хорезмі дає наступний метод розв'язку (алгоритм). Якщо ми хочемо знайти x в $ax^2 + bx = c$, то виконаємо:

1. Поділіть число b на два.

$$b_1 = b / 2.$$

2. Помножте b_1 саме на себе.

$$b_2 = b_1 * b_1.$$

3. Додаємо b_2 до c , пишемо як d .

$$d = b_2 + c.$$

4. Беремо корінь d , пишемо як e .

$$\sqrt{d} = e.$$

5. Від e відняти b_1 , і отримаємо x .

$$x = e - b_1.$$

Приклади:

$$1. x^2 + 10x = 39.$$

$$5 = 10 / 2.$$

$$25 = 5 * 5.$$

$$64 = 25 + 39.$$

$$\sqrt{64} = 8.$$

$$x = 8 - 5.$$

$$x = 3.$$

Ця формула заснована на схемі:

$$x^2 + 10x = 39$$
$$39 = \boxed{x^2} \boxed{10x} = \boxed{x^2} \boxed{5 * x}$$
$$\boxed{x^2} \boxed{5 * x} = 64$$
$$\boxed{x^2} \boxed{5 * x} = 25$$

$$2. 2x^2 + 10x = 48.$$

Спростіть рівняння

$$(2x^2/2 + 10x/2 = 48/2), \text{ тобто } x^2 + 5x = 24.$$

$$x^2 + 5x = 24$$

Поділіть число 5 на 2.

$$2,5 = 5 / 2.$$

Помножте 2,5 саме на себе.

$$6,25 = 2,5 * 2,5.$$

Додайте 6,25 до 24.

$$30,25 = 6,25 + 24.$$

Беремо корінь з числа 30,25.

$$\sqrt{30,25} = 5,5.$$

Від 5,5 віднімемо 2,5 і отримаємо x.

$$x = 5,5 - 2,5.$$

$$x = 3.$$

Правило Крамера

У лінійній алгебрі правило Крамера являє собою явну формулу для розв'язку системи лінійних рівнянь з кількістю рівнянь рівній кількості невідомих, яка справедлива, коли система має єдиний розв'язок. Правило назване на честь женевського математика Габріеля Крамера (1704–1752), який опублікував правило для довільної кількості невідомих у 1750 році.

Система двох лінійних рівнянь вигляду:

- 1) $ax + by = f,$
- 2) $cx + dy = g.$

Корені можна знайти за формулами:

$$x = (df - bg) / (ad - cb),$$
$$y = (ag - cf) / (ad - cb).$$

Для складніших систем лінійних рівнянь можна використовувати метод Гаусса, який оснований на елементарних перетвореннях.

Система рівнянь — це скінчений набір рівнянь, для яких шукаються спільні розв'язки.

Розв'яжіть цю систему методом Гаусса:

$$x + y = 3;$$

$$3x - 2y = 4;$$

Множення першого рівняння на -3 і додавання результату до другого рівняння усуває змінну x :

$$-3x - 3y = -9;$$

$$3x - 2y = 4;$$

$$-5y = -5;$$

Це останнє рівняння, $-5y = -5$, одразу означає $y = 1$. Зворотна підстановка $y = 1$ в вихідне перше рівняння, $x + y = 3$, дає $x = 2$.

Правило Крамера має геометричне тлумачення, яке також можна вважати доказом.

Дано систему рівнянь:

$$ax + by = f,$$

$$cx + dy = g.$$

Її можна розглядати як рівняння між векторами:

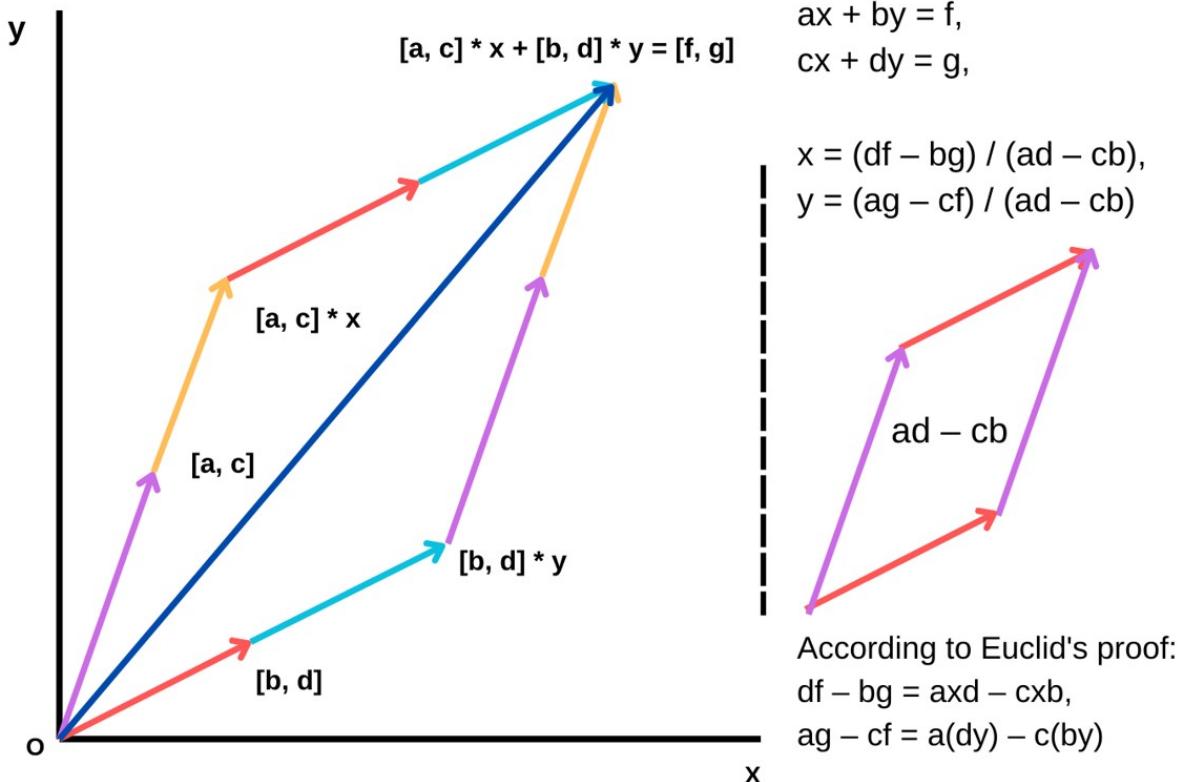
$$x * \langle a, c \rangle + y * \langle b, d \rangle = \langle f, g \rangle;$$

Корені можна знайти за формулами:

$$x = (df - bg) / (ad - cb),$$

$$y = (ag - cf) / (ad - cb).$$

Геометрична інтерпретація правила Крамера.

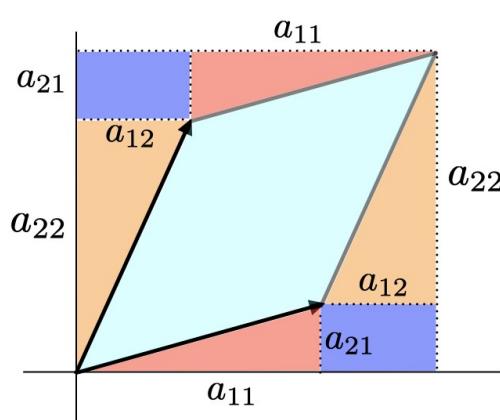
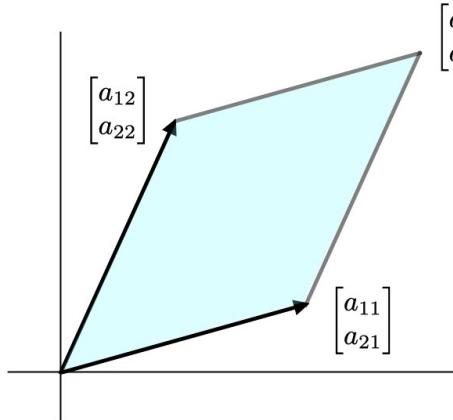


Площа паралелограма, визначена а і b, задається визначником (детермінантом) системи рівнянь:

$$a_{11} * a_{22} - a_{12} * a_{21};$$

The determinant

Let's compute the area of the parallelogram spanned by the columns of a matrix $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$



$$\begin{aligned} \text{area of parallelogram} &= (a_{11} + a_{12})(a_{21} + a_{22}) - a_{12}a_{22} - a_{11}a_{21} - 2a_{21}a_{12} \\ &= \cancel{a_{11}a_{21}} + a_{11}a_{22} + a_{12}a_{21} + \cancel{a_{12}a_{22}} - \cancel{a_{12}a_{22}} - \cancel{a_{11}a_{21}} - 2a_{21}a_{12} \\ &= a_{11}a_{22} - a_{21}a_{12} \end{aligned}$$

This number $a_{11}a_{22} - a_{21}a_{12}$ is called the “determinant” of A

Author of image (of determinant): Daniel O'Connor. License: [CC BY-SA 4.0](#)

Алгоритм Штрассена для множення матриць

Звичайний алгоритм множення матриць має кубічну складність $O(n^3)$, але алгоритм Штрассена працює з меншою складністю $O(n^{2.81})$, що робить його ефективнішим для великих матриць. Ось загальний шаблон алгоритму Штрассена у псевдокоді:

Штрассен_Множення_Матриць(A, B):

якщо розмір(A) == 1:

$C = \text{нова_матриця}(1, 1)$

$C[0][0] = A[0][0] * B[0][0]$

 повернути C

інакше:

 розділити матриці A і B на підматриці $A_{11}, A_{12}, A_{21}, A_{22}$ та $B_{11}, B_{12}, B_{21}, B_{22}$

$X_1 = \text{Штрассен_Множення_Матриць}(A_{11} + A_{22}, B_{11} + B_{22})$

$X_2 = \text{Штрассен_Множення_Матриць}(A_{21} + A_{22}, B_{11})$

$X_3 = \text{Штрассен_Множення_Матриць}(A_{11}, B_{12} - B_{22})$

$X_4 = \text{Штрассен_Множення_Матриць}(A_{22}, B_{21} - B_{11})$

$X_5 = \text{Штрассен_Множення_Матриць}(A_{11} + A_{12}, B_{22})$

$X_6 = \text{Штрассен_Множення_Матриць}(A_{21} - A_{11}, B_{11} + B_{12})$

$X_7 = \text{Штрассен_Множення_Матриць}(A_{12} - A_{22}, B_{21} + B_{22})$

$C_{11} = X_1 + X_4 - X_5 + X_7$

$C_{12} = X_3 + X_5$

$C_{21} = X_2 + X_4$

$C_{22} = X_1 - X_2 + X_3 + X_6$

 об'єднати $C_{11}, C_{12}, C_{21}, C_{22}$ у нову матрицю C

 повернути C

Кроки:

1. Якщо розмір матриці співпадає з одиницею (тобто матриці є 1×1), тоді виконується просте множення чисел цих матриць.

2. У протилежному випадку матриці A і B розбиваються на 4 підматриці кожна ($A_{11}, A_{12}, A_{21}, A_{22}$ і $B_{11}, B_{12}, B_{21}, B_{22}$ відповідно).

Розбиття матриці на чотири частини — це ключовий етап у методі Штрассена.

Якщо матриця A має розмір $n \times n$, то її можна розбити на чотири підматриці розміром $n/2 \times n/2$:

$$A = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix}$$

де $A_{11}, A_{12}, A_{21}, A_{22}$ — матриці розміром $n/2 \times n/2$.

3. Потім проводяться рекурсивні виклики функції $\text{Штрассен_Множення_Матриць}$ для визначення проміжних матриць X_1 до X_7 , які потрібні для обчислення кінцевого результату.

4. Остаточні підматриці $C_{11}, C_{12}, C_{21}, C_{22}$ обчислюються з використанням проміжних матриць X_1 до X_7 і потім об'єднуються в кінцеву матрицю C .

Цей алгоритм ефективний, оскільки використовує менше операцій у порівнянні з класичним методом множення для великих матриць. Такий підхід зменшує обсяг обчислень, але він ефективний тільки для певного діапазону розмірів матриць і може вимагати додаткової обробки для невідповідних розмірів. Але він працює оптимально лише для матриць розміром, який є степенем числа 2.

Для оптимальної роботи алгоритму Штрассена потрібно враховувати базовий випадок, коли розмір матриці стає дуже малим, що дозволяє використовувати звичайний алгоритм множення матриць для таких малих підматриць.

Теорія груп

Німецький математик Леонард Ейлер називав алгебру універсальною арифметикою, а потім, в 19 столітті, оформилось те, що називають універсальна алгебра, або абстрактна алгебра. В 1898 році англійський математик Альфред Норт Вайтгед видав книгу "Універсальна алгебра", в якій описує булеву алгебру, векторну алгебру, матричну алгебру, кватерніони та інше. Кожна з цих алгебр описує якую алгебраїчну структуру. Вивченням алгебраїчних структур займається абстрактна алгебра.

Теорія груп — розділ математики, який вивчає властивості алгебраїчної структури під назвою "група".

Теорія груп походить з часів Огюстена Коші (1789 — 1857). Йому належать перші спроби класифікації з метою формування теорії з ряду ізольованих фактів. Еварист Галуа показав, що кожному рівнянню кінцевого ступеня відповідає група кінцевого порядку, від якої залежать усі властивості рівняння.

У середині дев'ятнадцятого століття було зроблено багато доповнень, головним чином французькими математиками. Перший цілісний виклад теорії було надано в третьому виданні "Курс вищої алгебри" Жозефа Серре, яке було опубліковано в 1866 році. За ним у 1870 році вийшов "Трактат про підстановки та алгебраїчні рівняння" Каміля Жордана. Більша частина трактату Жордана присвячена розвитку ідей Галуа та їх застосуванню до теорії рівнянь.

Ніякого значного прогресу в теорії, окрім її застосувань, не було досягнуто до появи в 1872 році мемуарів норвезького математика пана Людвига Силова (Сюлова) "Теореми про групи підстановок" у п'ятому томі журналу "Математичні аннали". З моменту написання цих мемуарів ця теорія постійно розвивалася.

В 1897 році вийшла книга "Теорія груп" англійського математика Вільяма Бернсаайда (1852 — 1927).

Всі математики 19 століття використовували найвну теорію множин, тобто не описували аксіоми теорії множин.

Доведення теореми для абстрактної алгебраїчної структури означає, що ця теорема буде вірною для всіх моделей цієї структури. Це одна з ключових особливостей математичного доведення теорем у контексті абстрактної алгебри.

Моноїд — це алгебраїчна структура, що складається з не пустої множини, на якій визначена операція, яка є асоціативною, і в множині міститься спеціальний елемент (звичай називається "одиницею" або "нейтральним елементом"), такий, що множення будь-якого елемента на цей нейтральний елемент дає той самий елемент.

Група — це моноїд, в якому для кожного елемента існує обернений елемент, такий, що множення цього елемента на обернений дає спеціальний елемент (умовно - "одиниця"). У групі кожен елемент має обернений елемент, і ця операція обернення є обов'язковою для групи.

У алгебрі, кільце — це алгебраїчна структура, яка складається з множини разом з двома бінарними операціями: додаванням і множенням. Ці операції повинні задовольняти певним властивостям:

Замкненість відносно додавання: Це означає, що сума двох елементів кільця також є елементом кільця.

Асоціативність додавання: Вираз $"(a + b) + c"$ дорівнює $"a + (b + c)"$ для будь-яких елементів a, b і c в кільці.

Існування нейтрального елемента додавання: В кільці має існувати такий елемент, який не змінює інші елементи при додаванні. Цей елемент зазвичай позначається 0 .

Існування оберненого елемента відносно додавання: Для кожного елемента a в кільці повинен існувати обернений елемент $(-a)$, такий, що $a + (-a) = 0$.

Замкненість відносно множення: Це означає, що добуток двох елементів кільця також є елементом кільця.

Асоціативність множення: Вираз $(a * b) * c$ дорівнює $a * (b * c)$ для будь-яких елементів a, b, c в кільці.

Розподільність: Множення повинно розподілятися відносно додавання. Це означає, що для будь-яких елементів a, b, c в кільці, $a * (b + c) = (a * b) + (a * c)$ і $(b + c) * a = (b * a) + (c * a)$.

Кільце може бути комутативним (коли множення комутативне, тобто $a * b = b * a$ для всіх a, b) або некомутативним (коли множення не комутативне). Якщо кільце має одиницю, то вона позначається зазвичай як 1 , і вона має властивість, що $1 * a = a * 1 = a$ для будь-якого елемента a в кільці.

Щоб кільце стало полем, воно повинно відповідати двом основним вимогам:

Існування оберненого елемента відносно множення: Для кожного ненульового елемента a в кільці повинен існувати обернений елемент (позначається зазвичай як a^{-1}), такий, що $a * a^{-1} = 1$, де 1 - нейтральний елемент відносно множення. Ця вимога означає, що кожен ненульовий елемент має обернений елемент відносно множення, і операція множення є операцією, в якій немає нульових дільників.

Замкненість відносно додавання і множення: Кільце повинно бути замкнутим відносно операцій додавання і множення. Це означає, що для будь-яких двох елементів a, b в кільці, їх сума $(a + b)$ і добуток $(a * b)$ також повинні бути елементами кільця.

Якщо кільце відповідає обом цим вимогам, то воно стає полем.

Монойд \rightarrow Група \rightarrow Кільце \rightarrow Поле.

Ось кілька прикладів полів, які широко використовуються в математиці:

Поле раціональних чисел (Q): Це поле складається з усіх дробових чисел, де чисельник і знаменник є цілими числами, і знаменник не дорівнює нулю. Операції додавання і множення визначені звичайним способом.

Поле дійсних чисел (R): Це поле складається з усіх дійсних чисел, включаючи раціональні та ірраціональні числа. Воно є розширенням поля раціональних чисел.

Поле комплексних чисел (C): Це поле включає всі комплексні числа, які можна представити у вигляді $a + bi$, де a та b - це дійсні числа, а i - уявна одиниця ($i^2 = -1$). Поле комплексних чисел розширяє поле дійсних чисел.

Натуральні числа $(1, 2, 3, 4, \dots)$ і цілі числа $(\dots, -2, -1, 0, 1, 2, \dots)$ не формують поле згідно зі стандартними визначеннями алгебричних структур. Поле має включати в себе дві бінарні операції — додавання та множення — і відповідні властивості, а саме, існування обернених елементів.

Які найпопулярніші поняття в статистиці?

Медіана

Медіана — це значення, яке розділяє впорядкований набір даних на дві рівні половини. Іншими словами, це середнє значення двох центральних значень, коли дані впорядковані.

Для знаходження медіані потрібно впорядкувати дані у зростаючому порядку і взяти середнє значення, якщо кількість значень непарна. Якщо кількість значень парна, то медіана — це середнє арифметичне двох центральних значень.

Розглянемо набір даних: 2, 5, 8, 10, 14, 18.

Даний набір даних вже впорядкований. Таким чином, медіана — це середнє значення між 8 і 10, тобто $(8 + 10) / 2 = 9$. Середнє арифметичне: $(2 + 5 + 8 + 10 + 14 + 18) / 6 = 57 / 6 = 9.5$.

Розглянемо набір даних: 1, 5, 7, 10, 16. Набір впорядкований й кількість значень непарна, тому медіана = 7.

Середнє арифметичне — це сума всіх значень в масиві даних, поділена на їх кількість.

Медіана краща у випадках, коли є великі відхилення в даних, оскільки вона не чутлива до екстремальних значень, на відміну від середнього арифметичного.

Рухоме середнє (Ковзне середнє)

Ковзне середнє — це статистичний показник, який обчислюється шляхом усереднення значень певної кількості попередніх значень. Для обчислення ковзного середнього, ви берете середнє арифметичне значень певної кількості останніх спостережень. Потім з кожним новим спостереженням ви обновляєте це середнє значення, викидаючи найстаріше спостереження і додаючи нове.



Даний ряд чисел та фіксований розмір підмножини, перший елемент ковзного середнього отримується шляхом обчислення середнього значення початкової фіксованої підмножини числового ряду. Потім підмножина модифікується за допомогою "зсуву вперед"; іншими словами, виключаючи перше число з ряду та включаючи наступне значення у підмножину.

Простий приклад використання ковзного середнього:

Уявімо, що у вас є дані про щоденні ціни акцій компанії XYZ за останні 10 днів:

Day 1: 50
Day 2: 52
Day 3: 51
Day 4: 55
Day 5: 56
Day 6: 54
Day 7: 53
Day 8: 57
Day 9: 58
Day 10: 59

Щоб розрахувати ковзне середнє за 5 днів, спочатку ви берете середнє арифметичне перших 5 значень:

$$(50 + 52 + 51 + 55 + 56) / 5 = 52.8$$

Потім, коли надходить нове спостереження, викидаєте найстарше і додаєте нове, обчислюючи ковзне середнє за наступні 5 днів:

Day 2: 52
Day 3: 51
Day 4: 55
Day 5: 56
Day 6: 54

$$(52 + 51 + 55 + 56 + 54) / 5 = 53.6$$

Таким чином, ви можете спостерігати за ковзним середнім для визначення загального напрямку руху цін акцій та гладшої динаміки цін.

Ковзне середнє може бути використане для згладжування шумів в часових рядах даних, прогнозування майбутніх значень на основі попередніх, виявлення тенденцій або циклів в даних тощо.

Лінійна інтерполяція

Інтерполяція — це метод обчислення значень функції або даних в точках, які знаходяться між відомими значеннями. Основна ідея інтерполяції полягає в побудові нової функції (інтерполяційної функції), яка проходить через відомі точки. Ця нова функція дозволяє отримувати значення в будь-якій точці в межах інтервалу між відомими значеннями. Одним з прикладів інтерполяції є лінійна інтерполяція.

Лінійна інтерполяція — це метод апроксимації функції або значень між двома відомими точками шляхом побудови лінійного сегмента (відрізка) між цими точками.

Нехай у нас є дві точки з координатами (x_1, y_1) і (x_2, y_2) , і ми хочемо знайти значення функції для проміжної точки з координатою x між x_1 і x_2 . Лінійна інтерполяція використовує рівняння прямої лінії для побудови апроксимаційного відрізка:

$$y = y_1 + (x - x_1) * (y_2 - y_1) / (x_2 - x_1), \text{ де}$$

$$(x - x_1) * (y_2 - y_1) / (x_2 - x_1) — \text{нахил прямої}.$$

Це рівняння лінії, яке проходить через дві відомі точки. За допомогою цього рівняння можна обчислити апроксимоване значення функції для будь-якої точки між x_1 і x_2 .

Сліпе рандомізоване контролюване дослідження

"Blind randomized controlled trial" (RCT) – термін у науковому дослідженні, що вказує на специфічний дизайн клінічного експерименту. Розглянемо окремі частини цього терміну:

Randomized Controlled Trial (RCT): Це тип клінічного дослідження, де учасники випадковим чином розподіляються між групами, одна з яких отримує активне втручання (наприклад, новий лікарський препарат), а інша – контрольне (наприклад, плацебо або стандартний лікувальний підхід). Це робить результати більш об'єктивними та надійними. Якщо ви зібрали групу людей, в якій всі діти, тоді це не був рандомізований (випадковий) вибір з множини всіх людей.

Blind (Single-Blind або Double-Blind): У blind RCT учасники дослідження або деякі інші учасники (наприклад, лікарі або оцінювачі результатів) не знають, якій групі призначено активне втручання, а якій – контрольне. Якщо тільки учасники не знають, це називається "single-blind". Якщо й учасники, і ті, хто проводить дослідження, не знають, то це "double-blind".

Сліпе рандомізоване контролюване дослідження (blind randomized controlled trial) – це дослідження, де учасники випадковим чином розподіляються між групами, і чи не всі, або навіть ніхто з учасників, не знають, хто отримує активне втручання, а хто – контрольне. Це сприяє уникненню "людського фактора" або впливу особистих переконань на результати дослідження.

Одне з перших контролльних досліджень описане в першому розділі Книги Даниїла, де протягом 10 днів тестувалася група з 4 людей. Їм подавали різні страви, відмінні від їхнього звичайного харчування. Мета полягала в перевірці, чи вони будуть мати силу, харчуясь новою їжею.

Шотландський лікар Арчібальд Кокрейн (1909 — 1988) став відомим завдяки своїм внескам у розвиток методів систематичного огляду медичних досліджень та підходів до клінічних випробувань. Арчібальд Кокрейн відомий своєю книгою "Effectiveness and Efficiency: Random Reflections on Health Services". Ця книга пропонує використання рандомізованих контрольних випробувань, щоб зробити медицину ефективнішою.

Чотири вагомі книги в історії наукового методу:

"Елементи" Евклід,

"Принципи математики" Рассел та Вайтгед,

"Логіка наукового дослідження" Поппер,

"Ефективність" Кокрейн.

Нульова гіпотеза

Нульова гіпотеза — прийняте за замовчуванням припущення про те, що не існує зв'язку між двома подіями, що спостерігаються. Так, нульова гіпотеза вважається вірною, поки не можна довести зворотне. У 1935 році британський математик і біолог Рональд Фішер опублікував перше видання книги “Дизайн експериментів”, в якій було представлено нульову гіпотезу.

“Після” не тотожне “внаслідок”. Один з випадків фальшивої причини полягає в тому, що причинно-наслідковий зв'язок ототожнюється з хронологічним. Наприклад, “Сонце встає тому що півень співає”.

Якщо явище проявляється навіть при хибності вашої гіпотези, тоді воно не є її наслідком.

Наприклад, якщо ви проводите експеримент, де досліджуєте ефект нового лікування, нульова гіпотеза може стверджувати, що немає жодної різниці в ефективності між новим лікуванням і плацебо. Ваша мета під час статистичного тестування - визначити, чи маєте ви достатньо доказів, щоб відхилити цю нульову гіпотезу на користь альтернативної гіпотези, яка стверджує наявність ефекту чи різниці.

Кореляція

Кореляція — це статистичний показник, який вимірює ступінь взаємозв'язку між двома змінними. Його значення лежить в межах від -1 до 1. Два найпоширеніших типи кореляційних коефіцієнтів - це Пірсона та Спірмена.

1. Кореляція 1: Позитивна кореляція. Якщо значення однієї змінної збільшується, значення іншої також збільшується. Якщо кореляція дорівнює 1, це означає ідеальний позитивний лінійний взаємозв'язок.

2. Кореляція -1: Негативна кореляція. Якщо значення однієї змінної збільшується, значення іншої зменшується. Якщо кореляція дорівнює -1, це означає ідеальний негативний лінійний взаємозв'язок.

3. Кореляція 0: Відсутність лінійного взаємозв'язку. Якщо значення кореляції близьке до нуля, це вказує на відсутність лінійного взаємозв'язку між змінними.

Якщо кореляційний коефіцієнт між двома змінними дорівнює 0.5, це вказує на помірний позитивний лінійний взаємозв'язок між цими змінними. Однак важливо зазначити, що це не означає причинно-наслідковий зв'язок.

Значення $r=0.5$ свідчить про помірний позитивний лінійний взаємозв'язок між двома змінними. Зростання значень однієї змінної супроводжується помірним зростанням значень іншої змінної, і навпаки.

Кореляція не вказує на причинно-наслідкові зв'язки між змінними, а лише на те, наскільки вони лінійно взаємозалежні. Важливо також пам'ятати, що кореляція не завжди означає причинно-наслідковий зв'язок, а лише взаємозв'язок між змінними.

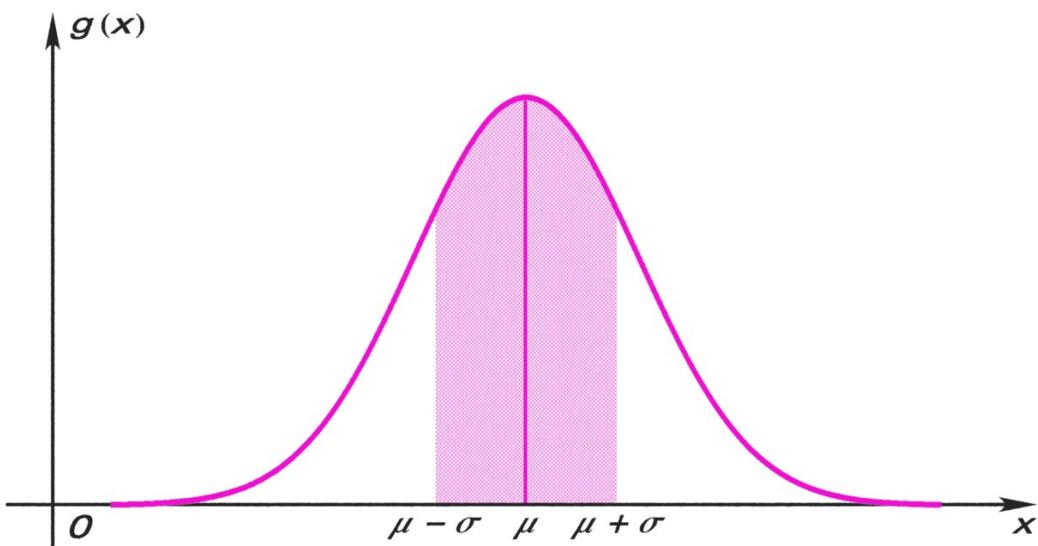
Упередження виживання

Упередження виживання (Survivorship bias) — це специфічний вид систематичної помилки в мисленні, коли в аналізі враховуються лише "вцілілі" чи "успішні" приклади, і не враховуються ті, що відпали або зазнали невдач. Ця помилка може привести до неправильного розуміння ситуації чи прийняття хибних рішень через відсутність повної картини. Зокрема, філософ Августин Аврелій вказував на помилку людей, які звертаються до астрологів, а саме на те, що вони надають значення тому, що відбулося, і не беруть до уваги всі випадки, коли астролог помиляється.

Під час Другої світової війни статистик Абрахам Вальд врахував упередження виживання у своїх розрахунках мінімізації втрат бомбардувальників від ворожого вогню. Вальд зазначив, що дослідження врахувало лише літаки, які повернулися з завдання — збиті бомбардувальники не були присутні для оцінки їх пошкоджень. Таким чином, дірки у літаках, що повернулися, були ділянками, попадання в які насправді дозволяє бомбардувальному повернутися на базу. Натомість він запропонував ВМС додати броні на ділянки, які були неушкоджені на вцілілих літаках, оскільки при попаданні у ці ділянки, літак буде збитий.

Дисперсія випадкової величини

Дисперсія (variance) — це міра розсіяння значень випадкової величини відносно середнього значення розподілу. Більші значення дисперсії свідчать про більші відхилення значень випадкової величини від центру розподілу. Інакше кажучи, це математичне сподівання квадрату відхилення цієї змінної від її очікуваного значення (її математичного сподівання).



Математично дисперсію випадкової величини можна визначити як середнє значення квадратів відхилень кожного значення від середнього значення. Якщо X - випадкова величина зі значеннями x_1, x_2, \dots, x_n та середнім значенням μ , то формула для дисперсії (σ^2 , або $\text{Var}(X)$) виглядає так:

$$\text{Var}(X) = \sigma^2 = (1/n) * \sum (x_i - \mu)^2, \text{ де } i \text{ змінюється від } 1 \text{ до } n.$$

Використання квадрату в формулі для дисперсії забезпечує відсутність від'ємних значень. Квадрат (степінь 2) дозволяє уникнути від'ємних значень, адже відхилення можуть бути як позитивними, так і негативними. Піднесення до квадрата забезпечує додатні значення.

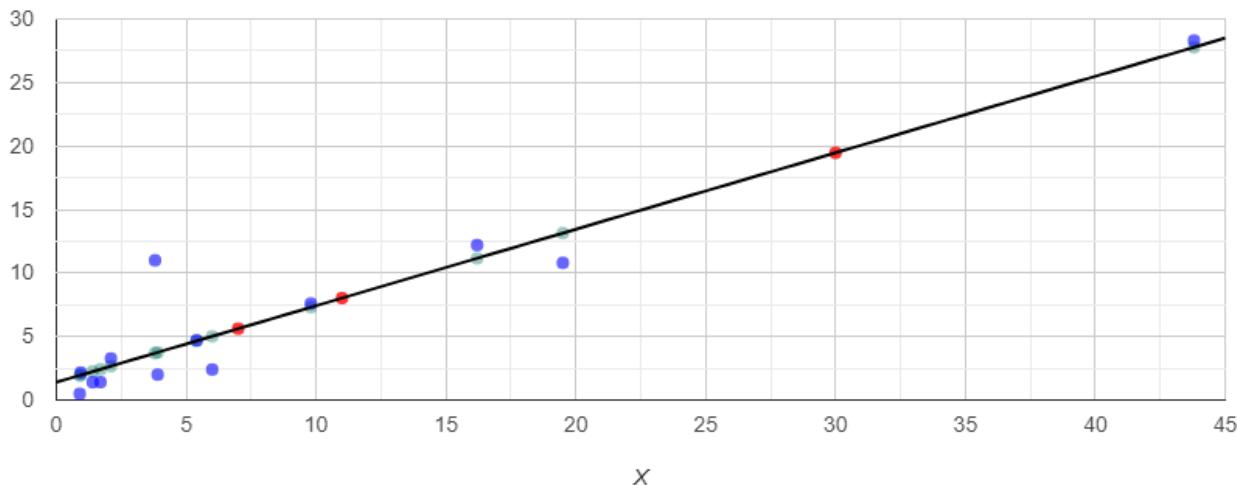
$$\text{Математичне сподівання } (\mu) = (x_1 * p_1) + (x_2 * p_2) + \dots + (x_n * p_n)$$

де x_1, x_2, \dots, x_n - це можливі значення змінної, і p_1, p_2, \dots, p_n - ймовірності відповідних значень. Зазвичай величина x_i множиться на ймовірність його виникнення і всі ці значення сумуються разом, щоб отримати математичне очікування.

Серед відомих розподілів випадкової величини є розподіл Гауса (нормальний розподіл), розподіл Пуассона, розподіл Максвелла, біноміальний розподіл.

Лінійна регресія

Лінійна регресія є статистичним методом, який використовується для вивчення залежності між однією змінною (називається залежною, Y) та однією або кількома іншими змінними (називаються незалежними, X). У випадку лінійної регресії припускається, що залежність між змінними є лінійною, тобто може бути описана рівнянням прямої лінії.



Метод лінійної регресії може використовуватися для передбачення значень залежної змінної (Y) на основі відомих значень незалежної змінної (X). При цьому можна робити як інтерполяцію (передбачення значень в межах відомих даних), так і екстраполяцію (передбачення значень за межами відомих даних).

Важливо відзначити, що лінійна регресія має свої припущення, і вона ефективна лише в тих випадках, коли залежність дійсно є лінійною. Якщо взаємозв'язок складніший, можуть бути застосовані складніші моделі регресії.

Рівняння лінійної регресії таке:

$$y = B * x + A$$

Тут y — залежна змінна, x — незалежна змінна, а A і B — коефіцієнти, що визначають нахил і перехоплення.

1. Обчисліть середнє значення:

- Середнє значення x: $x_{\text{mean}} = (\sum x) / n$
- Середнє значення y: $y_{\text{mean}} = (\sum y) / n$

2. Обчисліть коваріацію:

- Коваріація x і y: $\text{Cov}(x, y) = (\sum (x_i - x_{\text{mean}})(y_i - y_{\text{mean}})) / n$

3. Обчисліть дисперсію:

- Дисперсія x: $\text{Var}(x) = (\sum (x_i - x_{\text{mean}})^2) / n$

4. Обчисліть нахил (B):

- Нахил: $B = \text{Cov}(x, y) / \text{Var}(x)$

5. Обчисліть перехоплення (A):

- Перетин: $A = y_{\text{mean}} - B * x_{\text{mean}}$

A/B тестування

“A/B тестування — підхід статистичного тестування, щоб визначити, яка з двох систем або компонентів працює краще” (ISTQB Glossary)

A/B тестування — це методика експериментування, яка використовується в маркетингу та веброзробці для порівняння двох версій чого-небудь (наприклад, вебсайту, рекламного банера, електронної листівки й т. д.), щоб визначити, яка версія працює краще.

В процесі A/B тестування аудиторія розділяється на дві групи: контрольну групу (A) і тестову групу (B). Обидві групи отримують різні версії об'єкта, що тестується, а потім збираються та аналізуються дані про те, яка версія має кращі показники.

A/B тестування дозволяє визначити, яка з двох альтернатив є більш ефективною на основі конкретних метрик, таких як конверсія, клікабельність, час на сайті тощо. Цей метод допомагає покращити ефективність маркетингових кампаній та вебсайтів шляхом емпіричного тестування та аналізу результатів.

Які найпопулярніші алгоритми й теореми в економіці?

Теорія ігор має справу з інтерактивними ситуаціями, коли дві або більше особи, які називаються гравцями, приймають рішення, які спільно визначають кінцевий результат. Існує кооперативна гра, де гравці грають як команда заради спільної мети, і некооперативна, де кожен гравець має власну мету та вигоду. Також є ігри, коли всі гравці можуть програти все, та ігри, де завжди залишається переможець.

Гра називається кооперативною, якщо гравці можуть об'єднуватися в групи, взявши на себе деякі зобов'язання перед іншими гравцями і координуючи свої дії. Цим вона відрізняється від некооперативних ігор, в яких кожен зобов'язаний грати за себе.

З виходом у світ монографії американських математиків Джона фон Неймана та Оскара Моргенштерна "Теорія ігор і економічна поведінка" (1944), теорія ігор сформувалася як самостійна математична дисципліна. На відміну від інших галузей математики, які мають переважно фізичне, або фізико-технологічне походження, теорія ігор із самого початку свого розвитку була направлена на розв'язання задач, які виникають в економіці (а саме в конкурентній економіці).

Елементарна задача теорії ігор:

Вуличний музикант Ділан заробляє 60 доларів на день, граючи самостійно. Вуличний музикант Кеш заробляє 40 доларів на день, граючи самостійно (соло). Заради експерименту двоє музикантів Ділан і Кеш можуть спробувати зіграти дуетом, тобто разом, і подивитися, скільки грошей вони таким чином зароблять. Скажімо, Ділан і Кеш грали дуетом і разом заробили 120 доларів, що на 20 доларів більше, ніж якби вони грали окремо. Таким чином, цим двом музикантам вигідніше грати разом, бо разом вони заробляють більше. Залишається питання, як поділити суму, зароблену спільними зусиллями. Один зі способів полягає в наступному: із суми, заробленої спільною працею (тобто 120 доларів), кожен із двох музикантів відраховує собі суму, яку він заробив би, граючи поодинці, а залишок ділиться порівну. Тобто Ділан бере собі 60 доларів, Кеш бере 40 доларів (і це $120 - 60 - 40 = 20$), а залишок 20 доларів ділять навпіл, отримують по 10\$. Таким чином Ділан отримав 70 доларів, а Кеш отримав 50 доларів, і виявляється, що їм вигідніше грати разом.

В 17 столітті, ще до праці Лока, англієць і вчитель математики Томас Гоббс почав розвивати політичну теорію на основі того, що зараз називають теорією ігор. Суть його теорії така: "Люди незалежні агенти, які діють у світі для досягнення власних цілей. Якщо кожен такий агент не буде зважати на думку іншого, тоді буде безперервна війна всіх проти всіх. Щоб це уникнути, потрібно домовитись і встановити певні правила гри, які будуть мінімізувати конфлікти між агентами. Ці правила гри, тобто життя в суспільстві, і є законами уряду". Геніальність Гоббса в тому, що він логічно показав неможливість існування егоїстичного суспільства без влади.

"Справжня мати ремесел – математика", - писав Томас Гоббс в 10 розділі своєї книги "Левіафан".

Економіст Марк Блауг (Mark Blaug) є автором двох біографічних словників, присвячених економістам минулого і сьогодення: "Великі економісти після Кейнса" (1985) і "Великі економісти до Кейнса" (1986).

Рівновага Неша

Рівновага Неша — в теорії ігор так називається набір стратегій у некооперативних іграх для двох або більше гравців, в яких жоден учасник не може збільшити виграш, змінивши свою стратегію, якщо інші учасники не змінять свої стратегії. У теорії ігор рівновага Неша, названа на честь математика Джона Форбса Неша-молодшого, є найпоширенішим способом визначення рішення некооперативної гри, в якій беруть участь два або більше гравців. У рівновазі Неша передбачається, що кожен гравець знає стратегії рівноваги інших гравців, і жоден гравець не може нічого отримати, змінюючи лише свою власну стратегію. Джон Неш довів, що коли змішані стратегії (де гравець вибирає ймовірності використання різних чистих стратегій) дозволені, то кожна гра з кінцевою кількістю гравців, в якій кожен гравець може вибрати з кінцевої кількості чистих стратегій, має принаймні одну рівновагу Неша, яка може бути чистою стратегією для кожного гравця або може бути розподілом ймовірності за стратегіями для кожного гравця.

1 приклад рівноваги Неша:

Уявіть собі дві конкуруючі компанії: Компанію А і Компанію Б. Обидві компанії хочуть визначити, чи варто їм запускати нову рекламну кампанію для своєї продукції.

Якщо обидві компанії почнуть рекламиувати, кожна компанія залучить 100 нових клієнтів. Якщо тільки одна компанія вирішить розмістити рекламу, вона залучить 200 нових клієнтів, а інша компанія не залучить нових клієнтів. Якщо обидві компанії вирішать не рекламиувати, жодна з них не залучатиме нових клієнтів. Якщо обидві компанії не рекламиують, все ж рівноваги Неша не буде, бо кожна з компаній матиме потенційно кращу стратегію. Компанія А повинна рекламиувати свою продукцію, оскільки стратегія забезпечує кращу виплату, ніж можливість не рекламиувати. Така ж ситуація існує і для компанії В. Таким чином, сценарій, коли обидві компанії рекламиують свою продукцію, є рівновагою Неша, бо всім вигідно триматись обраних стратегій.

2 приклад рівноваги Неша:

Якщо дві однакові дороги ведуть до потрібного місця і по кожній з доріг їде однакова кількість машин, тоді ця ситуація буде рівновагою Неша, бо нікому з водіїв машин не вигідно змінювати дорогу, бо обидві дороги рівно навантажені.

3 приклад рівноваги Неша:

Дилема ув'язненого така:

Двоє підозрюваних, А і Б, арештовані. У поліції немає достатніх доказів для звинувачення, й ізолявавши їх один від одного, вони пропонують їм одну і ту ж операцію: якщо один свідчить проти іншого, а той зберігає мовчання, то перший звільняється, а другий одержує 10 років в'язниці. Якщо обидва мовчать, у поліції мало доказів, і вони засуджуються до 6 місяців. Якщо обидва свідчать проти один одного, вони одержують по 2 роки. Кожен ув'язнений вибирає, мовчати або свідчити проти іншого. Проте жоден з них не знає точно, що зробить інший. Що відбудеться?

Дилема з'являється, якщо припустити, що обидва піклуються тільки про мінімізацію власного терміну ув'язнення.

Уявимо міркування кожного з ув'язнених. Якщо партнер мовчить, то найкраще його зрадити і вийти на свободу (інакше — півроку в'язниці). Якщо партнер свідчить, то найкраще теж свідчити проти

нього, щоб одержати 2 роки (інакше — 10 років). Стратегія "свідчити" строго домінує над стратегією "мовчати". Аналогічно, інший ув'язнений приходить до того ж висновку.

Приклад гри в якій рівновага Неша веде до неоптимальних результатів:

Гра "Золоті кульки"

Кожному учаснику дістається набір із двох кульок, на кожній з яких написано "Розділити" і "Вкрасти", і вони повинні таємно обрати одну, щоб вказати свої наміри після перевірки, яка кулька є якою. Учасники можуть спілкуватися один з одним.

Якщо обидва вибирають "Розділити", кожен отримує половину джекпоту.

Якщо один обирає "Вкрасти", а інший - "Розділити", той, хто обрав "Вкрасти", виграє весь джекпот, і учасник, який обрав "Розділити", залишається з порожніми руками.

Якщо обидва обирають "Вкрасти", жоден з учасників не отримує жодних грошей.

В цій грі рівновага Неша буде, якщо обидва гравці оберуть варіант "Вкрасти", але тоді ніхто нічого не отримає.

Рівновага Неша в контексті довготривалих військових конфліктів може бути одним зі способів пояснення тривалості таких конфліктів, але вона не є єдиною або вичерпною причиною. Згідно з теорією, конфлікт може тривати, оскільки жодна зі сторін не має мотивації змінювати свою стратегію, яка вже була вибрана в умовах рівноваги. Однак в реальних військових конфліктах існує безліч інших факторів, таких як політичні, соціальні, економічні та культурні, які впливають на тривалість конфлікту.

Джон Форбс Неш-молодший (1928 – 2015) був американським математиком, який зробив фундаментальний внесок у теорію ігор. У пізнішій частині свого життя працюючи старшим математиком-дослідником у Прінстоунському університеті, він розділив Нобелівську премію з економічних наук у 1994 році з теоретиками ігор Рейнхардом Селтеном та Джоном Харсані.

Алгоритм Гейла-Шеплі

Проблема стабільного шлюбу — математична задача в області кооперативних ігор. Необхідно знайти стійкі відповідності між елементами двох груп, учасники яких мають власні вподобання. У простішому формулюванні: скласти подружні пари з наречених таким чином, щоб чоловіка з однієї сім'ї та дружину з іншої не приваблювало один одного більше, ніж до свого законного подружжя. Розв'язок проблеми було відзначено Нобелівською премією з економіки 2012 року. Розв'язок проблеми було описано в 1962 році математиками Девідом Гейлом з Університету Брауна і Ллойдом Шеплі (1923 – 2016) з Прінстоонського університету в статті “Вступ до коледжу та стабільність шлюбу”. Сукупність правил, дотримання яких завжди призводить до утворення стійких пар, називається алгоритмом Гейла-Шеплі.

Нижче наведено конкретний опис алгоритму Гейла – Шеплі;

Існує конструктивний метод пошуку одного з розв'язків проблеми.

1. чоловіки роблять пропозицію найбільш бажаній жінці;
2. кожна жінка з усіх отриманих пропозицій вибирає найкращу і відповідає “можливо”, всім іншим відповідає “ні”;
3. чоловіки, яким відмовляють, звертаються до наступної жінки у списку своїх уподобань, чоловіки, які отримали відповідь “можливо”, нічого не роблять;
4. якщо жінка отримала пропозицію крачу за попередню, то вона каже “ні” попередньому заявику (кому вона раніше сказала “можливо”) і каже “можливо” новому заявику;
5. якщо жінка отримала найкращу пропозицію, то вона каже “ні” колишньому заявику (якому раніше сказала “можливо”), а новому заявику каже “так” і не приймає подальших пропозицій;
6. кроки повторюються до тих пір, поки всі чоловіки не вичерпають список пропозицій, після чого жінки відповідають “так” на ті пропозиції “можливо”, які у них є на цю мить.

Максимальна кількість кроків для реалізації алгоритму: n^2 кроків, де n – кількість чоловіків і жінок.

Принцип Фішера

Принцип Фішера — це еволюційна модель, яка пояснює, чому співвідношення статей у більшості видів, які дають потомство шляхом статевого розмноження, становить приблизно 1:1 між самцями та самками. Принцип Фішера був викладений Рональдом Фішером у його книзі 1930 року "Генетична теорія природного відбору".

Припустимо дано початкову умову, що витрати батьків на виробництво потомства до завершення виховання однакові як для чоловічої, так і для жіночої статі:

Нехай самців буде народжуватися більше ніж самок.

Виникає конкуренція за самок між близькоспорідненими самцями, деякі з яких залишать потомства більше своїх братів, адже кожній самиці для запліднення вистачить і одного самця.

Конкуренція за самок зсуває співвідношення статей в бік домінування жіночої статі. Тому представники, у яких генетично переважало відтворення самців, в перспективі залишать в середньому набагато менше нащадків, ніж особини, що породжують самок. Переваги, що з'явилися, пов'язані з відтворенням самок, зміщують баланс статей до співвідношення 1:1. Вищенаведені міркування справедливі й у зворотному випадку — з переважанням самок. Тому 1:1 є оптимальним співвідношенням рівноваги. Говорячи сучасною мовою, співвідношення 1:1 є еволюційно стабільною стратегією.

Алгоритм, який завжди приносить прибуток

У деяких іграх не існує універсального алгоритму успіху для всіх, адже якби такий алгоритм був, то що було б, якби всі учасники гри застосували його одночасно. Особливо це стосується біржових спекуляцій. Якщо всім учасникам біржових торгів буде відомий точний прогноз зростання цін, то цього зростання або падіння не відбудеться, оскільки прогноз вплине на поведінку учасників, а отже, і на самі торги. Якби існував універсальний алгоритм (метод) успіху в біржових спекуляціях, то виникають суперечності, бо якщо його застосують всі відразу, то хто виграє?!

Фондова біржа — ринок, на якому здійснюється торгівля цінними паперами.

Основні види цінних паперів, які обертаються на фондовій біржі, — це акції й облігації.

Акціонери мають право на отримання частки прибутку компанії у вигляді дивідендів. Дивіденди розділяються між акціонерами у відсотковому співвідношенні до їхньої кількості акцій.

Не існує універсального алгоритму гарантованого виграшу на біржі.

Доказ методом від супротивного.

Припустимо, що існує універсальний алгоритм виграшу на біржі.

Тоді він повинен завжди приносити прибуток. Розглянемо ситуацію, де всі інвестори використовують цей алгоритм. Оскільки біржа базується на взаємодії між купівлею і продажем, існує обмін грошей між учасниками.

Алгоритм, який завжди приносить прибуток, означає, що гроші постійно збільшуються. Проте в економіці немає нескінчених ресурсів, і це призводить до суперечності. Такий алгоритм порушує економічні закони та реалії, тому його існування неможливе.

Отже, припущення про існування універсального алгоритму виграшу на біржі приводить до суперечностей, і можна вважати, що такого алгоритму не існує.

Гроші та криптовалюта

Класичне визначення грошей: Гроші — це товар, який функціонує як міра вартості й, чи то сам по собі, чи через свого представника, як засіб обміну. Отже, золото (або срібло) є грошима.

Досі у французькій мові слово для грошей - "argent", що також означає срібло.

“Гроші є мірою і стандартом вартості та засобом обміну” (Вільям Стенлі Джевонс, “Гроші та механізм обміну”, 1875)

Отже, гроші повинні мати високу ліквідність та достатню стабільність.

Загально прийнято, що перші паперові гроші розробили в середньовічному Китаї. Паперові китайські гроші були підкріплені цінними металами в державній скарбниці, або були випущені у вигляді обіцянки сплатити їх золотом або сріблом у майбутньому.

Це ще не були фіатні гроші у точному сенсі цього слова.

Фіатні гроші - це валюта, яка має вартість лише тому, що держава або центральний банк встановлюють її як законний засіб платежу. Вони не підтримуються жодним фізичним товаром, таким як золото чи срібло, існують лише у формі банкнот та монет, та їх вартість базується на державних законах і умовах суспільного договору.

Обіг фіатних грошей гарантує держава, вона зобов'язує всіх підприємців приймати ці гроші. Навіть, якщо фіатні гроші прямо не підкріплені матеріальними цінностями, зокрема, золотом, сріблом, діамантами, все ж вони мають високу цінність, бо їх емітент (держава) володіє великою кількістю людського ресурсу, земель та корисних копалин, які власник фіатних грошей може використовувати в державі та часто навіть поза нею.

Фіатні гроші, однозначно, просунутий вид грошей, бо використовуються саме як гроші, а не як товар подвійного призначення, типу золото. Наприклад, в 16 столітті відбулася девальвація ринку золота через те, що Іспанці знайшли багато золота в Америці. У людей часто виникала недовіра до чистоти монет (Закон Коперника — Грешема).

У 20 столітті фізики навчилися створювати золото (здійснивши цим мрію алхіміків), тому ціна золота не може бути вищою ніж коштує процес його виготовлення.

Держава не може свавільно, неконтрольовано, друкувати фіатні гроші, бо буде девальвація грошей, інфляція. Люди вимагатимуть більшу заробітну плату під час девальвації.

"Fiat money" означає "let it be money" (від "Dixitque Deus fiat lux et facta est lux").

Італійський математик Лука Пачолі (15 ст.) відомий як батько подвійної бухгалтерії, що стала основою сучасної бухгалтерської системи. Саме завдяки Пачолі ми використовуємо такі поняття як: дебет, кредит, баланс, часова мітка, порядок транзакцій.

Зі створенням мереж для комунікації, з'явилася можливість безготівкового розрахунку та розрахунок з людьми на відстані.

Централізовані банки, які зберігали гроші своїх клієнтів, створили комп'ютерні програми та мережі, з допомогою яких клієнт банку може віддалено користуватися своїм банківським рахунком, може проводити транзакції, здійснювати покупки, платежі онлайн. Безпека такої системи та її працездатність забезпечується банком. Банк використовує спеціальні протоколи, шифри, паролі та інше для захисту системи.

Фактично банк контролює ваш рахунок, якій відображає суму реальних державних грошей, які ви заздалегідь внесли в банк. Мережа, яка має головний вузол і головного адміністратора мережі — називається централізованою. Якщо відбувається атака на головний сервер (умовно мейнфрейм), то вийде з ладу вся мережа. Для захисту від таких атак банк періодично робить бекап даних, копію даних про транзакції. Банк, який якісно надає свої послуги, без шахрайства, матиме більше клієнтів (за "принципом невидимої руки" Адама Сміта).

Таким чином обіг фіатних грошей гарантується державою, а робота централізованих мереж для обміну грошей охороняється законом, але достатньою мірою покладається на порядність банку і на його бажання утримати клієнтів. Було багато випадків шахрайства в банках. Варто зауважити, що демократична держава утворена договором людей, і, значить, що люди забезпечують обіг фіатних грошей.

З вище наданого опису виникають чотири принципові питання, які можна розглядати разом, або незалежно. Ось ці питання:

1. Чи можна зробити платіжну систему без центрального сервера?
2. Чи можуть фіатні гроші гарантуватися не державою, яка виступає центром думки громадян, ніби централізує громадян, а гарантуватися кожним громадянином окремо, децентралізовано.
3. Як зробити повністю цифрові фіатні гроші, тобто віртуальні, але щоб при цьому вони мали цінність?
4. Як зробити децентралізовану платіжну систему безпечною?

Якщо ми забезпечимо таку мережу, тоді ми отримуємо мережу криптовалюти, яка є чесною, надійною й придатною для здійснення реальних обмінів матеріальними благами, послугами.

Вивчення децентралізованої мережі дало багато плодів, зокрема, були відкриті й розроблені: теорема CAP, протокол БітТорент, криптографічні принципи Proof of work.

З кінця 20 століття розглядався варіант створення децентралізованої системи для керування грошима та створення віртуальних грошей, або криптовалюти.

Основні принципи децентралізованих криптовалютних систем:

1. Не має головного сервера, всі учасники мережі рівні (peer to peer).
2. Дані про всі транзакції мережі зберігаються у кожного учасника мережі й періодично синхронізуються.
3. Програмне забезпечення, яким користуються учасники мережі має відкритий програмний код, щоб кожен міг бачити чи програмне забезпечення не є шахрайським.
4. Криптовалюта не залежить від державних (паперових) грошей, а є математичним об'єктом, наприклад, певним унікальним кодом.

(На відміну від підходу, який був ще з 1980-х років, коли банк не керував криптовалютою під час переведу грошей з рахунка на рахунок, а просто проводив транзакцію, яка по суті була зобов'язанням передати певну кількість реальних грошей іншому банку, або умовно іншій особі, збільшивши її онлайн рахунок.)

5. Створений чесний алгоритм синхронізації гілок мережі.
6. Чесний алгоритм випуску нових криптомонет.

У 2008 році під псевдонімом була опублікована стаття в якій пропонувався підхід до створення практичної децентралізованої мережі обміну грошима. Криптовалюта в цій статті була гарно названа bitcoin (біткоїн). Опублікована стаття була під псевдонімом Сатоші Накамото, за яким стояв комп'ютерний спеціаліст, який очевидно працював з групою інших спеціалістів.

"Потрібна електронна платіжна система, заснована на криптографічному доказі, а не на довірі, що дозволяє будь-яким двом бажаючим сторонам здійснювати транзакції безпосередньо одна з одною, не потребуючи довіrenoї третьої сторони" (Сатоші Накамото)

Розглянемо принципи, які запропонував Сатоші.

Сатоші залучив такі поняття як: блок транзакцій, ланцюжок блоків (блокчейн, Blockchain), біткоїн, proof of work (Доказ виконаної роботи), хеш-сума, genesis block.

Суто однорангова версія електронної готівки дозволить надсилати онлайн-платежі безпосередньо від однієї сторони до іншої, не проходячи через фінансову установу. Цифрові підписи є частиною рішення, але основні переваги втрачаються, якщо для запобігання подвійним витратам (double-spending) все ще потрібна надійна третя сторона. Ми пропонуємо вирішення проблеми подвійних витрат за допомогою однорангової мережі (peer-to-peer). Мережа позначає транзакції часовими мітками, хешуючи їх у поточний ланцюжок підтвердження роботи на основі хешування, утворюючи записи, який неможливо змінити без повторного виконання підтвердження роботи. Найдовший ланцюжок слугує не лише доказом послідовності подій, які спостерігалися, але й доказом того, що він походить від найбільшої потужності ЦП (центрального процесора).

Поки більша частина потужності ЦП контролюється вузлами, які не співпрацюють для атаки на мережу, вони створюватимуть найдовший ланцюжок і випереджатимуть атаки.

Ми визначаємо електронну монету як ланцюжок цифрових підписів.

Кожен власник передає монету наступному, підписуючи хеш попередньої транзакції та відкритий ключ наступного власника та додаючи їх до кінця монети.

Одержанувач може перевірити підписи, щоб перевірити ланцюжок власності. Звичайно, проблема полягає в тому, що одержувач платежу не може підтвердити, що один із власників не втратив монету двічі. Поширеним рішенням, якого ми хочемо уникнути, є впровадження довіреного центрального органу, або монетного двору, який перевіряє кожну транзакцію на наявність подвійних витрат. Єдиний спосіб підтвердити відсутність транзакції — бути в курсі всіх транзакцій.

Щоб досягти цього без довіреної сторони, транзакції мають бути публічно оголошенні, і нам потрібна система, щоб учасники погоджували єдину історію порядку їх отримання. Одержанувачу платежу потрібен доказ того, що під час кожної транзакції більшість вузлів погодилися, що це був перший отриманий платіж. Рішення, яке ми пропонуємо, починається з часових міток (timestamp). Мітка часу доводить, що дані повинні існувати в той час, очевидно, щоб потрапити в хеш. Кожна мітка часу включає попередню мітку часу у свій хеш, утворюючи ланцюжок, причому кожна додаткова мітка часу підсилює попередні. Щоб запровадити розподілений сервер часових міток на одноранговий основі, нам знадобиться використовувати систему підтвердження роботи, подібну до Hashcash Адама Бека. Підтвердження роботи передбачає сканування значення, яке під час хешування, наприклад за допомогою SHA-256, починається з кількох нульових бітів. Середня необхідна робота є експоненціальною щодо кількості необхідних нульових бітів і може бути перевірена виконанням одного хешу. Для нашої мережі з часовими мітками ми реалізуємо підтвердження роботи, збільшуєчи nonce у блоці, доки не буде знайдено значення, яке надає хешу блоку необхідні нульові біти. Після того, як зусилля ЦП були витрачені, щоб він задовольняв підтвердження роботи, блок не можна змінити, не повторивши роботу. Оскільки пізніші блоки прив'язуються до нього, робота зі змінами блоку включатиме повторне виконання всіх блоків після нього. Підтвердження роботи також розв'язує проблему визначення представництва в процесі прийняття рішень більшістю.

Щоб змінити минулий блок, зловмиснику доведеться повторно виконати перевірку роботи блоку та всіх блоків після нього, а потім наздогнати та перевершити роботу чесних вузлів. Ймовірність того, що повільніший зловмисник наздожене, експоненціально зменшується з додаванням наступних блоків. Щоб компенсувати збільшення швидкості апаратного забезпечення та різний інтерес до запущених вузлів з часом, складність підтвердження роботи визначається ковзним середнім, орієнтованим на середню кількість блоків на годину. Якщо вони генеруються занадто швидко, складність зростає.

Кроки для запуску мережі такі:

- 1) Нові транзакції транслюються на всі вузли.
- 2) Кожен вузол збирає нові транзакції в блок.
- 3) Кожен вузол працює над пошуком складного підтвердження роботи для свого блоку.
- 4) Коли вузол знаходить підтвердження роботи, він транслює блок усім вузлам.
- 5) Вузли приймають блок, лише якщо всі транзакції в ньому дійсні і ще не витрачені.

6) Вузли висловлюють своє прийняття блоку, працюючи над створенням наступного блоку в ланцюжку, використовуючи хеш прийнятого блоку як попередній хеш.

Вузли завжди вважають найдовший ланцюжок правильним і продовжуватимуть працювати над його розширенням. Якщо два вузли транслюють різні версії наступного блоку одночасно, деякі вузли можуть отримати ту чи іншу першими. У цьому випадку вони працюють над першою отриманою гілкою, але зберігають іншу гілку на випадок, якщо вона стане довшою. Нічия буде розірвана, коли буде знайдено наступне підтвердження роботи та одна гілка стане довшою; вузли, які працювали на іншій гілці, потім переключаться на довшу.

Біткоїни не мають централізованого емітента, такого як центральний банк у випадку традиційних валют. Замість цього, нові біткоїни створюються шляхом процесу, відомого як "добування" або "майнінг". Майнери вирішують складні математичні завдання, щоб підтверджувати та обробляти транзакції у мережі біткоїна, за що вони отримують нові біткоїни як винагороду. Таким чином, біткоїни створюються децентралізовано, через процес майнінгу, а не централізованим емітентом.

Відповідь майнера, який знайшов правильний nonce, може бути перевірена всіма іншими учасниками мережі за допомогою публічного ключа майнера та даних блоку. Коли майнер знаходить правильний nonce, він додає його разом з іншою інформацією до нового блоку. Потім він поширює цей блок мережею. Інші учасники можуть перевірити, чи відповідає nonce вимогам, застосовуючи ті ж самі обчислення до даних блоку, які були згенеровані майнером. Якщо обчислення підтверджуються, блок вважається правильним, і його додають до ланцюжка.

Нонс (nonce) — довільне число використовуване під час криптографічного зв'язку лише один раз. Нонс часто випадкове або псевдовипадкове число утворене протоколом автентифікації, для гарантування унеможливлення використання старих сеансів зв'язку в атаці повторного відтворення.

Genesis блок — це перший блок у блокчайні криптовалюти, такий як Bitcoin. Він містить початковий набір транзакцій і встановлює початковий стан мережі. Genesis блок також містить унікальний хеш, який є важливим елементом безпеки інфраструктури блокчейну.

Сатоші Накамото, псевдонім або група людей, що опублікувала опис Bitcoin, розпочала роботу над першим блоком, який став genesis блоком Bitcoin.

Як новий користувач, ви можете розпочати роботу з біткойнами, не розуміючи технічних деталей. Щойно ви встановите біткойн-гаманець на своєму комп’ютері чи мобільному телефоні, він згенерує вашу першу біткойн-адресу, і ви зможете створити нові, коли вам це знадобиться. Ви можете розкрити свої адреси своїм друзям, щоб вони могли заплатити вам, або навпаки. Насправді це дуже схоже на те, як працює електронна пошта, за винятком того, що адреси Bitcoin слід використовувати лише один раз.

Коли ви використовуєте одну адресу Bitcoin більше одного разу, це може розкрити додаткову інформацію про вашу фінансову історію. Наприклад, воно може дозволити аналітикам відстежувати рух коштів і асоціювати ваші транзакції, що може зробити вас більш вразливими перед аналізом та можливими атаками на приватність. Забезпечення використання нових адрес для кожної транзакції може допомогти зберегти вашу конфіденційність та безпеку.

Блокчайн (ланцюжок блоків) — це спільна публічна книга, на яку спирається вся мережа біткойн. Усі підтвержені транзакції включені в ланцюжок блоків. Це дозволяє біткойн-гаманцям обчислювати баланс, який можна витратити, щоб можна було перевірити нові транзакції, таким чином гарантуючи, що вони фактично належать споживачеві. Цілісність і хронологічний порядок ланцюжка блоків забезпечується криптографією.

Транзакція — це передача вартості між біткойн-гаманцями, які включені в ланцюг блоків. Біткойн-гаманці зберігають секретну частину даних, яка називається приватним ключем або насінням, яка використовується для підписання транзакцій, надаючи математичний доказ того, що вони надійшли від власника гаманця. Підпис також запобігає зміні транзакції будь-ким після її оформлення. Усі

транзакції транслюються в мережу та зазвичай починають підтверджуватися протягом 10-20 хвилин через процес, який називається майнінг.

Майнінг — це система розподіленого консенсусу, яка використовується для підтвердження незавершених транзакцій шляхом включення їх у ланцюг блоків. Він забезпечує дотримання хронологічного порядку в ланцюжку блоків, захищає нейтральність мережі та дозволяє різним комп'ютерам узгодити стан системи. Для підтвердження транзакції мають бути упаковані в блок, який відповідає дуже суровим криптографічним правилам і перевіряється мережею.

Ці правила запобігають зміні попередніх блоків, оскільки це призведе до недійсності всіх наступних блоків. Майнінг також створює еквівалент конкурентної лотереї, яка не дозволяє будь-якій особі легко додавати нові блоки послідовно до ланцюжка блоків. Таким чином, жодна група чи окремі особи не можуть контролювати те, що включено в ланцюг блоків, або замінювати частини ланцюга блоків, щоб повернути свої власні витрати.

Звичайний процес транзакції в мережі біткоїн можна розділити на кілька етапів:

1. Ініціювання транзакції: Користувачі мережі біткоїн можуть створювати транзакції, вказуючи отримувача та кількість біткоїнів, які вони хочуть відправити.
2. Підписання транзакції: Після створення транзакції, вона підписується приватним ключем відповідного гаманця. Це забезпечує автентичність та конфіденційність транзакції.
3. Трансляція у мережу: Підписана транзакція поширюється всією мережею біткоїн за допомогою пірдо-піра зв'язку.
4. Підтвердження транзакції: Транзакція включається до блоку транзакцій і додається до ланцюжка блоків (блокчейн). Після цього вона вважається підтвердженою.
5. Добування блоку і підтвердження: Для додавання блоку до ланцюжка блоків, майнери повинні розв'язати складну обчислювальну задачу, використовуючи алгоритм proof of work. Перший майнер, який успішно розв'язує цю задачу, отримує право створити новий блок та отримати винагороду за це. Після цього інші майнери перевіряють його роботу, і якщо вона правильна, то блок додається до ланцюжка, і транзакція вважається остаточно підтвердженою.
6. Оновлення стану гаманців: Після підтвердження транзакції, стан гаманців відправника та отримувача оновлюється, відповідно до результатів транзакції.

Цей процес дозволяє забезпечити безпеку, автентичність та незмінність транзакцій у мережі біткоїн.

В децентралізованій мережі кожен вузол зазвичай зберігає список адрес інших вузлів у мережі, які він знає.

Блокчейн системи мають свої переваги, але, звісно, і мінуси. Ось деякі з них:

1. Швидкість та масштабування: Один із найбільших недоліків блокчейну - обмежена швидкість та можливості масштабування. Особливо це стає помітним у великих мережах, де кількість транзакцій зростає швидко.
2. Вартість транзакцій: Завдяки процесу децентралізованої підтримки мережі, вартість транзакцій у більшості блокчейнів може бути досить високою, особливо у періоди перевантаження мережі.
3. Енергоефективність: Деякі блокчейн системи, зокрема Proof of Work, вимагають величезного обсягу обчислювальних ресурсів, що призводить до великого споживання електроенергії. Це може бути недопустимим з екологічної точки зору.
4. Приватність та анонімність: Багато блокчейнів забезпечують певний рівень анонімності, проте це може мати й негативні наслідки, такі як фінансування злочинності чи використання для інших незаконних цілей.
5. Великий обсяг даних: Через те, що всі транзакції зберігаються у кожному блоку, обсяг даних у блокчейні може швидко зростати, що створює проблеми для зберігання та синхронізації мережі.

Теорема САР. Теорема САР слабо впливає на Blockchain мережу, оскільки всі вузли в цій мережі рівні.

"БітТорент" (BitTorrent) — відкритий протокол обміну інформацією у мережах типу peer-to-peer. Автором проекту є Брем Коен, який створив першу версію у квітні 2001 разом із першим клієнтом з

тією ж назвою. Протокол розробляли таким чином, аби обмін файлами великих розмірів у мережі був полегшений для її учасників. Один із принципів роботи протоколу BitTorrent такий: завантаження на учасника, який розповсюджує певний файл, зменшується завдяки тому, що клієнти, які його завантажують, починають обмінюватися даними між собою одразу, навіть поки файл повністю не завантажено. Таким чином, клієнти, які завантажили певну частину великого файлу, одразу можуть бути джерелами його розповсюдження. Для отримання інформації про розповсюджувачів певного файла клієнт може звернутися до так званих трекерів.

Трекер (tracker) — спеціалізований сервер, який працює по протоколу HTTP. Трекер використовується для того, щоб клієнти могли знайти один одного. На трекері зберігаються IP-адреси клієнтів, вхідні порти клієнтів та хеш-суми, які унікальним чином ідентифікують об'єкти, що беруть участь у завантажуваннях.

В інформатиці задача про суму підмножини є важливою проблемою вибору в теорії складності та криптографії. Суть проблеми така: для заданої мультимножини цілих чисел, чи існує непорожня підмножина, сума елементів якої дорівнює нулю. Наприклад, якщо дано множину $\{-7, -3, -2, 5, 8\}$, то відповідь Так, тому що сума елементів підмножини $\{-3, -2, 5\}$ дорівнює нулю. Задача про суму підмножини є NP-повною.

Історія програмування

Портрет Джорджа Буля



“Безсумнівно, життя Буля знаменує собою епоху в науці людського розуму”
(Стенлі Джевонс, “Принципи науки”, 1874)

Британський математик Джордж Буль вивчаючи праці Арістотеля, Спінози, Ейлера, де Моргана, розробив булеву алгебру.

Джордж Буль у своїй праці "Закони мислення" (1854) не тільки описує булеву алгебру, а ще й залучає таке поняття як "множина". Джон Венн створив діаграми Венна для опису булової алгебри у формі множин та їх перетинів (по аналогії з діаграмами Ейлера зі збірки "Листи до німецької принцеси"). Потім, опираючись на булеву алгебру й двійкову систему, Джордж Стібіц розробив напівсуматор, який став основою для створення суматорів в процесорах. В 17 столітті Готфрід Лейбніц описав двійкову систему числення, вивчаючи триграми "Книги змін".

Двійкову систему числення ми використовуємо в процесорах, булеву алгебру ми використовуємо в мовах програмування, діаграми Венна в базах даних.

Теорема Кука — Левіна стверджує, що задача здійсненності булевих формул у КНФ (коротше, SAT) є NP — повною.

Історія програмування ≠ історія обчислювальної техніки ≠ історія електроніки.

Три ключові події сталися в історії програмування:

1. Формалізація законів логіки (Буль, Фреге),
2. Формалізація мови для запису алгоритмів для універсальної машини (Тюрінг, Черч). Опис універсальної обчислювальної машини.

3. Розробка бачення про те, що обчислювальна машина повинна бути цифровою, а не аналоговою, тобто працювати з інформацією та даними (Д.Стібіц, Н.Вінер, К.Шеннон).

Історичні етапи на шляху до програмування:

1. Прості абаки (рахівниці), арифметри, годинники, механічні моделі, аналогові комп'ютери.

До 19 століття були створені різні арифметри та прилади для розрахунків. Відомі Антикітерський механізм, арифметр Паскаля, палички Непера, логарифмічна лінійка, диференціальний аналізатор та інші (зокрема, логічне піано Стенлі Джевонса). Це були прилади для автоматизації обчислень, але ми не можемо їх вважати комп'ютерами в сучасному сенсі слова, тим більше програмованими приладами. Вони стосуються обчислювальної техніки загалом. Для програміста вони (спеціальні обчислювальні прилади) цікаві в контексті порівняння їх з універсальною обчислювальною машиною, тобто з сучасним комп'ютером.

2. Автоматичні читачі.

На початку 19 століття француз Жозеф Жаккар удосконалив ткацький верстат так, щоб той зчитував перфокарти й відповідно них створював візерунки на тканині. Але це ще не був програмований пристрій в сучасному сенсі слова.

Наприкінці 19 століття був створений табулятор Голлеріта, який міг рахувати дані на перфокартах, і також були створені музичні інструменти, які могли самі грати мелодії, які були записані на перфокартах. Проста перфокарта — це картонний прямокутний листок з отворами.

З допомогою табуляторів Голлеріта навіть зробили перепис населення в США та Російській імперії. Ці прилади не можна назвати програмованими, бо вони не програмувалися, а просто зчитували дані відповідно до своєї конструкції.

Програмований пристрій — це пристрій, який здатен виконувати програму, а не тільки працювати з даними. До речі, комп'ютер ENIAC (1945) не підпадає під це визначення, оскільки він не мав софтвера, тобто програми, яка б змушувала хардвер (електричні схеми) працювати з даними інакше. Якщо говорити про софтвер дуже приземлено, то це програмне забезпечення в пам'яті комп'ютера, а точніше певний стан реєстрів.

Початок програмування

3. Аналітична машина.

“Моїм незмінним питанням при отриманні будь-якої нової іграшки було:
Мамо, а що всередині?”
(Чарльз Беббідж, “Уривки з життя філософа”)

Аналітична машина Беббіджа була первістком прототипом реального обчислювального пристрою, комп'ютера.

Проте Чарльз Беббідж не зміг її сконструювати в 19 столітті, вона була створена у 20 столітті. Це була повністю механічна машина, без електрики. Двигун для неї планувався паровим, або взагалі розглядався ручний привід.

Ада Лавлейс писала в 1843 році: “Машина Беббіджа може упорядковувати та комбінувати літери чи будь-які інші загальні символи ніби це числові величини”.

4. Об'єктивна математика, конструктивізм в математиці.

Через цілий ряд причин з другої половини 19 століття математики захотіли створити універсальну мову для опису математичного мислення, при чому настільки просту, щоб можна було звести ці

обчислення до простих фізичних маніпуляцій та змусити машину їх виконувати. Це є ключовим моментом для історії програмування.

Ось низка причин, чому математики захотіли створити універсальну мову для конструктивної математики:

- a) Створення Булевої алгебри Джорджем Булем в 1854 році та створення в 1879 році Логіки предикатів Готлобом Фреге дало можливість формально записувати хід математичного мислення, доказу.
- b) Суперечки серед математиків про коректність того чи іншого доказу відродили ідею Лейбніца про необхідність створення точного фреймворку для математичного мислення (універсальної характеристики).
- c) Бажання математиків відкинути мислені химери й працювати з конструктивними об'єктами, що є ключем до прикладної математики.

Таке бажання привело до того, що німецький математик Давид Гільберт поставив цілий ряд запитань, на які було потрібно відповісти на шляху створення алгоритмів, які дозволяють робити математичні обчислення, зокрема, проблема розв'язності.

Проблема розв'язності — проблема, сформульована Давидом Гільбертом 1928 року: знайти алгоритм, який би брав як вхідні дані опис формальної мови та математичного твердження S цією мовою, і після скінченного числа кроків зупиняється й видавав одну з двох відповідей: "Істина" або "Хиба", залежно від того, чи є твердження S істинним, чи хибним. Не потрібно, щоб алгоритм давав якесь обґрунтування своєї відповіді, проте відповідь завжди має бути вірною. Такий алгоритм міг би, наприклад, визначити, чи є правдивими такі твердження, як гіпотеза Гольдбаха або гіпотеза Рімана, попри те, що жодного доведення (або спростування) цих тверджень поки не відомо.

В 1936 році Алонзо Черч та Алан Тюрінг опублікували праці, в яких показали, що не існує алгоритму для визначення істинності тверджень арифметики, а відтак і загальніша проблема розв'язання також не має розв'язку.

Перш ніж можна було дати відповідь на це питання, поняття "алгоритм" мало бути формально визначено. Це було зроблено Алонзо Черчем у 1935 році з концепцією "ефективної обчислюваності" на основі його λ -числення (лямбда-числення), і Аланом Тюрінгом наступного року з його концепцією машин Тюрінга. Тюрінг одразу визнав, що це еквівалентні моделі обчислень. Поняття алгоритм як "загальні правила обчислень" було визначене ще на початку 19 століття в математичному словнику Чарльза Хаттона. Тюрінгу і Черчу потрібно були знайти мову для опису будь-якого алгоритму. Машина Тюрінга може виконати будь-який алгоритм, вона має вічний цикл, команду зупинки, умовні оператори, операції читання/запису, арифметичні операції.

5. Тюрінг повний комп'ютер.

В 1945 році математик Джон фон Нейман описав проект комп'ютера EDVAC, який повинен був бути двійковою програмованою машиною з архітектурою фон Неймана, тобто зберігати дані та команди в одній пам'яті.

В 1948 році математик Норберт Вінер у своїй книзі "Кібернетика" чітко описує принципи створення комп'ютера:

1. Комп'ютер повинен бути цифровим (діджитал), а не аналоговим, бо цифровий має більшу точність.
2. Комп'ютер краще робити на основі двійкової логіки, бо це дає можливість використовувати логічні вентилі, які розробив Джордж Стібіц в Лабораторії Белла.
3. Комп'ютер повинен цілком контролюватися програмою, яку можна змінювати.

Двійкову систему числення описав Лейбніц, вивчаючи китайську "Книгу змін".

Якій перший сконструйований Тюрінг повний комп'ютер?

Свідомо створений перший Тюрінг повний комп'ютер напевно англійський EDSAC (1949), на основі проєкту EDVAC.

Хоча в 1998 році було доведено, що комп'ютер Z3 (1941) був Тюрінг повним, але дуже не зручним у користуванні, бо на ньому складно імітувати умовні оператори, яких прямим чином не було реалізовано.

Комп'ютер Mark 1 Говарда Ейкена не вважається Тюрінг повним, бо не мав умовних операторів та циклів.

6. Мови програмування

Мова Lisp (Лісп) стала першою функціональною мовою, яка була під сильним впливом лямбда-числення Алонзо Черча.

Мову Lisp розробив Джон Маккарті. Вона дозволяла працювати зі списками, масивами. Lisp дозволяла оголошувати функції й мала умовний оператор if. В мові Lisp було вперше реалізоване "збирання сміття" (гарбедж колекшн). Lisp мала динамічну систему типів та технологію "змикання" (closure).

У 50-х роках минулого сторіччя Джон Бекус створив нотація Бекуса — Наура розробляючи мову ALGOL. На першому Всесвітньому Комп'ютерному Конгресі, що відбувся у Парижі 1959-го він зробив доповідь на тему "Синтаксис та семантика пропонованої першої міжнародної алгебраїчної мови". Пізніше Пітер Наур удосконалив її.

В розробці ALGOL 60 брали участь Джон Бекус, творець мови Fortran (1957), та Едсгер Дейкстра, який потім писав про шкідливість оператора go to.

Algol 60 вплинув на Ніколаса Вірта, який створив мову Pascal (1970), Денніса Рітчі, який створив мову C (1972), та врешті на Б'ярна Страуструпа, який створив мову C++ з підтримкою об'єктно-орієнтованого програмування (ООП), яке вже було в мовах Simula (1967) й Smalltalk (1970). Вперше класи були реалізовані в мові Simula (Симула), в якій ви не могли створити об'єкт не створюючи спершу клас. Клас в ООП — це мовна конструкція, яка генерує об'єкти на основі певного шаблону. Мови вищого рівня компілюються в машинний код й потім виконуються, або інтерпретуються й виконуються без попередньої компіляції.

Структурне програмування — це парадигма програмування, яка базується на використанні структурних блоків коду для побудови програм.

Основні принципи структурного програмування включають:

Послідовність (Sequence): Виконання інструкцій в програмі відбувається послідовно, одна за одною.

Вибір (Selection): Використання умовних операторів для вибору між різними шляхами виконання програми в залежності від певних умов.

Цикли (Iteration): Використання циклів для повторення певного блоку коду доти, доки виконується певна умова.

Ці конструкції дозволяють отримати мову повну за Тюрінгом.

Структурне програмування виникло наприкінці 1950-х років із появою мов програмування ALGOL 58 і ALGOL 60, причому остання включала підтримку блочних структур. Фактори, що сприяли його популярності та широкому визнанню, спочатку в академічних колах, а потім і серед практиків,

включають публікацію впливового відкритого листа у 1968 голландським комп'ютерним вченим Едсгером Дейкстрою, який ввів термін "структурне програмування".

У 1968 році Едсгер Дейкстра опублікував свою статтю під назвою "Go To Statement Considered Harmful" ("Команда переходу (go to) вважається шкідливою"), в якій він висловив критику використання безумовних переходів (go to) у програмуванні та пропонував використовувати структурні конструкції, такі як послідовність, вибір та цикл, для забезпечення легкості зрозуміння та управління програмним кодом.

Озираючись на історію розвитку мов програмування можна виділити один цікавий факт, який здається не одразу можна вловити, а саме: Вчені (Рассел, Вайтгед, Пеано, Черч) розробили математичний формалізм для опису всієї математики, потім вчені (Тюрінг, фон Нейман, Бекус, Маккарті) створили комп'ютер, який може цей формалізм розуміти. Основний нюанс в тому, що програмісти досить швидко перескочили машинний код й подібний стиль програмування, наприклад, мову асемблера, й пішли по лінії розвитку компілятора, який би міг розуміти математичний формалізм, зокрема, теорію множин, булеву алгебру, функції першого порядку, які в принципі дозволяють описати всю математику. Мова асемблера чудова, але все ж прив'язана до конкретної фізичної машини, натомість сучасні мови програмування дозволяють писати програму як математичну поему, де замість реальних комірок пам'яті, просто математичні змінні, а машина, читаючи програму, сама розуміє, що для конкретної змінної треба виділити комірку пам'яті й зберегти в ній значення змінної.

FORTRAN (скорочено від FORmula TRANslator) — це одна з найдавніших і найпопулярніших мов програмування, розроблена в 1950-х роках компанією IBM. Початково мова FORTRAN була розроблена для обчислень і наукової обробки даних, особливо в галузі інженерії та науки. Мова FORTRAN надавала можливість швидко створювати програми для великих наукових обчислень, включаючи математичні моделі, обробку даних з експериментів та розв'язання складних рівнянь. Незважаючи на те, що вона була розроблена понад півстоліття тому, FORTRAN залишається використовуваною і досі в багатьох галузях, де потрібні високоефективні обчислення.

Знак "рівно" (=) з часів Давида Гільберта вважають бінарним оператором, тобто функцією з двома аргументами. Часто в літературі знак "дорівнює" (=) використовуються в іншому сенсі, а саме для позначення визначення. Наприклад, в мовах програмування знак дорівнює (=) часто використовують для присвоєння значить змінним, тобто для їх визначення. Наприклад, const a = 5;

7. Операційна система

Операційна система — це програмне забезпечення, яке керує різними аспектами роботи комп'ютера або іншого пристрою. Вона забезпечує інтерфейс між користувачем і апаратним забезпеченням, дозволяючи користувачеві виконувати програми та взаємодіяти з комп'ютером. Операційна система відповідає за керування ресурсами комп'ютера, такими як процесор, пам'ять, диски, мережа тощо, а також за забезпечення безпеки, стабільності та ефективності роботи пристрою. Деякі з найпопулярніших операційних систем включають Windows, macOS, Linux.

Компанія IBM розробили операційні системи OS/360 в 1965, OS/390 в 1995, і купила MS-DOS в команди Microsoft для IBM PC (1981).

У 1970-х років була створена UNIX в лабораторії Bell Labs. Вона стала першою операційною системою, яка була написана мовою програмування C. UNIX став важливим внеском в розвиток мережевих технологій та багаторівневих операційних систем.

В 1982 році Денніс Рітчі описував UNIX так: "Операційна система UNIX в основному складається з трьох частин: 1. Ядро або власне операційна система є частиною, яка відповідає за керування машини та контролює планування різноманітних програм користувача. 2. Shell або інтерпретатор команд

піклується про спілкування між користувачем і самою системою. З. Допоміжні програми (насправді найбільша частина), які виконують певні завдання, як-от редагування файлу, іншими словами, усі інші програми, які безпосередньо не є частиною ядра операційної системи”.

У 1984 році Apple представила Macintosh, який працював на власній операційній системі Mac OS. Mac OS була відома своїм інтуїтивним інтерфейсом та інноваційним дизайном, що забезпечило їй широку популярність серед користувачів.

У 1985 році Microsoft випустила Windows 1.0, що відкрило шлях для розвитку серії операційних систем Windows. Починаючи з Windows 3.0 у 1990 році, Windows став домінуючим операційним середовищем для користувачів персональних комп'ютерів.

Linux, який був створений Лінусом Торвальдсом у 1991 році, став відкритою операційною системою на основі UNIX. Він швидко набув популярності серед розробників і ентузіастів вільного програмного забезпечення.

8. Інтерфейс користувача

Презентація Дугласа Енгельбарта 1968 року називається “Мати всіх Демо”, тому що в ній Енгельбарт дуже вдало показав готовий продукт зі зручним інтерфейсом для простих користувачів або компаній. Енгельбарт сидів за комп’ютерним столом і проводив свою демонстрацію, говорячи в мікрофон, закріплений на його голові, його обличчя та монітор знімалися та відображалися проектором на великому екрані, на який дивилася аудиторія.

Під час презентації Дуглас Енгельбарт сидів за робочою станцією, яка складалась з монітора і блоку керування, до якого були підключені комп’ютерна мишка та клавіатура (точніше, дві клавіатури, одна велика з літерами для друку, інша маленька з кнопками для керування).

Комп’ютерна миша була особистою розробкою Енгельбарта, а все інше він розробляв спільно з іншими програмістами, використовуючи розробки свого часу.

Демонстрація комп’ютерної мишки була нововведенням. Дуглас Енгельбарт рукою рухав маленьку коробочку на столі (тобто мишку), і глядачі бачили, як курсор (стрілка) рухався по екрану.

Робоча станція Енгельбарта була підключена до віддаленого комп’ютера (обчислювальної машини), який обробляв дії Енгельбарта.

Дуглас Енгельбарт міг відкривати текстові документи, друкувати текст і зберігати їх. Збереження виконано віддаленим комп’ютером на віддаленому носії. Примітно, що система Енгельбарта була багатокористувальською, тобто до віддаленого комп’ютера можна було під’єднати кілька робочих станцій (терміналів), за якими могли працювати кілька людей.

Презентація Дугласа Енгельбарта надихнула багатьох комп’ютерників, зокрема команду, яка створила комп’ютер Xerox Alto у 1973 році. Комп’ютер Xerox Alto (Зірокс Алто) насправді був комп’ютером, який підтримував багато терміналів, тобто до якого можна було під’єднати багатьох користувачів.

Важливо, що Xerox Alto, як і система Енгельбарта, була багатокористувальською, мала мишу та вікна на робочому столі.

Ларрі Теслер проводив А/В тестування комп’ютерної миші в 1970-х роках в Xerox Parc. Він виявив, що людям зручніше працювати з мишкою та клавіатурою, ніж тільки з клавіатурою.

У 1979 році Стів Джобс побачив комерційний потенціал Xerox Alto, який керувався мишею та мав графічний інтерфейс користувача (GUI). Це призвело до розробки не надто успішної Apple Lisa у 1983 році, а потім до прориву Macintosh у 1984 році, першого масового комп’ютера з графічним інтерфейсом користувача.

Вони мали інтерфейси з урахуванням WYSIWYG та Drag-and-drop.

Комп’ютер Xerox Alto мав клавіатуру QWERTY. Американський винахідник Крістофер Шоулз створив розкладку "QWERTY" для клавіатури друкарської машинки у 1870-х роках.

9. Комп'ютерна мережа

Перша велика комунікаційна мережа була телеграфною з використанням кодів Морзе. Завдяки Александру Беллу була створена телефона мережа. Потім була створена радіомережа, завдяки Марконі.

Засноване у 1958 році агентство оборонних досліджень США ARPA було ключовим діячем у розвитку Інтернету. В ньому створили ARPANET, попередника Інтернету, яка вперше була введена в експлуатацію в 1969 році.

Багатотермінальні системи, що працюють в режимі поділу часу, стали першим кроком на шляху створення локальних обчислювальних мереж.

Історія протоколу Ethernet сягає коріння в середину 1970-х років, коли команда інженерів в компанії Xerox PARC (Xerox Palo Alto Research Center) в США працювала над розробкою технології для забезпечення зв'язку між комп'ютерами у межах офісу. Одним із ключових дослідників був Роберт Меткалф, який пізніше став співзасновником компанії 3Com.

Сьогодні Ethernet залишається основним протоколом для побудови локальних мереж і підтримує широкий спектр пристройів і технологій зв'язку.

Винайдений Тімом Бернерсом-Лі в 1989 році, протокол HTTP став фундаментом для вебсайтів та інтернет-послуг, що ми використовуємо щодня.

Dot-com Bubble (криза підприємств з суфіксом ".com"): Це період в історії Інтернету, коли стартапи та компанії, що працювали в Інтернеті, отримали величезні інвестиції, але були переоцінені. З підвищеннем цін акцій на біржі у 1990-х роках була створена "пузырчаста" економіка, яка врешті-решт стала причиною краху багатьох компаній в кінці 2000 року.

Проект ЗДАС (Загальнодержавна автоматизована система), розроблений Віктором Глушковим, був амбітним планом автоматизації управління економікою СРСР. Він передбачав створення обчислювальних центрів по всій території Радянського Союзу та їхнє поєднання в одну мережу для збору та обробки інформації. Цей проект мав великий потенціал для оптимізації управління та підвищення ефективності економіки.

Незважаючи на амбітні плани та розрахунки Глушкова щодо економічних переваг проекту, він так і не був здійснений через відмову радянського керівництва фінансувати його.

10. Інтегральна схема

Транзистор був винайдений у 1947 році вченими Джоном Бардіном, Уолтером Браттейном та Вільямом Шоклі в лабораторії Bell Telephone Laboratories. Інтегральні схеми були спроектовані та розроблені Джеком Кілбі в 1958 році в Texas Instruments та Робертом Нойсом в 1959 році в Fairchild Semiconductor. Планарна технологія була винайдена Жаном Горні (Jean Hoerni) у 1959 році також в лабораторії Fairchild Semiconductor.

Перший комерційно успішний мікропроцесор на ім'я Intel 4004 був випущений у 1971 році. Цей чип був розроблений фірмою Intel і мав 4-бітну архітектуру. Він був створений для використання в калькуляторах, проте його значення виявилося набагато ширшим.

Intel 4004 мав тактову частоту 740 кГц і містив приблизно 2,300 транзисторів. Хоча ці параметри незначні у порівнянні з сучасними стандартами, варто зазначити, що в цей час він був дивовижним технологічним досягненням. Головними проєктувальниками процесора були Федеріко Фаджин і Тед Гофф.

Комп'ютер Altair 8800, створений у 1975 році, був одним з перших комп'ютерів, які надавали елементарний інтерфейс (набір лампочок та тумблерів) для налаштування процесора Intel 8080. Інтерес до Altair 8800 швидко зрос після того, як він був представлений на обкладинці випуску журналу "Популярна електроніка" за січень 1975. В Altair 8800 була материнська плата та шина дана S-100, яка потім стала одним зі стандартів для шин даних.

Intellic — серія ранніх мікрокомп'ютерів компанії Intel, розроблена у 1970-х роках як платформа для створення програмного забезпечення мікропроцесорних систем. Intellic були одними з перших мікрокомп'ютерних систем, що продавалися, навіть раніше за Altair 8800. Перша серія Intellics включала Intellic 4 для процесорів 4004.

11. Пам'ять комп'ютерів

Історія розвитку пам'яті комп'ютерів охоплює використання різноманітних технологій. Ось короткий огляд деяких з них:

1. Пам'ять на конденсаторах: Використовувалася у перших електронних обчислювальних машинах. Інформація зберігалася у заряджених конденсаторах, проте ця технологія була нестійкою і вимагала постійного оновлення даних.
2. Пам'ять на реле: Ця технологія використовувала електромеханічні реле для зберігання та обробки інформації. Вона також була застарілою і малоекективною з огляду на швидкість та об'єм.
3. Пам'ять на магнітних дисках (HDD): Ця технологія стала широко використовуватися в особистих комп'ютерах та серверах. Дані зберігалися на магнітних дисках, що дозволяло зберігати великі обсяги інформації.
4. Магнітна стрічка (бобіни, касети): Ця технологія використовує магнітні стрічки для зберігання даних.
5. Пам'ять на магнітних осердях: Ця технологія також використовувалася для зберігання даних у ранніх електронних комп'ютерах. Вона вимагала спеціальних магнітних ядер, щоб представляти біти інформації.
6. Ртутні лінії затримки: Ця технологія використовувала коливання ртуті у тонких трубках для зберігання даних. Вона була використана у деяких ранніх комп'ютерах, але була замінена більш ефективними технологіями.
7. Трубка Вільямса: Це запам'ятовувальний пристрій на основі електронно-променевої трубки, призначений для зберігання в електронному вигляді двійкових даних. Запам'ятовувальні трубки використовувалися як пам'ять на деяких ранніх комп'ютерах, таких як IBM 701.
8. Транзистори з плавним затвором: Ця технологія базується на використанні транзисторів з плавним затвором для зберігання даних. Вона є основою для сучасних пам'ятей DRAM (динамічної оперативної пам'яті), яка використовується в більшості сучасних комп'ютерів.

12. Дисплей.

1. Філ Фарнсуорт відомий завдяки винаходу електронної передавальної трубки — "диссектора", створенню на її основі електронної системи телебачення та тим що вперше в Америці 1 вересня 1928 року публічно продемонстрував передачу рухомого зображення. Згодом диссектор не витримав конкуренції з "іконоскопом" Володимира Зворікіна.

2. LCD — це скорочення від "рідкокристалічний дисплей" (Liquid Crystal Display) і це технологія показу зображення, що використовується в багатьох електронних пристроях, таких як телевізори, монітори, смартфони та годинники.

Принцип роботи LCD полягає в контролюваному перерозподілі рідкокристалічних молекул. Рідкокристалічні молекули можуть відповісти на електричне поле, змінюючи свою орієнтацію. Коли електричний заряд подається до певних областей рідкокристалічного шару, молекули в цих областях переорієнтовуються. Це створює або блокує світло, яке проходить через рідкокристалічний шар, утворюючи зображення на дисплеї.

Зазвичай LCD складається з двох пластин, між якими знаходитьсь рідкокристалічний шар. Один з пластин має транзистори, які керують подачею електричного заряду до рідкокристалічного шару, тим самим управлюючи світловими елементами, які утворюють зображення.

В 1973 році відбулася презентація першого прототипу LCD-дисплея компанії RCA.

3. Світлодіоди були винайдені у 1962 році Ніколасом Голом та Жоржем Кріоном в корпорації General Electric. Перші світлодіоди були червоного кольору. Пізніше, у 1972 році, вони розробили світлодіоди іншого кольору — зеленого. А вже у 1994 році, японські вчені (зокрема, Накамура Сюдзі) створили сині світлодіоди. Ці три кольори (червоний, зелений і синій) є основою для створення всіх інших кольорів шляхом їх змішування в різних пропорціях.

Покоління комп'ютерів зазвичай визначаються за технологічними та архітектурними змінами, які відбулися з часом. Поділ умовний і не точний, бо історично деякі покоління перетинались і навіть екземпляри одного покоління виникали перед екземплярами іншого. Ось деякі основні покоління комп'ютерів:

Перше покоління (1940-1956): Електронні лампи використовувалися для обчислень. Ці комп'ютери були великими, дорогими та енерговитратними.

Друге покоління (1956-1963): Транзистори замінили електронні лампи, що призвело до зменшення розміру, ваги та вартості комп'ютерів.

Третє покоління (1964-1971): Виникнення інтегральних схем (мікросхем) дозволило збільшити продуктивність та знизити ціни.

Четверте покоління (1971-2010): З'явлення мікропроцесорів, що інтегрують тисячі транзисторів на кристалі, прискорило розвиток комп'ютерної технології. Поява особистих комп'ютерів та літакопів.

П'яте покоління (2010-нині): Зростання обчислювальної потужності, мобільність та збільшення кількості підключених пристрій. Розвиток хмарних технологій, штучного інтелекту та інтернету речей.

Література

Загальні праці з математики, теорії обчислення, програмування:

- Джеймс Андерсон, “Дискретна математика і комбінаторика” (James Anderson, Discrete Mathematics With Combinatorics, 2003),
- Дональд Кнут, “Конкретна математика” (Donald Knuth, "Concrete Mathematics", 1994),
- Ріхард Курант, “Що таке математика?” (Richard Courant and Herbert Robbins, "What is Mathematics?", first published in 1941),
- Ервін Маделунг, “Математичний апарат фізики” (Erwin Madelung, “Die mathematischen Hilfsmittel des Physikers”, 1922, Berlin),
- Петро Овчинников, “Вища математика”,
- Яків Зельдович, Ісаак Яглом, “Вища математика для початківців фізиків та техніків” (оригінал на російській),
- Ілля Бронштейн, “Довідник з математики” (оригінал на російській, є англійський переклад),
- Мері Боас, “Математичні методи у фізичних науках” (Mary Boas, "Mathematical Methods in the Physical Sciences", 1966),
- Граніно Корн, “Математичний посібник для вчених та інженерів” (Granino Korn, "Mathematical Handbook for Scientists and Engineers", first published in 1961),
- Елліот Мендельсон, “Вступ до математичної логіки” (Elliott Mendelson, "Introduction to Mathematical Logic", originally published in 1964),
- Володимир Ільїн, “Основи математичного аналізу” (оригінал на російській),
- Волтер Рудін, “Принципи математичного аналізу” (Walter Rudin, "Principles of mathematical analysis", first edition 1953),
- Володимир Зорич, "Математичний аналіз" (оригінал на російській),
- Альфред Ахо, “Проектування та аналіз комп'ютерних алгоритмів” (Alfred Aho, Jeffrey Ullman, John Hopcroft, The Design and Analysis of Computer Algorithms, 1974),
- Джейффрі Ульман, Джон Хопкрофт “Теорія автоматів, мови та обчислення” (John Hopcroft, Jeffrey Ullman, Introduction to Automata Theory, Languages, and Computation, 1979, 2000),
- Томас Кормен, “Введення в алгоритми” (Thomas Cormen, "Introduction to Algorithms", 1989),
- Michael Sipser, "Introduction to the Theory of Computation" (1997),
- Kenneth Ross, Charles Wright, “Discrete Mathematics” (2002),
- Alfred Aho, Jeffrey Ullman, “Compilers: Principles, Techniques, and Tools” (1985),
- Andrew S. Tanenbaum, “Modern Operating Systems 3rd Edition” (2007),
- Edgar F. Codd, “The Relational Model for Database Management: Version 2” (1990),
- Richard Hamming, “Numerical Methods for Scientists and Engineers” (1962),
- Niklaus Wirth, “Algorithms + Data Structures = Programs” (1976),

- Douglas Crockford, "How JavaScript Works" (2018),
- Axel Rauschmayer, "JavaScript for impatient programmers" (2022),
- Axel Rauschmayer, "Tackling TypeScript: Upgrading from JavaScript" (2021)
- Andy Hunt, Dave Thomas, "The Pragmatic Programmer" (1999),
- Glenford Myers, "The Art of software testing" (2011),
- I. M. Порубльов, "Дискретна математика" (Черкаси, 2018),
- Erich Gamma, Richard Helm, "Design Patterns: Elements of Reusable Object-Oriented Software" (1994).
- Lawrence Washington, Wade Trappe, "Introduction to cryptography with coding theory" (2002),
- Джордан Елленберг, "Сила математичного мислення", вид. Наш формат, (How Not to Be Wrong: The Power of Mathematical Thinking, 2014),
- Harold Abelson, Gerald Jay Sussman, "Structure and Interpretation of Computer Programs" (1984, 1996),
- Josh Patterson, Adam Gibson, "Deep Learning: A Practitioner's Approach" (2017),
- Andriy Burkov, "The Hundred-Page Machine Learning Book" (2019),
- Річард Гемінг, "Чисельні методи для науковців та інженерів" (Richard Hamming, "Numerical Methods for Scientists and Engineers", 1962),
- Вільям Феллер, "Введення в теорію ймовірностей та її застосування" (William Feller, "An introduction to probability theory and its applications", 1950),
- В. Баженов, П. Венгерський, "Інформатика. Комп'ютерна техніка. Комп'ютерні технології" (Київ, 4 вид., 2012),
- Christopher Manning, Hinrich Schütze, "Foundations of Statistical Natural Language Processing" (1999),
- А. О. Новацький, "Комп'ютерна електроніка" (КПІ, 2018),
- Martin Fowler, "Refactoring: Improving the Design of Existing Code" (1999),
- А. Є. Конверський, "Сучасна логіка" (Київ, 2017),
- В. Олифер, Н. Олифер, "Компьютерные сети" (изд. 4, Киев, 2010),
- Ендрю Таненбаум, Todd Austin, "Архітектура комп'ютера" (Andrew Tanenbaum, Todd Austin, "Structured computer organization", ed. 6, 2013),
- Дж. Клейнберг, Е. Тардос, "Алгоритми: розробка та застосування" (Jon Kleinberg, Eva Tardos "Algorithm Design", Pearson, 2005),
- Ендрю Таненбаум, Девид Уэзеролл, "Компьютерные сети" (Andrew Tanenbaum, David Wetherall, "Computer Networks", ed. 5, 2012),

- Джеймс Тантон, "Енциклопедія математики" (James Tanton, "Encyclopedia of Mathematics", 2005).
- Роберт Мартін "Чиста архітектура" (Robert Martin, "Clean Architecture" (2017)),
- Pierre Bourque (co-editor), "Guide to the Software Engineering Body of Knowledge (SWEBOK)", IEEE, 2014,
- Roger Pressman, "Software Engineering: A Practitioner's Approach" (7 ed., 2010),
- Shanqing Cai, Stanley Bileschi, "Deep Learning with JavaScript Neural networks in TensorFlow.js" (Manning, 2020),
- Jez Humble, David Farley, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" (Addison-Wesley, 2010),
- Paul M. Duvall, Steve Matyas, Andrew Glover, "Continuous Integration: Improving Software Quality and Reducing Risk" (Addison-Wesley, 2010),
- Роберт Мартін, "Чистий код" (вид. Фабула, 2019),
- Адітья Бхаргава, "Грокаємо алгоритми. Ілюстрований посібник для програмістів і допитливих" (вид. ArtHuss, 2024),
- Lisa Crispin, Janet Gregory, "Agile Testing: A Practical Guide For Testers And Agile Teams" (Addison-Wesley, 2009),
- Jon Erickson, "Hacking: The Art of Exploitation" (2008),
- Bertrand Meyer, "Object-Oriented Software Construction" (1988),
- Robert C. Martin, "Agile Software Development: Principles, Patterns, and Practices" (2003),
- Kenneth Rubin, "Essential Scrum: A Practical Guide to the Most Popular Agile Process",
- Ian Goodfellow, Aaron Courville, "Deep Learning" (2015),
- J. Clark Scott, "But how Do it Know? The Basic Principles of Computers for Everyone" (2009).

Книги по історії обчислювальної техніки:

- Фредерік Брукс, "Міфічний людино-місяць" (Fred Brooks, "The Mythical Man-Month", 1975),
- Герман Голдстайн, "Комп'ютер від Паскаля до фон Неймана" (Herman Goldstine, "The Computer from Pascal to von Neumann", 1972),
- Рафаїл Самойлович Гутер, "Від абака до комп'ютера" (оригінал на російській, 1981),
- Франко Сорезіні, "Історія автоматичного обчислення" (оригінал на італійській, 20 століття),
- Ентоні Ралston, "Енциклопедія комп'ютерних наук" (Anthony Ralston, "Encyclopedia of Computer Science", originally published in 1971),
- Барон Боуден, "Швидше, ніж думка. Симпозіум з цифрових обчислювальних машин" (B.Bowden, "Faster than thought. A symposium on digital computing machines", Pitman Publishing Corp., 1966),
- Роджер Пенроуз, "Новий розум імператора" (Roger Penrose, The Emperor's New Mind, 1989),
- Майкл Вільямс, "Історія обчислювальної техніки" (Michael Williams, "A History of Computing Technology", Wiley-IEEE, 1985),
- Ендрю Таненбаум, "Операційні системи" (Andrew Tanenbaum, "Modern Operating Systems"),
- Чарльз Баше, "Ранні комп'ютери IBM" (Charles Bashe, "IBM's Early Computers", 1986),
- Чарльз Петцольд, "Код: таємна мова інформатики" (Charles Petzold, "Code: The Hidden Language of Computer Hardware and Software", Microsoft Press, 2000),
- Віктор Глушкиков, "Енциклопедія кібернетики" (20 століття),
- Мічіо Каку, "Майбутнє розуму" (Michio Kaku, "The Future of the Mind", 2014),
- Р. Гутер, Ю. Полунов, "От абака до компьютера", (2-е изд., испр. и доп. — М.: Знание, 1981),
- А. Боголюбов, "Математики. Механики. Биографический справочник" (Киев: Наукова думка, 1983),
- Джін Саммет, "Мови програмування: історія й основи" (Jean E. Sammet, "Programming languages: History and Fundamentals", New Jersey, 1969),
- Мартін Девіс, "Математики та походження комп'ютера" (Martin Davis, "Engines of logic: mathematicians and the origin of the computer", W. W. Norton & Company, 2001),
- Девід Кан, "Зломники кодів" (David Kahn, "The Codebreakers", 1967, 1996),
- R. Clark, "The Man Who Broke Purple" (Little Brown & Co, 1977),
- Джеймс Тантон, "Енциклопедія математики" (James Tanton, "Encyclopedia of Mathematics", 2005),

- Brad Cox, “Object-Oriented Programming: An Evolutionary Approach” (Addison Wesley Longman Publish, 1986).

Вебсторінки:

<https://www.scrum.org/>,

<https://agilemanifesto.org/>,

<http://www.extremeprogramming.org/>,

<https://developer.mozilla.org/en-US/>,

<https://learn.microsoft.com/>,

<https://www.w3.org/TR/html-aria/>,

<https://www.conventionalcommits.org/en/v1.0.0/>,

<https://semver.org/>,

<https://bitcoin.org/uk/>,

<https://mathworld.wolfram.com/>,

https://glossary.istqb.org/en_US/,

<https://www.crockford.com/mckeeman.html>,

<https://wiki.haskell.org/Haskell>,

<https://www.geeksforgeeks.org/>,

<https://exploringjs.com/impatient-js/>,

<https://computerhistory.org/>,

<https://ocw.mit.edu/>,

<https://www.rigb.org/>,

<https://encyclopediaofmath.org/>,

<https://www.tensorflow.org/>,

<https://google.github.io/styleguide/>,

<https://refactoring.guru/>,

<https://www.typescriptlang.org/>,

<https://www.w3schools.com/>,

<https://brain.js.org/>,

<https://dev.to/>,

<https://amturing.acm.org/>,

<https://web.mit.edu/>,

<https://ultimalelectronicsbook.com/>,

<https://www.computer.org/education/bodies-of-knowledge/software-engineering>.

<https://learn.microsoft.com/uk-ua/>,

<https://nvie.com/posts/a-successful-git-branching-model/>,

<https://www.e-magnetica.pl/>

Емейл автора книги:

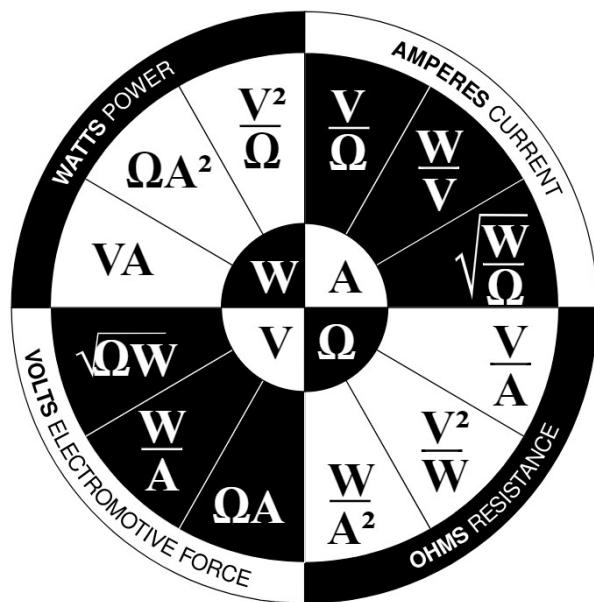
dmytropopov@ieee.org,

dmytro1popov@gmail.com;

Доповнення

Основи електрики

Робота з електрикою потребує дотримання техніки безпеки. Електричний струм може бути небезпечним. Користуйтесь ізоляційними рукавичками, запобіжниками, та постійно перевіряйте чи певний дріт під напругою. Для гасіння пожежі внаслідок електричного замикання використовують вуглекислий (CO_2) вогнегасник. Малі струми, які не перевищують 5 міліампер, зазвичай не вважаються небезпечними для людини.



License of image: [CC BY-SA 3.0](#), Author of image: Per Mejdal Rasmussen.

Хоча різні електричні та магнітні явища вже були відомі до нашої ери (див. книги Лукреція та Вітрувія), методологічно вивчати магнетизм та електрику почали тільки в сімнадцятому столітті. Книга Вільяма Гілберта про магніт (1600) була тільки початком цього процесу, бо його розквіт почався з другої половини 17 століття. Бажаючи перевірити досліди англійця Вільяма Гілберта, в 1663 році німецький вчений Отто фон Геріке створив прилад для добування статичної електрики (Геріке описав це в книзі 1672 року, разом зі своїм експериментом про Магдебурзькі півкулі, які трималися разом шляхом створення в них часткового вакууму).

Отто фон Геріке використовував сірчану кулю, яку можна було обертати та терти вручну, внаслідок чого вона електризувалася. Ісаак Ньютон запропонував використовувати скляну кулю замість сірчаної. Приблизно в 1706 році Френсіс Хоксбі вдосконалів основну конструкцію за допомогою своєї фрикційної електричної машини, яка дозволяла швидко обертати скляну сферу на вовняній тканині. Френсіс Хоксбі виявив світіння на цій кулі в темряві (як повідомляється в книзі “Фізико-механічні досліди Хоксбі” (1719) та “Оптиці” Ньютона).

У 1745 році, в Лейдені, Пітер ван Мушенбрюк, наслідуючи дослідження Евальда фон Клейста, вирішив експериментально перевірити припущення, що електрика є певною “рідиною”. Він використав скляну банку, яка містила дріт, що з'єднувався з електричною машиною, яка заряджалася тертям. З урахуванням того, що скло є ізолятором, заряд накопичився на дроті, що знаходився всередині банки, створюючи ефект як на обкладках конденсатора.

Італійські вчені Луїджі Гальвані, Алессандро Вольта, і англійські вчені Гемфрі Деві та Майкл Фарадей вивчали хімічні джерела струму. Алессандро Вольта створив першу батарею. Батарея Вольти, також відома як гальванічний елемент, була винаходом італійського фізика і хіміка Алессандро Вольта у 1800 році. Цей винахід став першим хімічним джерелом електричного струму. Батарея Вольти складалася з пластин цинку та міді, між якими була тканина промочена у розчині солі. Електроліт – це речовина, яка проводить електричний струм внаслідок дисоціації на іони. Приклади електролітів включають кислоти, солі, луги. Алессандро Вольта побудував стовп Вольти, який є набором цинкових і мідних пластин з шарами просоченої кислотою матерії між ними. Коли Вольта під'єднав дроти до кінців цього стовпа, він побачив, що по них тече струм. Якщо в лимон вставити цинковий і мідний дріт, між кінцями дроту потече слабкий струм (блíзько 1 вольта).

Майкл Фарадей виявив явище електромагнітної індукції у 1831 році. Під час своїх досліджень він помітив, що зміна магнітного поля біля обмоток катушки індукує електричний струм у цих обмотках. На основі цих дослідів були створені генератори електричного струму, трансформатори, електродвигуни, а також різноманітні пристрої для перетворення, передачі та вимірювання електроенергії (гальванометри, вольтметри, амперметри). Закон Фарадея каже: Для будь-якого замкнутого контуру індукована електрорушійна сила (ЕРС) дорівнює швидкості зміни магнітного потоку, що проходить через цілий контур, взятого зі знаком "мінус". Закон Фарадея математично записав Джеймс Клерк Максвелл.

У 1822 році Томас Йоганн Зеебек виявив, що ланцюг, виготовлений із двох різнорідних металів при різних температурах буде відхиляти магнітну стрілку компаса. Такий ланцюг називають термопара (два провідники із різнорідних матеріалів, з'єднані на одному кінці).

Причина ефекту Зеебека полягає в тому, що зміна температури впливає на рух носіїв заряду (наприклад, електронів або "дірок") у матеріалі. У термоелектричних матеріалах, таких як деякі напівпровідники, носії заряду рухаються від гарячих до холодних ділянок матеріалу, створюючи різницю потенціалів. Ця різниця потенціалів призводить до виникнення електричного струму, який може бути використаний для зберігання або генерації електроенергії. Термоелектричний ефект виникає через різницю у температурі між двома контактуючими матеріалами, які мають різні електричні властивості. Один з матеріалів виявляється більш провідним, тоді як інший є менш провідним. При зміні температури виникає різниця у концентрації носіїв заряду у цих матеріалах, що призводить до появи електричної напруги між ними.

21 квітня 1820 року данський фізик і хімік Ганс Крістіан Ерстед (1777 – 1851) опублікував своє відкриття, що стрілка компаса була відхиlena від магнітної півночі Землі електричним струмом поблизу, що підтвердило прямий зв'язок між електрикою та магнетизмом.

Його початкова інтерпретація полягала в тому, що магнітні ефекти випромінюються з усіх боків дроту, по якому проходить електричний струм, а також випромінюються світло і тепло (зокрема, як виявив Вільям Гершель, інфрачервоне світло нагріває матеріали). Через три місяці Ерстед почав інтенсивніше дослідження і незабаром опублікував свої висновки, які показали, що електричний струм створює кругове магнітне поле, протікаючи через дріт.

Висновки Ерстеда викликали багато досліджень електродинаміки в науковій спільноті, впливнувши на розробку французьким фізиком Андре-Марі Ампером єдиної математичної формули для представлення магнітних сил між провідниками зі струмом.

Сила Ампера діє між двома провідниками, притягуючи або відштовхуючи їх один від одного. Вона виникає через взаємодію магнітних полів, створених струмами. Формула для обчислення сили Ампера виглядає як $F = (\mu_0 * I_1 * I_2) / (2 * \pi * r)$, де μ_0 – магнітна проникливість вакууму, I_1 та I_2 – сили струмів, а r – відстань між ними.

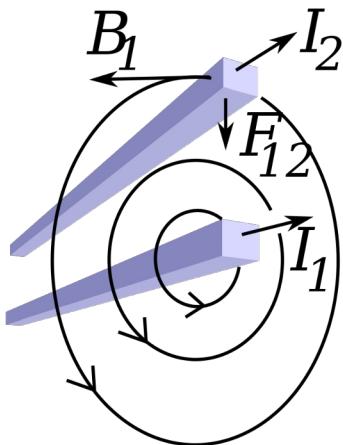
Сила Лоренца діє на заряджену частинку в електромагнітному полі. Формула для обчислення сили Лоренца виглядає як $F = q(E + v \times B)$, де q – заряд частинки, E – електричне поле, v – швидкість частинки, а B – магнітне поле, \times – векторний добуток.

Векторний добуток:

$$\mathbf{a} \times \mathbf{b} = (a_y b_z - a_z b_y) \mathbf{i} + (a_z b_x - a_x b_z) \mathbf{j} + (a_x b_y - a_y b_x) \mathbf{k},$$

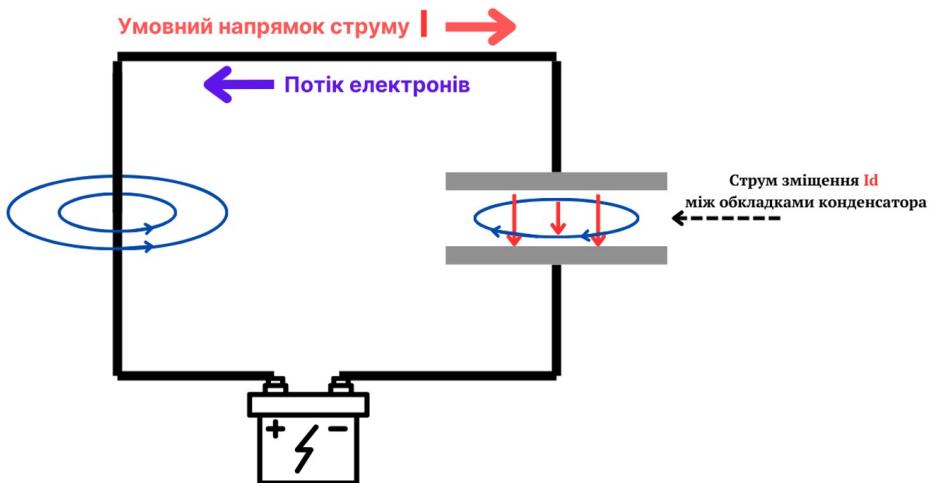
де \mathbf{i} , \mathbf{j} , \mathbf{k} – одиничні вектори вздовж осей x, y, z.

Навколо провідника з електричним струмом утворюється магнітне поле, силові лінії якого оточують провідник (як обмотка навколо серцевини гітарної струни). Вважається, що лінії магнітного поля спрямовані за годинниковою стрілкою, якщо дивитися в напрямку руху струму. Правило гвинта: “Якщо напрямок поступального руху гвинта збігається з напрямком струму в провіднику, то напрямок обертання ручки гвинта збігається з напрямком вектора магнітної індукції”. У магнітостатиці силу тяжіння або відштовхування між двома струмопровідними дротами часто називають силовим законом Ампера. Причина відштовхування криється у взаємодії магнітних полів.



На малюнку: два дроти зі струмом притягуються один до одного: нижній провід має струм I_1 , який створює магнітне поле B_1 . По верхньому дроту проходить струм I_2 через магнітне поле B_1 , тому (через силу Лоренца) на провід діє сила F_{12} . (Не показано одночасний процес, коли верхній дріт створює магнітне поле, що призводить до дії сили на нижній дріт.)

На малюнку зображений конденсатор, на кінцях якого під'єднано металеве коло, по якому протікає електричний струм (в ролі джерела живлення можна використовувати гальванічний елемент). Таким чином, конденсатор заряджається, і між його обкладками присутнє зростаюче електричне поле.



У той час як електричне поле конденсатора змінюється, через з'єднувальний провід (дріт), оточений круговими лініями магнітного поля, тече *струм провідності*. Уявіть собі, що цей малюнок завершено і відповідні лінії поля накреслені навколо всіх інших відрізків дроту. Витягнута таким чином оболонка закінчується з обох боків, коли дроти входять в конденсатор. На противагу цьому Максвелл стверджував, що ця оболонка магнітних силових ліній не має кінців, а утворює замкнуте порожнє кільце, тобто змінне електричне поле конденсатора також оточене круговими силовими лініями магнітного поля. Тому змінне електричне поле конденсатора отримало своєрідну назву, воно називається *струмом зміщення*, оскільки воно має головну характерну особливість електричного струму, а саме, магнітне поле (і електричне), хоча в ньому немає потоку електронів. Навколо змінного магнітного поля виникає змінне електричне поле. Якщо в ньому немає потоку електронів, то така система електричних і магнітних полів називається *струмом зміщення*.

Введемо два типи частинок, виявлені в експериментах Джозефа Томсона та Ернеста Резерфорда, а саме електрон й протон. Протон відштовхується протоном, але притягується електроном. З іншого боку, електрон притягується до протона і відштовхується від електрона. Узгоджено, що електрони несуть негативний електричний заряд, а протони — позитивний.

Слова плюс (+) та мінус (-) для позитивного та негативного заряду ввів Бенджамін Франклін приблизно в 1746 році. Франклін вважав, що річ з надлишком “електричної рідини” є позитивно зарядженою, а з недостачею цієї рідини є негативно зарядженою. В цьому випадку струм тече від плюса до мінуса. Нині заведено навпаки, а саме, тіло з надлишком електронів є негативним, а з браком електронів є позитивним.

Одніменні заряди відштовхуються, різноміенні — притягаються. Крім того, ця електрична сила діє лише на певній відстані (сила обернено пропорційна квадрату відстані). З цього ми можемо зробити висновок, що навколо частинки існує електричне поле (силове поле), яке впливає на інші частинки в ньому. Для спрощення, зараз ми представимо електрони та протони як мікроскопічні кульки (хоча насправді вони є чудернацькими об'єктами, які відрізняються від звичайної твердої частинки корпускулярно-хвильовим дуалізмом та підпадають під принцип невизнаності Гайзенберга).

Маса протона ($1,67 \times 10^{-24}$ г) приблизно у 2000 разів більша за масу електрона ($9,11 \times 10^{-28}$ г).

Електрон відштовхує інші електрони, але водночас не розпадається сам. Це вже наводить на думку, що електрон є інакшим об'єктом, ніж просто невелика кулька. В наш час розподіляють всі взаємодії в природі на 4 категорії: Сильні, Електромагнітні, Слабкі, Гравітаційні. Сильні взаємодії утримують нейтрони та протони в ядрі атома. Слабкі відповідають за ядерний розпад. Гравітаційні — за гравітацією (зокрема, за силу земного тяжіння). Електромагнітні — за відштовхування тіл та притягання, за магнетизм та електрику. Електрон є фундаментальною частинкою, тобто його не можна розкласти на інші частинки. Протон є складеною частинкою, він складається з так званих кварків. Крім протонів, існують також складені частинки, які називаються нейтронами. Нейтрони є нейтральними, тобто вони не мають заряду і не відштовхуються між собою, протонами або

електронами. Хоча нейтрон може перетворитися на протон, випромінюючи електрон. Встановлено, що електрон, протон, нейтрон та кварки можуть бути розкладені на енергію ($E = mc^2$), просто кажучи, ми можемо перетворити їх на рух.

Маса спокою абсолютна. Інерційна маса характеризує інерційність тіл і фігурує у виразі другого закону Ньютона: якщо дана сила в інерціальній системі відліку однаково прискорює різні тіла, їм приписується однаакова інерційна маса.

Називемо імпульсом матеріальної точки вектор, що дорівнює добутку маси точки на її швидкість: $\mathbf{p} = mv$.

У релятивістській механіці, оскільки маса залежить від швидкості, тобто:

$$m = m_0/\sqrt{1-v^2/c^2}, \quad (m - \text{релятивістська маса}),$$

імпульс частинки визначається за формулою:

$$\mathbf{p} = m_0 v / \sqrt{1-v^2/c^2},$$

де m_0 - маса спокою, v - швидкість частинки, c - швидкість світла.

Релятивістська енергія спокою частинки:

$$E_0 = m_0 c^2, \quad \text{де } E_0 - \text{енергія спокою}, \quad m_0 - \text{маса спокою}, \quad c - \text{швидкість світла}.$$

Повна релятивістська енергія частинки дорівнює:

$$E = mc^2 = (m_0 c^2) / \sqrt{1-v^2/c^2}.$$

Кінетична енергія тіла визначається за формулою:

$$E = mc^2 - m_0 c^2.$$

Одиниці величин в рівнянні Айнштайнa, як вони записані, є одиницями метричної системи (система сантиметр-грам-секунда). Якщо m дано в g (грамах), а c у см/сек (сантиметрах на секунду), то числове значення mc^2 є значенням енергії E в ергах.

Вибух ядерної бомби, формула $E = mc^2$ та портрет Альберта Айнштайнa зображені на обгортаці американського журналу "Таймс" за липень 1946 року. Айнштайн вивів формулу $E = mc^2$ експериментально, досліджуючи ядерний розпад. Приблизно в 1933 році в Кембриджі Джон Кокрофт і Ернест Томас Сінтон Волтон спостерігали виділення енергії, коли атом розділили на дві частини, загальна маса яких була меншою за масу вихідного атома (дефект маси). У 1951 році Ернест Волтон і Джон Кокрофт отримали Нобелівську премію з фізики за роботу з трансмутації атомних ядер з допомогою штучно прискорених частинок.

Данський астроном Оле Рьомер (Ремер) в 1676 році помітив, що коли Земля на своїй орбіті знаходиться далі від Юпітера, затемнення супутника Юпітера Іо відстають від обчислень на 22 хвилини. Звідси він отримав значення для швидкості світла 220 000 км/с — неточне значення, але близьке до істинного. Через півстоліття, в 1728 році, відкриття аберрації дозволило Джеймсу Бредлі підтвердити скінченність швидкості світла й уточнити її оцінку: отримане Бредлі значення становило 308 000 км/с. Теорію про те, що світло відхиляється гравітацією, розробили Джон Мічелл, Йоганн Солднер, Альберт Айнштайн і Артур Еддінгтон.

Оле Рьомер помітив і вказав Джованні Кассіні, що час між затемненнями (зокрема, Іо) ставав коротшим, коли Земля та Юпітер наближалися, і довшим, коли Земля віддалялася від Юпітера. Рьомер вважав, що час, необхідний для того, щоб світло подолало діаметр земної орбіти, становило приблизно 22 хвилини (це трохи більше, ніж визначено зараз: приблизно 16 хвилин 40 секунд). Його відкриття було представлено Французькій академії наук і узагальнено.

З теорії електромагнетизму Максвелла і з експерименту Майкельсона та Морлі слідувало, що швидкість світла однаакова у всіх системах відліку. На основі цього Альберт Айнштайн розробив спеціальну теорію відносності, а потім загальну теорію відносності (ЗТВ).

Канадсько-американський математик Саймон Ньюком (1835 — 1909) у 1891 році отримав значення 299 810 км/с з похибкою 50 км/с, а Альберту Майкельсону в 1926 році вдалося зменшити похибку до 4 км/с і отримати значення 299 796 км/с для швидкості. Подальший прогрес був пов'язаний з появою лазерів, які характеризуються дуже високою стабільністю частоти випромінювання, що дозволило визначати швидкість світла, одночасно вимірюючи довжину хвилі та частоту їх випромінювання.

Фізичною основою роботи лазера є явище вимушеного (індукованого) випромінювання. Суть явища полягає в тому, що збуджений атом (або інша квантова система) здатний випромінювати фотон під дією іншого фотона, не поглинаючи його, якщо енергія початкового фотона дорівнює різниці між енергіями рівнів атома до і після випромінювання. У цьому випадку випромінюваний фотон когерентний з фотоном, який викликав випромінювання. Таким чином, світло посилюється.

7 липня 1960 року Теодор Майман презентував свій лазер. Його основу складав монокристал штучного рубіна, що створював інтенсивне випромінювання червоного кольору.

Швидкість світла як фундаментальна фізична стала:

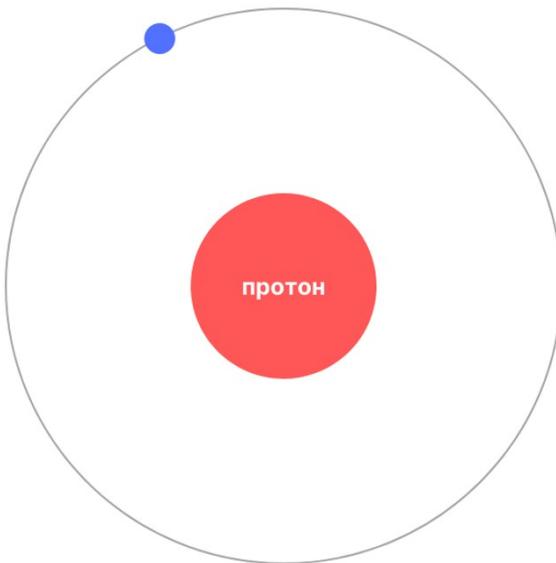
$$c = 299\,792\,458 \text{ м/с.}$$

Дослід Майкельсона і Морлі показав, що швидкість світла також є постійною (з точністю 10^{-18}) незалежно від того, рухається система координат, відносно якої був випущений промінь, і вимірюється його швидкість чи ні. Цей експеримент суперечить принципу відносності Галілея, оскільки за законом додавання швидкостей швидкість об'єкта в іншій інерціальній системі відліку повинна бути меншою, ніж у консервативній системі над нею, тобто системі, яка містить її як підсистему.

Атоми складаються з нейтронів, протонів і електронів. (Колись атом вважали неділимим, отже його названо атомос — неділимий.) У простій планетарній моделі атом представлений ядром, що складається з протонів і нейтронів, навколо якого рухаються електрони по стаціонарних орбітах. Молекули складаються з одного або різних атомів, а речовини є сполуками однієї або різних молекул. Молекула — найменша частинка даної речовини, яка несе її хімічні властивості. Якщо в атомі переважають електрони, то атом називається від'ємно зарядженим, але якщо протонів в атомі більше, ніж електронів, то він позитивно заряджений, і нейтрони не враховуються. Також атом можна називати в цих випадках негативним або позитивним іоном (іоном). Якщо кількість протонів і електронів у атомі рівна, то атом називається нейтральним. Процес перетворення нейтрального атома в іон (позитивний або від'ємний) називається іонізацією.

Плазма — стан речовини, в якому її атоми іонізовані, тобто електрони відірвані від ядер. Плазму називають четвертим агрегатним станом речовини на відміну від твердого, рідкого та газоподібного. Електрони фіксуються з допомогою люмінофорів, ядерних емульсійних пластинок, камер Вільсона. Коли частка пролітає через спеціальну емульсію, вона залишає за собою слід у вигляді дрібних збурень.

Атом водню складається з одного протона, навколо якого рухається один електрон.



Планетарна модель демонструє які частинки є в атомі й те що головна маса атома (ядро) зосереджена в його центрі. Центр атома можна визначити аналізуючи проникність матеріалів.

Браунівський рух — це явище, коли дрібні частинки рухаються в рідині або газі через зіткнення з молекулами середовища. Воно є результатом теплових коливань молекул, роблячи рух частинок випадковим та непередбачуваним. Це явище має значення в фізиці, хімії та біології, допомагаючи підтвердити існування атомів та молекул.

У 1816 році англійський хімік Вільям Праут опублікував дві роботи, в яких вказав, що атомні маси, виміряні для елементів, відомих на той час, виявилися цілими кратними атомній масі водню. Зараз відомо, що відхилення від мас виражених в одиницях атомної маси водню викликані в основному тим, що більшість елементів є сумішшю декількох ізотопів з різними масами та дефектом маси ядер елементів.

Водень (H) - Атомна маса: 1.008, Заряд ядра: +1,
Гелій (He) - Атомна маса: 4.0026, Заряд ядра: +2,
Літій (Li) - Атомна маса: 6.94, Заряд ядра: +3,
Берилій (Be) - Атомна маса: 9.0122, Заряд ядра: +4,
Бор (B) - Атомна маса: 10.81, Заряд ядра: +5,
Вуглець (C) - Атомна маса: 12.01, Заряд ядра: +6,
Кисень (O) - Атомна маса: 16.00, Заряд ядра: +8.

Атомні маси вимірювали за методом, запропонованим італійським вченим Амедео Авогадро (1776 – 1856), який зробив логічний висновок, що при одинакових об’ємі, тиску та температурі кількість атомів у двох ємностях з газами різних речовин буде однакова.

Якщо температура двох газів однакова, то атоми цих газів несуть однакову кількість кінетичної енергії. Якщо збільшити кількість атомів в одному газі, то кінетична енергія кожного окремого атома газу впаде, бо розподілиться між більшою кількістю атомів, але температура в стані рівноваги буде тією ж. Якщо підвищити температуру назад до попередньої, то об’єм газу (або тиск) збільшиться. Дійсно, якщо надути повітряну кулю газом тієї ж температури, то її об’єм зростає пропорційно кількості газу (атомів). Отже, закон Авогадро правильний.

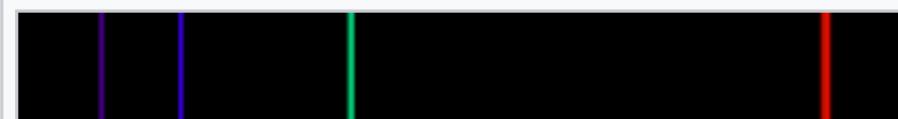
Якщо при одинакових об'ємі, тиску і температурі газ в одній посудині важить 1 кілограм, а в іншій 0,5 кг, то атомна вага елемента в першій посудині буде $1/n$, а в другій — $0,5 / n$, де n — точна кількість атомів, яку важко встановити, хоча можна використовувати відносні значення, наприклад, одиницю виміру як моль.

Цікаво, що коли повітря швидко стискається, його температура підвищується. І навпаки, якщо стиснений газ випустити з посудини, він буде здаватися холоднішим. Простіше кажучи, якщо відкрити балон зі стисненим газом, температура якого дорівнює температурі навколошнього середовища, то газ, що виділяється, буде холоднішим, ніж температура за межами балона. Це пов'язано з тим, що температура навколошнього середовища і температура балону мають тенденцію прийти до температурної рівноваги, як випливає з другого закону термодинаміки. Таким чином, балон зі стисненим повітрям в кінцевому підсумку набуде температуру, рівну температурі навколошнього середовища. Але оскільки повітря в балоні щільніше, енергія, що надходить до одного атома газу, менша за енергію, що надходить до цього газу в тому ж об'ємі за межами балона. Тому при вивільненні стисненого газу він буде холоднішим. Точніше, один куб такого газу, що виділяється, буде холодніше такого ж об'єму газу в кімнаті. За таким принципом працюють холодильники. Виробництво холодильників почалося в кінці XIX століття.

Йозеф фон Фраунгофер помітив, що коли порівняти безперервний спектр Сонця з дискретним спектром парів натрію, тоді жовта лінія, що визначає спектр натрію, на його місці в спектрі, якраз відповідає темній лінії D, яка знаходитьться в спектрі Сонця. Таємничі лінії пояснили в 1860 році відомий вчений Густав Кірхгоф (1824-1887) разом з Робертом Бунзеном (1811-1899). Вивчаючи сонячний спектр, вони пропускали промінь сонячного світла крізь натрієве полум'я, потім розгортали його призмою і відразу з подивом помітили, що лінія D сонячного спектра стала ще темнішою і ширшою. Очевидно, що лінія D має зв'язок з натрієм, очевидно, вона могла виявитися на сонячному спектрі лише тому, що сонячний промінь вже десь раніше проходив через пари натрію, які поглинали його частину. Тому в спектрі була темна лінія D. Отже, пара натрію існує в атмосфері, що оточує Сонце.



Неперервний спектр



Лінійний спектр випромінювання



Лінійний спектр поглинання

Експеримент Гейгера-Марсдена — експеримент, в ході якого було доведено існування атомного ядра — позитивно зарядженої серцевини атома, у якій зосереджена майже вся маса і яка займає крихітну, всього одну мільярдину, частку його об'єму. Альфа-частинка (позитивний іон гелію-4) складається з 2 нейтронів і 2 протонів. В експерименті альфа-частинки, які випромінювали радій, проходили через вузький отвір у свинцевій пластині, з них утворювався добре колімований пучок, який падав на мішень з найтоншої золотої фольги. Перед початком чергової серії спостережень з камери викачували повітря. У ній був покритий сірчистим цинком пересувний екран, що випромінює під ударами альфа-частинок дуже слабкі спалахи світла. Змінюючи положення екрану, можна було реєструвати частинки, що відбилися від мішені під будь-яким кутом. Спалахи спостерігали через віконце у стінці камери за допомогою 50-кратного мікроскопа. Тільки невелика частина альфа-часток відхилялася більш ніж 90°. Більшість частинок летіло прямо через фольгу з незначним відхиленням.

Щоб пояснити цей дивний результат, Резерфорд припустив, що позитивний заряд атома зосереджений у крихітному ядрі у його центрі.

Хоча Ернест Резерфорд зміг зробити деякі емпіричні висновки про структуру атома, найвна
планетарна модель не працює на практиці.

Електрон притягується до протона як ми знаємо. Так чому ж електрон в атомі не падає на його ядро?!
Крім того, згідно з класичною електродинамікою, прискорений електрон генерує змінне електричне поле, що призводить до емісії електромагнітних хвиль (фотонів). Це електромагнітне випромінювання викликає втрату енергії електрона, і, відповідно до класичної теорії, може привести до його падіння на ядро. Данський фізик Нільс Бор запропонував аксіоматичну модель атома з обмеженими орбітами на яких електрон не виділяє енергію.

Квантова механіка пояснює, що електрон не падає на ядро і не виділяє постійно енергію, оскільки електрони в атомі мають квантові енергетичні рівні та знаходяться на стаціонарних орбітах з визначеними енергіями.

Для переходу на вищий рівень потрібно, щоб електрон поглинув фотон. Енергія фотона дорівнює константі Планка (h) помноженій на його частоту (f) і, таким чином, пропорційна його частоті або обернено довжині хвилі (λ).

Спектр випромінювання і спектр поглинання атомів дискретний. (Для речовин та молекул спектр неперервний, суцільний).

Але є нюанс. Що означає переход між орбітами атома?! Якщо є переход між орбітами, то під час цього переходу електрон мав би виділяти енергію неперервно, а не квантами (порціями). Але цього не відбувається під час експериментів, тому вчені ввели поняття “квантовий стрибок між орбітами”. Квантовий стрибок — явище, що властиве самим квантовим системам і відрізняє їх від класичних систем, де будь-які переходи виконуються поступово.

Є ще один нюанс, в яке місце орбіти потрапить електрон?! Якщо квантовий стрибок не має траєкторії, то й невідоме розміщення електрона після стрибка. Також потрібно враховувати хвильову природу електрона (Рівняння Шредінгера та Рівняння Дірака). Висновок, що атом не можна повноцінно представити будь-якою моделлю класичної механіки, якою є планетарна модель Резерфорда і частково модель Бора.

У хімії та атомній фізиці електронну оболонку можна розглядати як орбіту, на якій рухаються електрони навколо ядра атома. Кожна оболонка може містити лише фіксовану кількість електронів: перша оболонка може утримувати до двох електронів, друга оболонка може утримувати до восьми ($2 + 6$) електронів, третя оболонка може утримувати до 18 ($2 + 6 + 10$) і так далі. Загальна формула полягає в тому, що n оболонка в принципі може утримувати до $2(n^2)$ електронів.

Дослід Девіссона — Джермера — фізичний експеримент з дифракції електронів, проведений Кліntonом Девіссоном та Лестером Джермером у 1927 році, що підтверджує гіпотезу Луї де Бройля, за якою матеріальні частинки окрім корпускулярних мають також і хвильові властивості.

Клінт Девіссон і Лестер Джермер помітили, що коли електрони, що прискорюються, ударяються об поверхню нікелю, виникають максимуми інтенсивності, які не можна пояснити, розглядаючи електрон як частинку, що стикається з поверхнею, заповненою сферичними атомами нікелю, які повинні були б розсіювати електрони у всіх напрямках. Однак явище, що спостерігалося, було схоже на дифракцію рентгенівських променів на кристалічній поверхні, відкрите в 1912 році німецьким фізиком Максом фон Лауе з його співробітниками.

Фотоефект — це випромінювання електронів матеріалом під дією електромагнітного випромінювання (світла).

Експериментальні результати не узгоджуються з класичним електромагнетизмом, який передбачає, що безперервні світлові хвилі передають енергію електронам, які потім випромінюються, коли накопичують достатньо енергії. Натомість експериментальні результати показують, що електрони вивільняються лише тоді, коли світло перевищує певну частоту — незалежно від інтенсивності світла чи тривалості впливу.

Генріх Герц встановив, що заряджений провідник, освітлений ультрафіолетовим промінням, швидко втрачає свій заряд, а електрична іскра виникає в іскровому проміжку при меншій різниці потенціалів. Помічене явище було описане Герцом в його статтях 1887—1888 років, але залишилося без пояснення.

Повне пояснення фотоефекту належить Альберту Айнштайну, який використав ідею німецького фізика Макса Планка про те, що світло випромінюється і поширюється окремими порціями — квантами, які отримали назву фотонів. Для обчислення енергії кванта світла Макс Планк запропонував просту формулу:

Енергія фотона прямо пропорційна частоті

$$E = hf$$

де

E — це енергія (звичай у джоулях),

h — стала Планка,

f — це частота (звичай у герцах),

Крім того,

$$E = hc/\lambda,$$

де

E — енергія фотона,

λ — довжина хвилі фотона,

c — швидкість світла у вакуумі,

h — стала Планка.

Айнштайн висловив припущення, що фотоефект відбувається внаслідок поглинання електроном одного кванта випромінювання, а інші кванти не можуть брати участь у цьому процесі. Тоді енергія одного кванта світла (фотона) витрачається на подолання бар'єру (виконання роботи виходу, відризу від матеріалу) і надання кінетичної енергії фотоелектрону.

Ефект Комптона за природою є подібним до фотоефекту — різниця полягає в тому, що при фотоефекті фотон повністю поглинається електроном, тоді як при комптонівському розсіюванні він лише змінює напрямок руху й енергію.

Явище непружного розсіяння рентгенівських і гамма-променів на електронах відкрив 1923 року Артур Комптон, за що отримав Нобелівську премію за 1927 рік. Важливість відкриття зумовлена тим, що в класичній фізиці зміна довжини електромагнітної хвилі при розсіюванні на вільній зарядженій частинці неможлива.

Експерименти з фотоном на інтерферометрі Маха-Цендера та експерименти з двома щілинами показали, що фотон ніби інтерферує сам з собою, але при цьому детектується як частинка.

У подібних експериментах фотони проявляють явища інтерференції, які звичай характерні для хвиль, такі як інтерференційні смуги на детекторі. Однак, коли спробувати відслідкувати шлях фотона, він проявляє властивості частинки, бо детектор зафіксує його на одному конкретному місці. Цей парадокс виникає через квантову природу частинок, де їхнє становище може бути описане як ймовірність знаходження у певному місці (хвильова функція), але сам факт спостереження може "закріпити" їхнє положення. Це фундаментальний аспект квантової механіки, який ще залишається предметом активних досліджень та дискусій.

Атом водню (гідрогену) є нейтральним, бо заряд електрона компенсує заряд протона. Звичайний стан речовини в природі є нейтральним, тобто у речовині майже стільки ж протонів, скільки в ній електронів. Речовина може бути заряджена позитивно або негативно шляхом віднімання або додавання певної кількості носіїв заряду, тобто речовину можна іонізувати. Кожне тіло може бути заряджене до певного рівня, який називається електричною ємністю тіла. Від'ємний заряд (електрони) може накопичуватися в тілі до великих значень (див. генератор Ван де Граафа), а загальний позитивний заряд залежить від кількості протонів в тілі. Очевидно, що позитивна ємність тіла залежить від кількості протонів в ньому. Ємність визначається, як відношення заряду тіла Q до різниці потенціалів U : Здебільшого ємність позначається латинською літерою C й вимірюється у фарадах.

Експерименти показують, що електричний заряд буде розташований на поверхні кулі або циліндуру. Це пов'язано з властивістю електрона відштовхувати собі подібних.

Якщо речовина переважно складається, скажімо, з протонів, чому вона не розпадається, оскільки протони відштовхуються? Це повинно статися з протонами, але речовина все одно не розпадається. Це через те, що атоми мають певні зв'язки один з одним, які формують їх у структуру, яку називають кристалічною решіткою (ґраткою).

Цікаво, що алмаз і графіт складаються лише з одного хімічного елемента (атома) - вуглецю, але мають абсолютно різні властивості. Графіт чорний і добре проводить електрику, алмаз прозорий і є ізолятором. Як може один хімічний елемент мати такі різні властивості? Відповідь полягає в тому, що розташування цих елементів у матеріалі є різним, тобто графіт і алмаз, хоча складаються з одного і того ж хімічного елемента, все ж мають різну кристалічну решітку. У кристалографії кристалічна ґратка (решітка) — це опис упорядкованого розташування атомів, іонів або молекул у кристалічному матеріалі.

Алотропія — це властивість деяких хімічних елементів існувати у двох або більше формах, у тому ж фізичному стані, відомих як алотропи елементів. Алмаз і графіт — це два алотропи вуглецю: чисті форми одного й того ж елемента, які відрізняються кристалічною структурою.

П'єзоелектрика — це електричний заряд, який накопичується в певних твердих матеріалах, наприклад, кристалах, у відповідь на прикладене механічне навантаження. Французькі фізики Жак і П'єр Кюрі (1859 — 1906) відкрили п'єзоелектрику в 1880 році.

Кварцовий резонатор — п'єзоелектричний резонатор, основною складовою частиною якого є кристалічний елемент з кварцу (SiO_2).

На пластинку, вирізану з кристала кварцу належним чином, наносять 2 і більше електродів — провідних смужок. При подачі напруги на електроди завдяки зворотному п'єзоелектричному ефекту відбувається згинання, стиснення або зсув залежно від того, яким чином вирізаний кристал щодо кристалографічних осей, конфігурації збудливих електродів і розташування точок кріплення.

Власні коливання кристала внаслідок п'єзоелектричного ефекту наводять на електродах додаткову ЕРС (Електрорушійна сила) і тому кварцовий резонатор електрично поводиться подібно до резонансного ланцюга, — коливального контуру, складеного з конденсаторів, індуктивності та резистора.

Якби не було молекулярних зв'язків, протони розлетілися б, і речовина розпалася б. Кристалічна решітка складається з атомів, об'єднаних ковалентними зв'язками, тобто вони мають спільній електрон (але є інші типи зв'язків, наприклад, йонні). Один зі способів взаємодії атомів називається ковалентним зв'язком (див. Август Кекуле й бензольні кільця). Це зв'язок, що формується за участю спільнії пари електронів — по одному від кожного з двох атомів. Ми можемо припустити, що електрони цієї пари будуть належати обом атомам. Також існує йонний зв'язок (див. Сванте Арреніус) між атомами, який являє собою електростатичне притягування заряджених частинок. Згідно з законом Кулона, електричні заряди одного знака відштовхуються, а протилежні — притягуються. Тому позитивно заряджена частинка (катіон) і негативно заряджена (аніон) притягуються одне до одного. Аніон — це негативно заряджений іон. Кухонна сіль NaCl (натрій хлорид) є прикладом йонного зв'язку, де Na є катіоном, а Cl^- — аніоном. Кристалічні решітки можуть мати різні форми в залежності від атомів. Електромагнітні сили діють між атомами. Між частинами атома діють електромагнітні та ядерні сили. Кожна речовина здатна проводити електричний струм, але залежно від властивостей цієї провідності матеріали розділяють на провідники та діелектрики (ізолятори). Здатність проводити

електричний струм означає здатність пропускати потік електронів через себе. Електричний струм починає протікати, коли є тіло з надлишком електронів, тобто негативно заряджене тіло, і тіло з нестачею електронів, тобто позитивно заряджене тіло, між якими прокладено провідник, тобто вони з'єднані. Ми говоримо, що такі тіла мають різницю потенціалів, яка буде компенсована потоком, встановиться в рівновагу, коли між тілами прокладено провідник. Щоб досягти різниці потенціалів, тіло трутъ (трибоелектрика), метали поміщають в кислоту (електроліз), в магнітне поле (електромагнітна індукція). Провідники мають лінійний графік збільшення струму зі зростанням різниці потенціалів, діелектрики мають нелінійний графік, який показує зростання струму зі значною різницею потенціалів.

Провідник добре проводить струм, оскільки в ньому, точніше, в його атомах, є багато електронів, які легко відкріпляються від своїх орбіт і переходят у вільне плавання, в той час, як в діелектриках всі електрони твердо закріплені на орбітах навколо ядра і важко відкріпити їх, оскільки для цього потрібна відносно велика напруга, яка залежить від різниці потенціалів. (Напругу між двома електрично зарядженими тілами вимірюють у вольтах.) Таким чином, ми кажемо, що діелектрики мають великий опір, хоча в кожного провідника також є той чи інший опір (для протікання струму). (Електричний опір речовини вимірюють в омах). В нормальніх умовах не існує абсолютноних провідників або абсолютної ізоляторів. Зазвичай опір провідника збільшується при нагріванні, але для деяких надпровідників та електролітів навпаки, опір зменшується зі збільшенням температури. Сила струму залежить від напруги та опору провідника. Сила струму I в ділянці електричного кола визначається як кількість електричних зарядів, які проходять за одиницю часу (1 секунду) через поперечний переріз провідника. (Сила струму вимірюється в амперах). Закон Ома ($V = IR$): Сила струму (I) на однорідній ділянці кола прямо пропорційна напрузі (V) на кінцях цієї ділянки та обернено пропорційна її опору (R).

Миттєвий струм (або сила струму) визначається за формулою:

$$I = \lim(\Delta t \rightarrow 0) \frac{\Delta q}{\Delta t} = dq/dt.$$

Δq - це кількість електричного заряду, яка протікає через деякий переріз провідника за час Δt .

Магнітне поле створюється рухомими зарядами (проводник зі струмом), електростатичне — стаціонарними зарядами. Магнітне поле є вихоровим (сили магнітного поля закриті), електростатичне — потенціальним (сили електростатичного поля відкриті). Для вивчення електричного поля можна приклейти металеві смужки фольги навколо електрично зарядженого тіла, які набудуть форми ліній поля. Для вивчення магнітного поля навколо магніту чи електричної котушки розсипають металеву стружку.

Для вивчення напрямку магнітного поля можна взяти певну кількість залізної стружки, розсипати її на папір, а потім провести під папером магніт. Стружка набуде форми магнітного поля.

Навіть якщо розбити магніт на дві частини, обидві частини матимуть два полюси. Магніт має два полюси, північний і південний, вони відрізняються своїми властивостями. Протилежні полюси притягаються, а однакові відштовхуються. Магніт, підвішений на нитці, розміститься у просторі так, що вказуватиме на північ і південь Землі (це вже знав П'єр Пелерин де Марікур в 13 столітті). Магніт притягує лише деякі тіла, електрика діє на всі речовини. Гравітація (сила тяжіння) також діє на всі тіла (навіть на фотони світла). Магнітне поле може змінювати поляризацію світла (Ефект Фарадея). Магніти притягають тіла, виготовлені з певних металів, таких як сталь. Проте мідь, золото, срібло, алюміній та свинець не реагують на магніт.

Для створення магнітів беруть матеріали: залізо, неодим, нікель, бор, кобальт, самарій, титан, диспрозій, тербій та їх суміші. Створюється форма, в яку заливають розплавлений метал. Цю форму розміщують в сильному магнітному полі (створеному електричними котушками). Метал твердішає (вистигає) перебуваючи під впливом цього поля, а його частинки вирівнюють свій магнітний момент таким чином, щоб відповідати зовнішньому магнітному полю. Відома так звана точка Кюрі (на честь П'єра Кюрі). Точка Кюрі — це значення температури, при якій магніт втрачає свої властивості.

Діамагнетики не несуть магнітного поля і дуже слабо реагують на нього. Парамагнетики — це речовини, які можуть бути намагнічені. Феромагнетики, від слова ферум (залізо) — це речовини, які є постійними магнітами.

Для створення магніту, метал розміщують в статичному магнітному полі, якщо ж поле динамічне (змінне), тоді метал починає нагріватися й плавитися (як в мікрохвильовці).

Котушка створює змінне магнітне поле високої частоти, що проникає в метал. Магнітне поле спричиняє вихрові струми, кругові електричні струми всередині металу з допомогою електромагнітної індукції. Вихрові струми, що протікають через метал з певним електричним опором, нагрівають його відповідно до закону Джоуля про нагрівання провідника (він же Закон Джоуля — Ленца). Закон Джоуля про нагрівання провідника говорить: “Коли електричний струм поширюється вздовж провідника, тепло, що виділяється за певний час, пропорційне опору провідника, помноженому на квадрат електричної інтенсивності (ампераж)”. Тобто приріст тепла (Q) буде значно більшим ніж приріст струму (I), а саме кількість тепла буде більша від сили струму в 1 разів. У своїй книзі “Про нові магнітні дії” (1846) Фарадей описує такі магнітні та діамагнітні метали: Магнітні – Залізо (ферум, Fe), Нікель (Ni), Кобальт (Co), Марганець (Mn), Хром (Cr), Церій (Ce), Титан (Ti), Паладій (Pd), Платина (Pt), Осмій (Os). Діамагнітні – Вісмут (Бісмут, Bi), Сурма (Стибій, Sb), Цинк (Zn), Олово (Sn), Кадмій (Cd), Натрій (Na), Ртуть (Hg), Свинець (Pb), Срібло (Ag), Мідь (Cu), Золото (Au), Арсен (Миш’як, As), Уран (U), Родій (Rh), Іридій (Ir), Вольфрам (W). Діамагнітні матеріали, які відштовхуються магнітним полем, тобто прикладене магнітне поле створює в них наведене магнітне поле в протилежному напрямку, викликаючи силу відштовхування. Навпаки, парамагнітні та феромагнітні матеріали притягаються магнітним полем.

Отже, електричний струм — це потік електронів.

Електричний струм може текти через будь-які предмети, навіть через вакуум. Все залежить від напруги. При великих напругах виникає пробій діелектрика та електронна лавина. Зокрема, це спостерігається в блискавці.

Електрон вважається фундаментальною частинкою матерії.

Саме слово "електрон" та "електрика" походить від давньогрецького слова, що означало бурштин. Древні греки помітили, що бурштин (по їхньому електрон) має властивість притягувати предмети коли його потреш хутром тощо, також після цього він пускає дрібні іскри в темряві. Бурштин є діелектриком, який можна електризувати тлінням та електростатичною індукцією.

З електричними властивостями пов'язували притягування предметів та іскри.

В 18 столітті виявили, що блискавка це також електричне явище, коли змогли зарядити нею Лейденську банку (конденсатор).

Всі матеріали складаються з фундаментальних частинок. Електрони вже не означають бурштин, а вказують на фундаментальні частинки, які мають навколо себе електричне поле та під час свого руху генерують магнітне поле.

Але електричні явища не пов'язані тільки з електронами, бо існують протони, які значно більші ніж електрон, але також володіють електричним зарядом, проте з протилежним знаком.

Протон відштовхується від протона, але притягується до електрона.

Відкриття електрона як частинки належить Емілю Йоганну Віхерту (1861 – 1928) та Джозефу Томсону, який у 1897 році встановив, що відношення заряду до маси для катодних променів не залежить від вихідного матеріалу. Джордж Джонстон Стоуні (1826—1911) — ірландський фізик. Він найбільш відомий тим, що ввів термін електрон як “основну одиничну кількість електрики”. Електрони фіксуються з допомогою люмінофорів, ядерних емульсійних пластинок, камер Вільсона (в яких пар конденсується під дією частинок, які пролітають крізь нього залишаючи треки).

Протони формують ядро атома, як експериментально виявив Резерфорд (бомбардуючи матеріал електронами), а електрони рухаються навколо цього ядра.

Як виявили Бенджамін Франклін та Шарль Кулон: Однокомпонентні заряди відштовхуються, а різноміненні притягаються.

В часи Франкліна та Кулона електричний заряд генерувати трибоелектрикою, заряджаючи діелектрик тертям, а потім переносячи цей заряд в Лейденську банку або на скляну чи металеву кулю. Сферичні поверхні мають здатність накопичувати велику кількість електричного заряду.

Генератор Ван де Граафа може генерувати тисячі вольтів електрики з допомогою тертя і властивостей електронів розміщуватися на поверхні сфери.

Було виявлено (дослід Штерна-Герлаха), що в електронів є квантовий магнітний момент. Напівцілий спін електрона — це квантова властивість електрона, що вказує на його спрямований магнітний момент. Електрон може мати тільки спін "вгору" або "вниз", якщо спін спрямований уздовж або протилежно напрямку зовнішнього магнітного поля. Зверніть увагу, класичний магніт може мати різний напрямок магнітних моментів, про те в електрона він квантується.

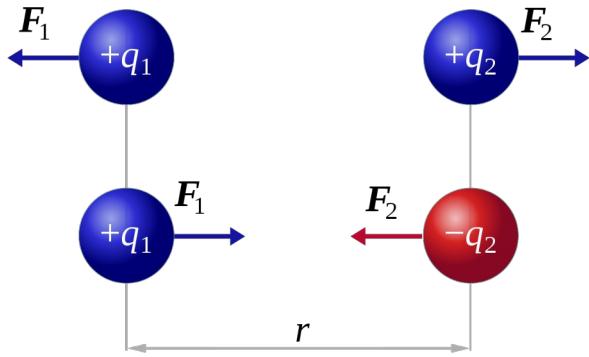
Електрон та протон для простоти можна уявляти як дрібну частинку (кульку). Але нині відомо, що протони та нейтрони мають корпускулярно-хвильовий дуалізм. Тобто в деяких випадках поводяться як частинки, а в деяких як хвилі. Проте, нині вчені мають консенсус, що на фундаментальному рівні все є хвильами в певних полях (Квантова теорія поля).

Найпростіші крутільні ваги були винайдені Шарлем Кулоном і застосовані ним для вивчення взаємодії (точкових) електричних зарядів, а також для вивчення взаємодії магнітних полюсів. Пристрій складався зі скляного циліндра, по окружності якого була нанесена кутова шкала. У центрі кришки циліндра був отвір, в який вставлялася скляна трубка. Через трубку в циліндр пропускали срібну нитку, на якій підвішували легкий скляний важіль, на кінці якого була кулька. Шарль Кулон ввів стрижень з кулькою на кінці через бічний отвір в кришці циліндра. Якщо обидві кульки були електрично заряджені, сили, що виникають у результаті їх взаємодії, призводили до обертання срібної нитки. Кут повороту визначали з допомогою кутової шкали. Якщо одна кулька була заряджена, а інша — ні, то вони притягувалися електростатичною індукцією. Якщо дві кулі зарядили одинаковим зарядом, вони відштовхнулися. Електростатична індукція — це перерозподіл електричного заряду в об'єкті, викликаний впливом поруч зарядів.

Між двома зарядами у спокої (достатньо малими, щоб знехтувати їх формою і вважати заряди точковими) діє сила, прямо пропорційна добутку зарядів і обернено пропорційна квадрату відстані між ними, що відштовхує заряди, якщо вони однакового знака або притягує, якщо вони протилежного знака. Носіями заряду є електрон (-) і протон (+).

Закон Кулона можна записати за формулою:

$$F = e_1 e_2 / r^2, \text{ де } e_1, e_2 \text{ — точкові заряди, } F \text{ — сила, що діє між ними, } r \text{ — відстань між ними.}$$



$$|F_1| = |F_2| = k_e \frac{|q_1 \times q_2|}{r^2}$$

При практичних розрахунках точковий заряд — це заряд, розмірами носія якого в порівнянні з відстанню, на якому розглядається електростатична взаємодія, можна знехтувати. Іноді також визначається як електрично заряджена матеріальна точка. Матеріальна точка (частинка) — це фізична модель, яку використовують замість тіла, розмірами якого в умовах даної задачі можна знехтувати. Саме для точкових зарядів сформульований закон Кулона.

Експеримент Толмена Стюарта (1916) продемонстрував, що вільні заряди в металі мають негативний заряд, і забезпечив кількісне вимірювання відношення їх заряду до маси (q/m). Експеримент полягав у різкій зупинці котушкі дроту, що швидко обертася, і вимірюванні різниці потенціалів, яка утворювалася між кінцями дроту. У провідному тілі, яке зазнає прискореного руху, інерція змушує електрони в тілі відставати від загального руху. При лінійному прискоренні на кінці тіла накопичується негативний заряд; тоді як для обертання негативний заряд накопичується на зовнішньому ободі. Накопичення зарядів можна виміряти гальванометром.

Експеримент з краплею олії був проведений Робертом Міллікеном і Гарві Флетчером у 1909 році для вимірювання елементарного електричного заряду (заряду електрона). Експеримент полягав у спостереженні крихітних електрично заряджених крапельок олії, розташованих між двома паралельними металевими поверхнями, які утворюють пластини конденсатора. Пластини були орієнтовані горизонтально, одна пластина над іншою. Туман розпилених крапель олії був введений через невеликий отвір у верхній пластині та іонізований рентгенівським випромінюванням, роблячи їх негативно зарядженими. Спочатку при нульовому прикладеному електричному полі виміряли швидкістьпадаючої краплі. Оскільки обидві сили по-різному залежать від радіуса краплі, а отже, масу та силу тяжіння можна визначити (використовуючи відому густину олії). Далі між пластинами прикладали напругу, що індукує електричне поле, і регулювали до тих пір, поки краплі не перебували в механічній рівновазі, що вказувало на те, що електрична сила та сила тяжіння були в рівновазі. Використовуючи відоме електричне поле, Міллікен і Флетчер могли визначити заряд на краплі олії.

Повторивши експеримент для багатьох крапель, вони підтвердили, що всі заряди були маленькими цілими, кратними певному базовому значенню, яке було виявлено $1,5924(17) \times 10^{-19}$ C, близько 0,6% різниці від прийнятого зараз значення $1,602176634 \times 10^{-19}$ C.

Кулон: $C (Кл) = \text{const}$ (через стандарт), $C = A * c$.

Один кулон дорівнює кількості електричного заряду, що проходить через поперечний переріз провідника при силі струму один ампер за секунду.

Елементарний електричний заряд — це фундаментальна фізична константа, мінімальна частина (квант) електричного заряду, що спостерігається в природі у вільних довгоживучих частинках. Відповідно до змін у визначеннях базових одиниць СІ, у Міжнародній системі одиниць (СІ) це рівно $1.602\ 176\ 634 \cdot 10^{-19}$ C в Міжнародній системі одиниць (СІ).

Ампер: $A = Кл / с$.

Якщо сила струму в провіднику 1 ампер, то за одну секунду через поперечний переріз проходить заряд в 1 кулон.

Сила струму в частині електричного кола визначається як кількість електричного заряду, який проходить через переріз провідника за одиницю часу. Сила струму вимірюється в амперах, який рівний протіканню заряду в один кулон за одну секунду крізь поперечний переріз частини провідника.

Міліампер — одиниця виміру напруги електричного струму, що дорівнює тисячній частині ампера. Зазвичай струм від 10 до 20 mA (міліампер), що протікає через тіло людини протягом 1-2 секунд, може викликати фібриляцію шлуночків серця і призвести до смерті.

Німецький фізик Георг Ом (1789 – 1854) повернувся до ідеї Шарля Кулона і спроектував крутильні ваги. Точним і чутливим гальванометром виявилася підвішена на еластичній нитці магнітна стрілка. У перших дослідах, результати яких були опубліковані Омом у 1825 році, спостерігалася “втрата сили” (зменшення кута відхилення стрілки) зі збільшенням довжини провідника, підключенного до полюсів вольтового стовпа (перетин провідника був постійним). Оскільки одиниця вимірювання не було, довелося вибрати еталон – “стандартний дріт”. Як залежна змінна спостерігалося зменшення сили, що діє на магнітну стрілку. Досліди показали природне зменшення цієї сили зі збільшенням довжини провідника. У творі “Визначення закону, за яким метали проводять контактну електрику” (1826) Георг Ом формулює свій знаменитий закон Ома, а потім об’єднує всі свої роботи з цього питання в книзі “Гальванічний ланцюг, розроблений математично”, в якій він дає теоретичний висновок його закону, заснованого на теорії, подібної до теорії теплопровідності Фур'є. Якщо провідник струму накрити шматочками фольги, то можна побачити, що шматки фольги на початку провідника сильно відхиляються один від одного, на кінці провідника вони відхиляються набагато слабше, що свідчить про падіння струму в провіднику. Зазвичай, зі збільшенням температури, опір провідника збільшується.

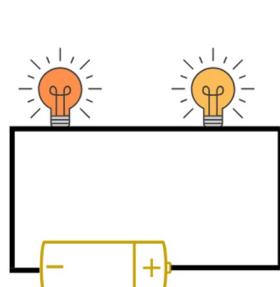
Знаючи опір одного метра кабелю (дроту), можна визначити, скільки кабелю на котушці, не вимірюючи лінійкою і не розмотуючи котушку, а лише вимірявши опір всього кабелю на котушці.

Закон Ома для постійного струму.

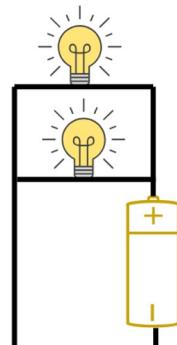
Струм, виміряний між двома кінцями провідника, прямо пропорційний напрузі між цими кінцями та обернено пропорційний опору цього провідника.

Закон Ома можна записати за формулою:

$I = U / R$, де I - сила струму, U - напруга, R — опір. Опір вимірюється в омах.



Послідовне з'єднання провідників



Паралельне з'єднання

Коли провідники з'єднані послідовно, сила струму в будь-якій частині кола однакова: $I = I_1 = I_2 = \dots = I_n$ (оскільки сила струму визначається кількістю електронів, що проходять через поперечний переріз провідника, а якщо в ланцюзі немає вузлів, то всі електрони в ньому будуть текти по одному провіднику).

Загальна напруга в ланцюзі при послідовному з'єднанні, або напруга на полюсах джерела живлення, дорівнює сумі напруг на окремих ділянках кола: $U = U_1 + U_2 + \dots + U_n$.

Загальний опір в колі при послідовному з'єднанні резисторів дорівнює сумі опорів в окремих ділянках кола: $R = R_1 + R_2 + \dots + R_n$.

Сила струму в нерозгалуженій частині кола дорівнює сумі струмів в окремих паралельно з'єднаних провідниках: $I = I_1 + I_2 + \dots + I_n$.

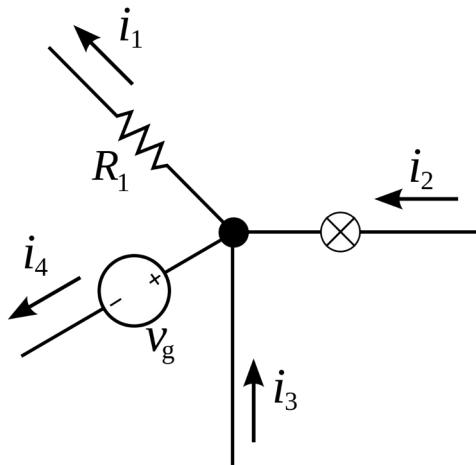
Напруга на ділянках кола і на кінцях всіх паралельно з'єднаних провідників однакова: $U = U_1 = U_2 = \dots = U_n$.

Для резисторів, з'єднаних паралельно, їх загальний опір дорівнює: $R = (R_1 R_2 \dots R_n) / (R_1 + R_2 \dots + R_n)$.

Коли резистори з'єднані паралельно, їх загальний опір буде менше найменшого опору в ланцюзі.

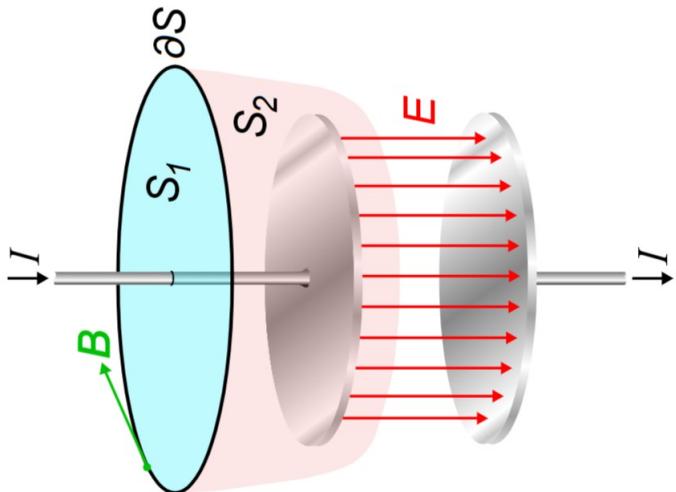
Провідники, які чинять опір, нагріваються за законом Джоуля-Ленца. Застосувавши закон Ома для ділянки кола, закон Джоуля-Ленца можна записати як: $Q = I^2 R t$, де Q – кількість теплоти, I – сила струму, R – опір провідника, t — час проходження струму. Слід зауважити, що опір провідника збільшується зі збільшенням його температури, але для напівпровідників та електролітів все навпаки.

Перше правило Кірхгофа говорить, що скільки струму втікає у вузол, стільки з нього і витікає. Однак при розрахунках слід враховувати, що це правило застосовується лише у випадку дуже малої ємності вузла (або не враховуючи її). При цьому спрямований до вузла струм прийнято вважати позитивним, а спрямований від вузла негативним, тобто їх сума дорівнює нулю. Якщо два проводи з низьким опором під'єднати до одного проводу з високим опором, то струм який надходить від двох проводів в точку з'єднання (вузол) буде дорівнює струму який виходить з вузла по одному проводу з високим опором. Правило сформульоване Густавом Кірхгофом у 1845 році.



Струм, що входить до будь-якого переходу, дорівнює струму, що виходить із цього переходу. $i_2 + i_3 = i_1 + i_4$. Це правило можна продемонструвати на прикладі доріг, а саме: скільки б доріг не вело в одну точку, якщо з точки виходить тільки одна дорога, то потік руху буде залежати саме від швидкості руху на цій дорозі.

Приклад, що показує дві поверхні S_1 і S_2 , які мають спільний обмежувальний контур ∂S . Однак S_1 пронизує струм провідності, тоді як S_2 пронизує *струм зміщення*. Поверхня S_2 закрита під пластинкою конденсатора.



Джеймс Максвелл опублікував свою двотомну роботу про електрику “Трактат про електрику і магнетизм” у 1873 році. У цьому трактаті Максвелл дав уніфіковану (класичну) теорію електрики та магнетизму, поєднуючи попередні розробки. Німецький фізик Генріх Герц (1857 — 1894) високо оцінив трактат Максвелла і сформулював його основні рівняння в сучасній формі. Максвелл з допомогою своєї теорії зробив ряд припущень, які були експериментально підтвердженні (в дослідах Герца, 1888), а саме, про існування струму зміщення, про існування електромагнітних хвиль.

Векторний добуток — білінійна, антисиметрична операція на векторах у тривимірному просторі. На відміну від скалярного добутку векторів евклідового простору, результатом векторного добутку є вектор, а не скаляр.

В рівняннях Максвелла: x – векторний добуток, $*$ - скалярний добуток. Жирним шрифтом виділені векторні величини.

Рівняння Максвелла в диференціальній формі (в системі МКС - метрична система одиниць):

$$1) \nabla^* \mathbf{E} = 4\pi\rho,$$

Електричне поле, що відповідає будь-якому розподілу заряду, визначається із закону Кулона.

Густота заряду визначається як:

$$\rho = \lim_{\Delta V \rightarrow 0} (\Delta q / \Delta V),$$

де Δq електричний заряд в об'ємі ΔV .

2) $\nabla^* \mathbf{B} = 0$.

Магнітних зарядів не існує.

3) $\nabla \times \mathbf{E} = -(1/c) * (\partial \mathbf{B} / \partial t)$.

Закон індукції Фарадея говорить: Для будь-якого замкнутого контуру індукована електрорушійна сила (ЕРС) дорівнює швидкості зміни магнітного потоку, що проходить через цілий контур, взятого зі знаком "мінус". Або іншими словами: Генерована ЕРС пропорційна швидкості зміни магнітного потоку.

4) $\nabla \times \mathbf{B} = (4\pi/c)\mathbf{j} + ((1/c) * (\partial \mathbf{E} / \partial t))$,

Закон Ампера.

5) $F = q\mathbf{E} + q * (\mathbf{v}/c \times \mathbf{B})$.

Електромагнітна сила (Сила Лоренца), що діє на заряд q , являє собою комбінацію сили, що діє в напрямку електричного поля \mathbf{E} , пропорційної величині поля і кількості заряду, і сили, що діє під прямим кутом до магнітного поля \mathbf{B} і швидкості, пропорційної величині магнітного поля, заряду і швидкості.

Тут

\mathbf{E} — напруженість електричного поля,

\mathbf{B} — вектор магнітної індукції,

ρ — густота електричного заряду,

\mathbf{j} — густота електричного струму,

c — швидкість світла,

$\pi = 3.1415\dots$

∇ — диференціальний оператор Гамільтона або оператор набла, тоді як:

$\nabla \times \mathbf{E} \equiv \text{rot}(\mathbf{E})$;

$\nabla^* \mathbf{E} \equiv \text{div}(\mathbf{E})$;

Дивергенція (div) у векторному аналізі, яку часто позначають як "div", є фундаментальною операцією, що застосовується до векторних полів у математиці та фізиці. Дивергенція поля є мірою того, наскільки інтенсивно вектори поля "виходять" з даної точки простору. Чим більше дивергенція, тим більше "виходить" векторів з цієї точки (тобто точка є джерелом, а для негативних зарядів - стоком).

Векторне поле (\mathbf{F}) — функція, або відображення, яке кожній точці даного простору ставить у відповідність вектор. Функція приймає радіус-вектор точки й повертає вектор конкретного поля.

"Радіус-вектор" - це вектор, який починається в початковій точці координат і закінчується в точці, яка відповідає положенню об'єкта. У тривимірному просторі радіус-вектор може бути представлений як (x, y, z) , де x, y і z - це координати об'єкта по відповідних осях.

Математично розбіжність векторного поля \mathbf{F} у трьох вимірах обчислюється за такою формулою:

$$\text{div}(\mathbf{F}) = \nabla \cdot \mathbf{F} = \partial F_x / \partial x + \partial F_y / \partial y + \partial F_z / \partial z$$

Тут ∂ представляє часткові похідні, а ∇ (оператор Гамільтона) є векторним оператором, який

використовується для обчислення різних похідних у векторному аналізі. Похідні можна знайти методом скінчених різниць (FDM).

Інтерпретація:

1. Якщо $\operatorname{div}(F) > 0$ у точці, це означає, що векторне поле поширюється від цієї точки.
2. Якщо $\operatorname{div}(F) < 0$, поле "стікає" або "всмоктується" в цю точку.
3. Якщо $\operatorname{div}(F) = 0$, поле не має ні джерела, ні "стоку" в цій точці.

У векторному аналізі "ротор" — це оператор, який застосовується до векторного поля, у результаті чого утворюється інше векторне поле. Ротор векторного поля відображає, як поле "циркулює" або "обертается" в даній точці простору. Він позначається як "rot, curl" або $\nabla \times F$ і є важливим поняттям у векторному численні.

Математично, ротор ($\nabla \times F$) векторного поля F у трьох вимірах обчислюється таким чином:

$$\nabla \times F = (\partial F_z / \partial y - \partial F_y / \partial z, \partial F_x / \partial z - \partial F_z / \partial x, \partial F_y / \partial x - \partial F_x / \partial y);$$

Ось що представляють компоненти вектора rot:

1. $(\partial F_z / \partial y - \partial F_y / \partial z)$: цей компонент представляє тенденцію векторного поля до "скручування" або обертання навколо осі x .
2. $(\partial F_x / \partial z - \partial F_z / \partial x)$: цей компонент представляє тенденцію обертання поля навколо осі y .
3. $(\partial F_y / \partial x - \partial F_x / \partial y)$: цей компонент представляє тенденцію обертання поля навколо осі z .

Інтерпретація:

- Якщо ротор векторного поля дорівнює нулю ($\nabla \times F = 0$) у точці, тоді це вказує на відсутність обертання чи циркуляції.
- Якщо ротор не дорівнює нулю, це означає, що в цій точці відбувається обертання або циркуляція, а напрямок вектора ротора вказує вздовж осі обертання.

Концепція ротора є важливою у фізиці, зокрема в електромагнетизмі, динаміці рідини та вивчені обертальної поведінки у векторних полях. Це допомагає описати такі явища, як циркуляція рідини навколо вихору, поведінка магнітних полів навколо струму тощо.

Векторне поле — це область простору, кожній точці якого присвоєно значення певного вектора.

Скалярне поле — це область простору, кожній точці якої присвоєно значення скаляра.

Скаляр — величина, яка повністю визначається в будь-якій системі координат одним числом або функцією, яка не змінюється при зміні просторової системи координат. Функція, яка приймає скалярні значення, називається скалярною функцією.

Скаляр завжди описується одним числом, а вектор може бути описаний двома чи більше числами. Оскільки кожна точка поля визначається своїм радіус-вектором r , визначення векторного поля еквівалентно визначенню деякої вектор-функції $a(r)$, а у випадку скалярного поля воно еквівалентно скалярній функції $\phi(r)$.

Радіус-вектор — вектор, який визначає положення точки в просторі відносно деякої задалегідь заданої точки, яка називається початком координат.

Розглянемо скалярне поле функції $\phi(r) = \phi(x, y, z)$. Таким полем є, наприклад, температурне поле нерівномірно нагрітого тіла. Відповідно, функція $\phi(x, y, z)$ поверне температуру в певній точці простору (поля) нагрітого тіла, яка задана координатами x, y, z .

Нехай функція ϕ в точці r має значення q . Припустимо, що з точки r ми рухаємося вздовж деякого вектора s , тобто підставляємо точки вектора s у функцію $\phi(x, y, z)$. Таким чином, значення $\phi(x, y, z)$ може змінюватися. Позначимо зміну значення $\phi(x, y, z)$ від однієї точки до наступної як $d\phi = \phi(x + dx, y + dy, z + dz) - \phi(x, y, z)$ відповідно dx, dy, dz позначають зміни координат від однієї точки до іншої

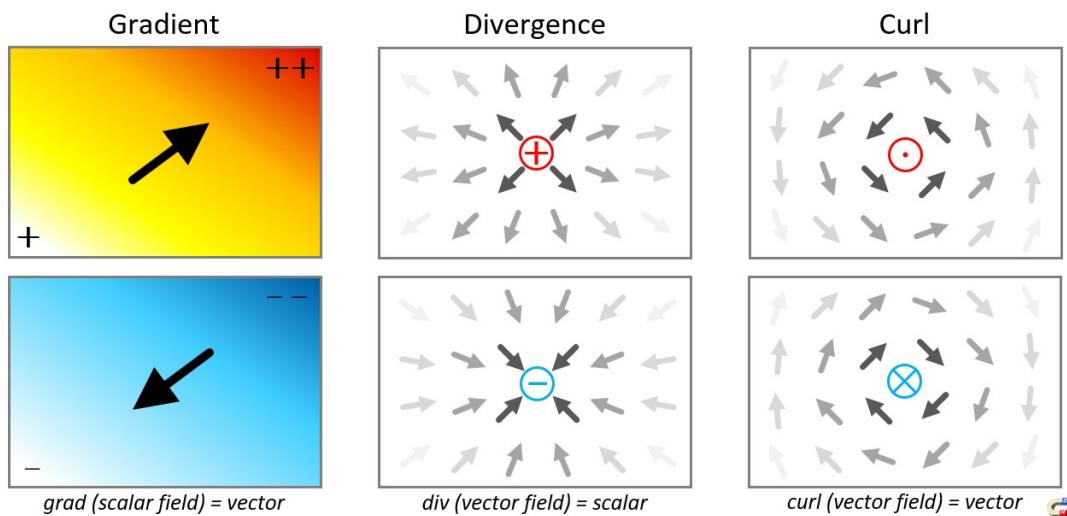
точки вектора \mathbf{s} . Позначимо через ds довжину вектора від однієї точки s до іншої, якщо точка знаходиться в центрі поля, то ds - збільшення радіус-вектора.

Позначимо через $d\phi/ds$ похідну від скаляра $d\phi/ds = \lim(ds \rightarrow 0) (\phi(x + dx, y + dy, z + dz) - \phi(x, y, z)) / ds$;

Розглянемо точки поля, де $\phi(x, y, z)$ приймає значення, наприклад, q .

Нехай \mathbf{n} позначає нормальну до поверхні, утворену точками поля, для яких справедливо рівняння $\phi(x, y, z) = q$, у напрямку збільшення ϕ .

Вектор, чисельно рівний $d\phi / dn$ і спрямований по нормальні $\phi = q$ в бік зростання значень функції ϕ , називається скалярним градієнтом: $\text{grad}(\phi) = (d\phi / dn) * \mathbf{n}$;



Author: S. Zurek, E-Magnetica.pl, CC-BY-4.0.

Дивергенція

$\text{div}(\mathbf{a}) = \nabla \cdot \mathbf{a}$. (Повертає скаляр)

Ротор

$\text{rot}(\mathbf{a}) = \nabla \times \mathbf{a}$. (Повертає вектор)

Градієнт

$\text{grad}(\mathbf{a}) = \nabla \mathbf{a}$. (Повертає вектор)

$\nabla \cdot \nabla = \nabla^2$, де ∇^2 – оператор Лапласа.

Оператор Лапласа

$\nabla^2 f = (d^2(f) / d(x)^2) + (d^2(f) / d(y)^2) + (d^2(f) / d(z)^2)$.

Оператор Лапласа (∇^2) часто використовують в записі рівняння Шредінгера.

Копенгагенська інтерпретація — ймовірнісне трактування рівнянь квантової механіки, в якому вектор стану квантової системи визначає амплітуду ймовірності.

Копенгагенська інтерпретація склалася в 1927 році під час співпраці Вернера Гайзенберга і Нільса Бора в Копенгагені, Данія.

Дві квантові системи з абсолютно однаковими хвильовими функціями матимуть однакові ймовірності результатів, але конкретні результати вимірювань можуть відрізнятися в серії експериментів через випадкові флуктуації, властиві квантовому світу. Просто кажучи, рівняння Шредінгера не передбачає стан системи, а передбачає ймовірність спостереження квантової системи в певному стані.

Рівняння Шредінгера у квантовій механіці описує еволюцію квантової системи з часом та визначає хвильову функцію системи, яка описує ймовірність знаходження системи у різних станах. Ервін Шредінгер опублікував рівняння Шредінгера в 1926 році. Загально відомо, що Ервін Шредінгер та Альберт Айнштайн до кінця життя не приймали імовірнісне трактування рівнянь квантової механіки як остаточне. Вони прагнули знайти рівняння, яке б однозначно передбачало стан квантової системи.

Рівняння Шредінгера:

$$\hat{H}\psi(t) = i\hbar * \partial\psi(t)/\partial t;$$

$$\text{Оператор Лапласа } \nabla^2 f = (d^2(f) / d(x)^2) + (d^2(f) / d(y)^2) + (d^2(f) / d(z)^2);$$

\hat{H} представляє оператор Гамільтона, який описує повну енергію системи.

Якщо частинка немає потенційної енергії, то гамільтоніан найпростіший. Для одного виміру:

$$\hat{H} = -(\hbar^2/2m) * \nabla^2;$$

$\psi(t)$ — це хвильова функція, яка є математичною функцією, яка описує квантовий стан частинки як функцію як положення, так і часу t .

i — уявна одиниця ($\sqrt{-1}$).

\hbar — приведена стала Планка, яка приблизно дорівнює $1,0545718 * 10^{-34}$ Джоуль-секунд.

Формально нормування хвильової функції означає наступне:

$$\int |\Psi(x)|^2 dx = 1,$$

де $\Psi(x)$ - хвильова функція, $|\Psi(x)|^2$ - квадрат амплітуди хвильової функції (імовірність знаходження частинки в інтервалі dx), а інтеграл береться за всіма значеннями x в області, що розглядається.

Це рівняння говорить нам, як хвильова функція $\psi(t)$ змінюється з часом. Простіше кажучи, він пов'язує енергію квантової системи зі швидкістю зміни хвильової функції в часі. Розв'язування рівняння Шредінгера дозволяє нам визначити дозволені рівні енергії та відповідні хвильові функції для квантової системи, що, у свою чергу, надає інформацію про поведінку частинок на квантовому масштабі.

Рівняння Шредінгера використовується для опису квантових властивостей атома водню. Атом водню є одним з найпростіших квантових систем.

Хвильова функція, яка задовольняє рівняння Шредінгера, містить інформацію про стан електрона в атомі водню, включаючи його енергію та ймовірність знаходження в певному місці.

Література:

- I.M.Кучерук, "Загальний курс фізики Том 2: Електрика і магнетизм",
- Г.Ф.Бушок, "Курс фізики Книга 2: Електрика і магнетизм",
- Іван Вакарчук, "Квантова механіка" (Львів, 2012),
- R. Penrose, "The Road to Reality", (Jonathan Cape, 2004),

- Mary Boas, "Mathematical Methods in the Physical Sciences" (1966),
- Max Born, "Atomic Physics" (8th Edition 1989),
- Linus Pauling, "General Chemistry" (1947),
- L.Cooper, “An introduction to the meaning and structure of physics”, (New York : Harper & Row, 1968),
- Andre Koch Torres Assis, “The experimental and historical foundations of electricity”, (Apeiron, 2018, 308 pages),
- Richard Feynman, Robert B. Leighton, “The Feynman Lectures on Physics, volume 2” (second edition 2005),
- Paul Scherz, Simon Monk, “Practical Electronics for Inventors” (Fourth Edition, 2016),
- Steven Weinberg, "Foundations of Modern Physics" (2021),
- David J. Griffiths, “Introduction to Electrodynamics” (fifth edition 2024),
- Leonard Susskind, George Hrabovsky, "The Theoretical Minimum" (2013).