АЛГОРИТМІЧНІ СТРАТЕГІЇ
РЕКУРСІЯ

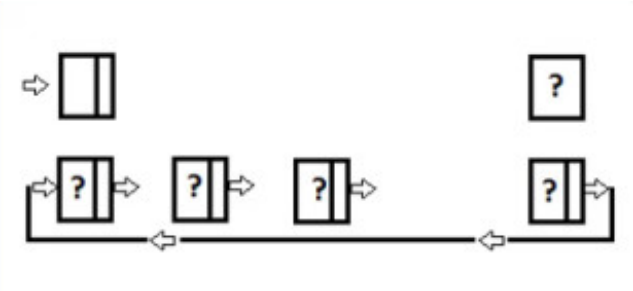
Лекція 2

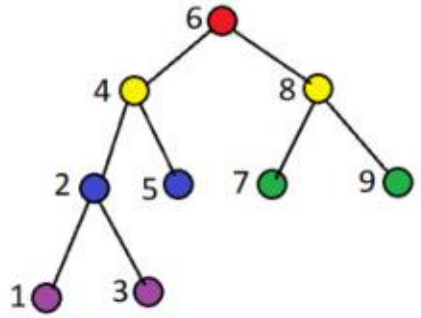
Дисципліна «Алгоритми та структури даних»

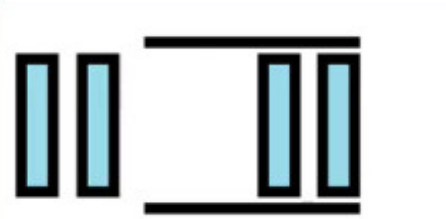
Базові алгоритми програмування

Алгоритми роботи із структурами даних - визначають базові принципи і методологію, які використовуються для реалізації, аналізу і порівняння алгоритмів

До таких структур відносяться зв'язні списки і рядки, дерева, абстрактні типи даних, такі як стеки і черги



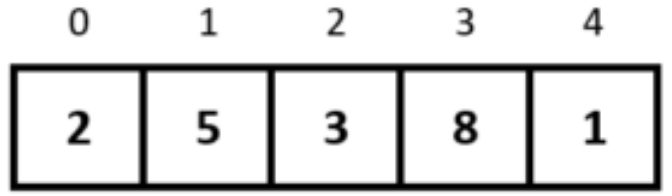


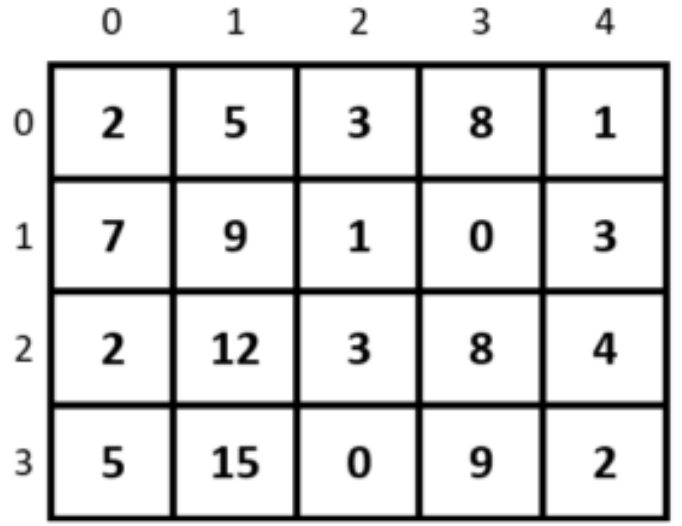


Базові алгоритми програмування

Алгоритми сортування, призначені для впорядкування масивів і файлів

З алгоритмами сортування пов'язані, зокрема, черги за пріоритетом, завдання вибору і злиття

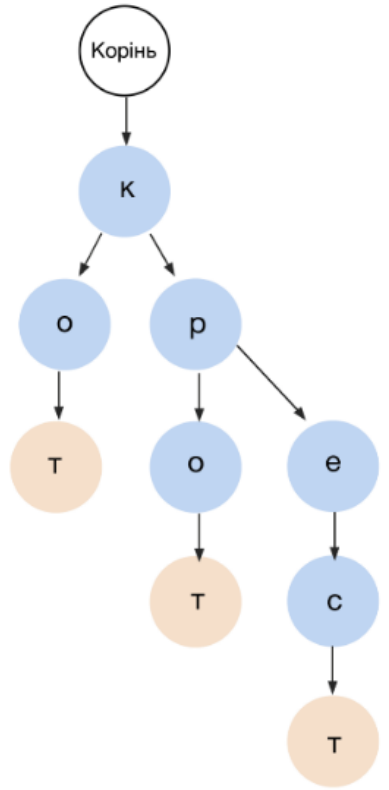


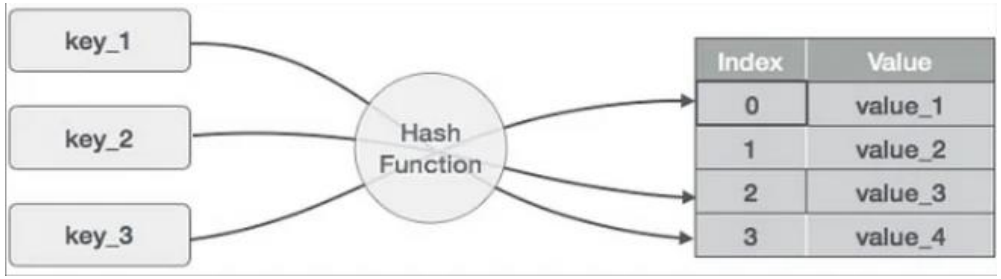


Базові алгоритми програмування

Алгоритми пошуку, призначені для пошуку конкретних елементів у великих колекціях елементів

До них відносяться основні і розширені методи пошуку з використанням дерев і перетворень цифрових ключів, у тому числі дерева цифрового пошуку (префіксні дерева), збалансовані дерева, хешування, а також методи, які підходять для роботи з дуже великими файлами

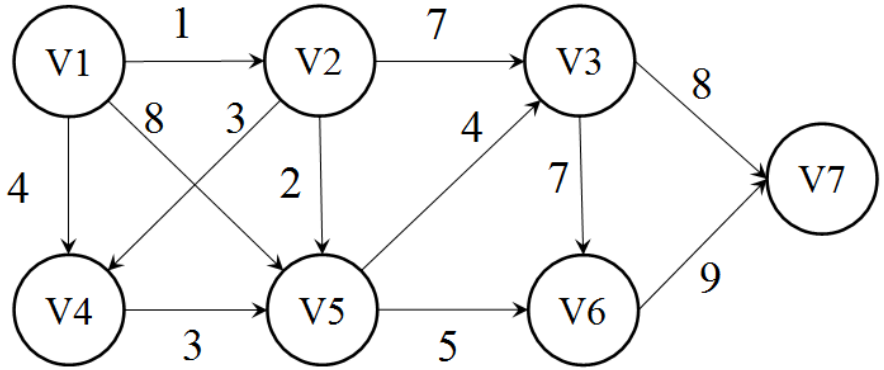


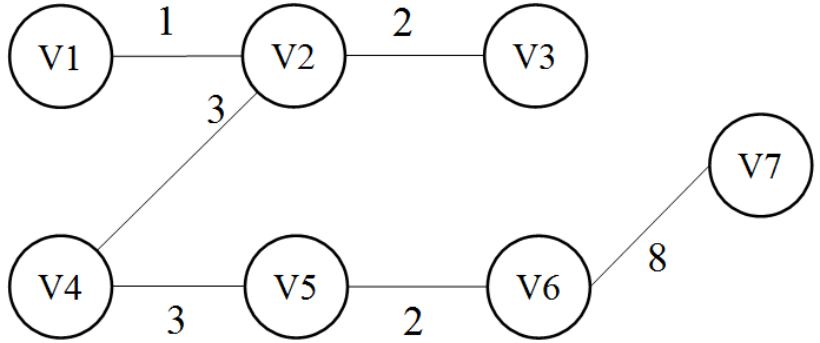


Базові алгоритми програмування

Алгоритми на графах корисні при рішенні ряду складних і важливих завдань

Загальна стратегія пошуку на графах розробляється і застосовується до фундаментальних завдань зв'язності, у тому числі до завдання відшукування найкоротшого шляху, побудови мінімального остовного дерева, до завдання про потоки в мережах тощо





Базові алгоритми програмування

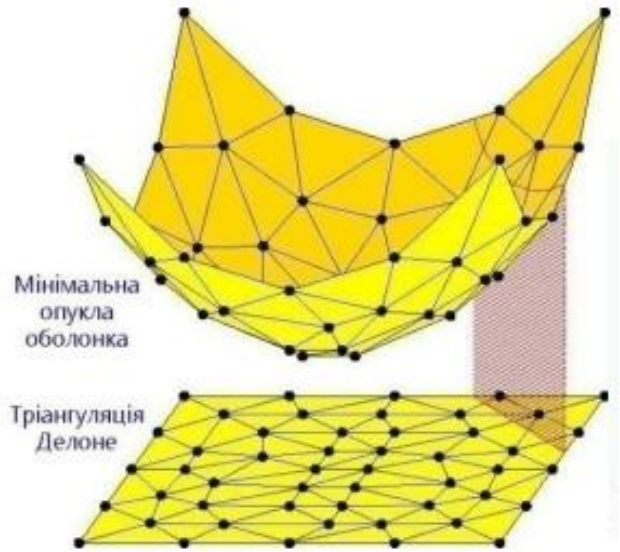
Алгоритми обробки рядків включають методи обробки послідовностей (у тому числі довгих послідовностей) символів. Пошук в рядку призводить до зіставлення з еталоном, що у свою чергу веде до синтаксичного аналізу

Геометричні алгоритми - це методи рішення завдань з використанням точок і ліній (і інших простих геометричних об'єктів)

До них відносяться алгоритми побудови опуклих оболонок, заданих набором точок, визначення перетинів геометричних об'єктів, рішення завдань відшукування найближчих точок і алгоритму багатовимірного пошуку

“Python” “16.09.2025” ‘Запоріжжя’

Рядок - складний тип даних**,** упорядкована послідовність символів, що використовується для зберігання та представлення текстової інформації



Найпоширеніші стратегії алгоритмізації

Стратегії алгоритмізації - це високорівневі, загальні підходи або парадигми для розв'язання цілого класу проблем. Вони є концептуальною основою, філософією, яка визначає, як ми будемо шукати рішення

Стратегія не містить конкретних кроків, а має лише загальну ідею

- алгоритми грубої сили;
- жадібні алгоритми;
- «розділяй і володарюй»;
- алгоритми з поверненням;
- евристичні алгоритми;
- зіставлення із зразком і алгоритми обробки рядків/текстів;
- алгоритми чисельної апроксимації;
- динамічне програмування;
- …..

Алгоритми грубої сили (brute-force)

Метод “грубої сили” полягає в переборі всіх можливих варіантів розв’язання задачі та виборі найкращого. Цей метод дозволяє знайти коректний результат, але він не є ефективним для великих обсягів даних

Недоліки:

- **Низька ефективність:** для більшості складних задач ці алгоритми є дуже повільними, оскільки кількість варіантів, які потрібно перевірити, зростає експоненційно
- **Висока обчислювальна складність:** час виконання часто вимірюється в експоненційній або факторіальній залежності від розміру вхідних даних: **O(n!)**, **$O(2^n)$**

Приклади

- **Пошук елемента в масиві:** найпростіший приклад - послідовний пошук; алгоритм просто перебирає кожен елемент масиву, порівнюючи його з шуканим значенням, доки не знайде збіг
- **Задача комівояжера:** Один з класичних прикладів, де алгоритм грубої сили є неефективним. Задача полягає в пошуку найкоротшого маршруту, що проходить через усі міста. Алгоритм грубої сили перевіряє всі можливі маршрути, що вимагає розрахунків факторіалу (n-1)!, де n - кількість міст.

Жадібна стратегія (Greedy)

Розв’язує завдання розміру *n* покроково

На кожному кроці жадібний алгоритм знаходить найкращий локальний розв’язок, який можна одержати, враховуючи наявну інформацію

Як правило, розмір завдання при цьому зменшується на 1. Після того, як буде виконано всі *n* кроків, алгоритм повертає загальний розв’язок

Складність жадібного алгоритму залежить від трьох основних етапів, які він виконує: сортування, перебір і наповнення (найчастіше вона визначається етапом сортування)

загальна складність алгоритму дорівнює

O(n log n)

Завдання: У вас є рюкзак, який може витримати максимальну вагу W = 15 кг. У вашому розпорядженні є набір предметів, кожен з яких має свою вагу та цінність. Заповнити рюкзак таким чином, щоб отримати максимальну загальну цінність, не перевищивши максимальну вагу

Набір предметів:

Предмет A: вага = 7 кг, цінність = 40
Предмет B: вага = 2 кг, цінність = 10
Предмет C: вага = 4 кг, цінність = 20
Предмет D: вага = 5 кг, цінність = 25
Предмет E: вага = 3 кг, цінність = 15

Результат

Наповнення рюкзака: Предмет A (повністю), Предмет B (повністю), Предмет C (повністю), Предмет D (частина)
Максимальна цінність: 80

Стратегія “розділяй і володарюй” (“divide and conquer”)

Стратегія полягає в розбитті великої задачі на менші підзадачі того ж типу. Кожна підзадача вирішується окремо, а результати об’єднуються, щоб отримати рішення великої задачі. Цей метод дозволяє розв’язувати складні задачі та зменшувати кількість коду. Складність коливається від **O(log n)** до **O(nlogn)**, що робить цю стратегію дуже ефективною

Приклади

1. Сортування злиттям (Merge Sort)

• **Розділяй:** масив ділиться навпіл доти, доки кожен підмасив не міститиме лише один елемент (один елемент вважається відсортованим).

• **Володарюй:** кожен окремий елемент розглядається як відсортований підмасив.

• **Об'єднуй:** відсортовані підмасиви послідовно об’єднуються, формуючи все більші та більші відсортовані масиви, поки не буде отримано один повний відсортований масив.

2. Бінарний пошук (Binary Search) - використовується для пошуку елемента у відсортованому масиві.

• **Розділяй:** вибирається середній елемент масиву. Якщо шуканий елемент менший, пошук продовжується в лівій частині; якщо більший - у правій.

• **Володарюй:** проблема зводиться до меншої, але ідентичної підпроблеми (пошук у половині масиву).

• **Об'єднуй:** об'єднання не потрібне, оскільки рішення (знайдено/не знайдено) повертається на одному з рекурсивних кроків

Динамічне програмування (Dynamic Programming)

Стратегія динамічного програмування полягає в розбитті великої задачі на менші підзадачі та вирішенні їх окремо з метою оптимізації процесу розв'язання задачі. Стратегія є дуже ефективною та часто використовується для розв'язання складних задач

Головна перевага динамічного програмування полягає в тому, що воно перетворює експоненційну складність рекурсивних рішень на **поліноміальну** (наприклад, **O(n)**, **O(n²)**, **O(n³)**)

Числа Фібоначчі: **0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...** ; складність **O(n)**
Задача про рюкзак (цілочисельна): складність **O(n×W)** (кількість підзадач залежить від кількості предметів (**n**) та максимальної ваги рюкзака (**W**)

Методи реалізації стратегій

Методи - це конкретні реалізації, покрокові інструкції, які застосовують певну стратегію до розв'язання конкретної задачі. Якщо стратегія – це "як ми будемо думати", то метод – це "що ми будемо робити".

Приклади методів:

- **Сортування злиттям (Merge Sort) -** метод, який реалізує стратегію **«розділяй і володарюй»;**
- **Алгоритм Дейкстри -** метод, який реалізує **жадібну стратегію** для пошуку найкоротшого шляху;
- **Алгоритм послідовного пошуку -** метод, що реалізує стратегію **грубої сили**;
- **Табуляція чисел Фібоначчі -** метод, який реалізує стратегію **динамічного програмування**.

Метод декомпозиції - основний принцип, на якому ґрунтуються багато стратегій

Метод декомпозиції - це підхід до розв'язання складної проблеми шляхом її **розбиття на менші, простіші та більш керовані частини**. Кожну з цих частин (підзадач) можна розв'язати незалежно, а потім об’єднати для отримання остаточного рішення.

Розглянемо метод декомпозиції на прикладі сортування масиву. Одна з найпопулярніших технік сортування **- сортування злиттям (Merge Sort) -** є класичним прикладом декомпозиції

Завдання: відсортувати масив чисел від меншого до більшого.

Початковий масив: [8, 3, 1, 7, 0, 10, 2]

Метод декомпозиції. Сортування злиттям

1. Декомпозиція (Розбиття)

Замість того, щоб сортувати весь масив одразу, розбиваємо його на дві менші, рівні частини. Потім кожну з цих частин розбиваємо знову, і так доти, доки не залишиться масив, що складається з одного елемента. Масив з одного елемента вже відсортований - це **базовий випадок**

[8, 3, 1, 7, 0, 10, 2]

↓ Розбиття

[8, 3, 1, 7] і [0, 10, 2]

↓ Розбиття

[8, 3] і [1, 7] та [0, 10] і [2]

↓ Розбиття

[8], [3], [1], [7], [0], [10], [2] (кожен масив з одного елемента є відсортованим)

2. Вирішення підзадач (Злиття та сортування). Починається процес **злиття (merge)**: беремо два відсортовані масиви і з'єднуємо їх, щоб отримати один більший, теж відсортований масив:

здійснюємо злиття [8] і [3], щоб отримати [3, 8]
здійснюємо злиття[1] і [7], щоб отримати [1, 7]
здійснюємо злиття[0] і [10], щоб отримати [0, 10]
Масив [2] залишається
[3, 8], [1, 7], [0, 10], [2]

3. Об'єднання рішень. Продовжуємо злиття, поки не об'єднаємо всі масиви в один:

здійснюємо злиття[3, 8] і [1, 7], щоб отримати [1, 3, 7, 8]
здійснюємо злиття[0, 10] і [2], щоб отримати [0, 2, 10]
здійснюємо злиття двох масивів, щоб отримати остаточне рішення
[1, 3, 7, 8] і [0, 2, 10] → **[0, 1, 2, 3, 7, 8, 10]**

У результаті отримуємо повністю відсортований масив. Весь процес сортування був виконаний за допомогою простих операцій об'єднання, що значно спростило задачу.

Поняття рекурсії: визначення та основні принципи

Рекурсія – це процес, коли функція викликає саму себе з певними аргументами. Це означає, що виконання функції залежить від виконання цієї ж функції з іншими аргументами. Рекурсія може бути простою, коли функція викликає саму себе один раз, або складною, коли функція викликає саму себе кілька разів з різними аргументами

Принципи рекурсії:

Базовий випадок

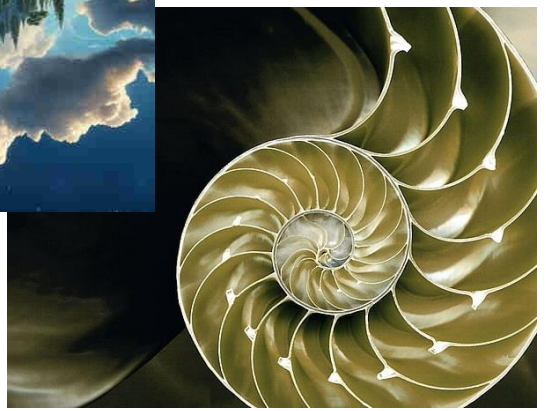
Рекурсія повинна мати базовий випадок, коли функція не викликає саму себе. Цей випадок служить базою для рекурсії і дозволяє зупинити процес виконання функції

Рекурсивний випадок

Рекурсія повинна також мати рекурсивний випадок, коли функція викликає саму себе з іншими аргументами. Цей випадок служить вхідними даними для рекурсії

Приклади рекурсії

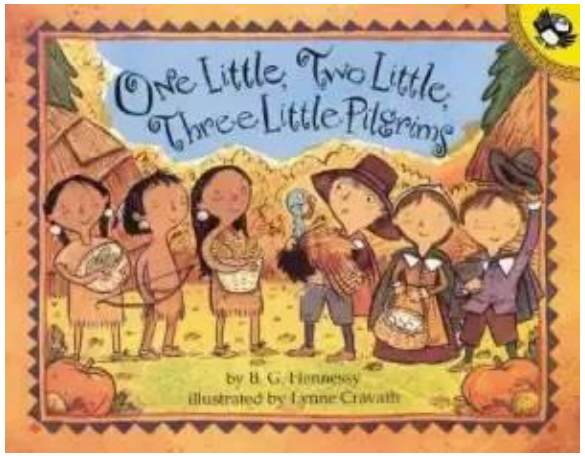
- **Фрактали**. Фрактальні візерунки, такі як сніжинки або геометричні фігури, будуються повторенням того ж шаблону на різних масштабах
- **Процеси зростання у природі**. Наприклад, розгалуження дерев або зростання кристалів
- **Математичні послідовності**. Наприклад, послідовність Фібоначчі визначається як сума двох попередніх чисел у послідовності, створюючи рекурсивний шаблон
- **Дзеркальні відбиття**. При розташуванні двох дзеркал паралельно одне одному відбувається нескінченне відображення зображення одного дзеркала в іншому
- **Спіральні візерунки у природі**. Наприклад, спіральні візерунки морських раковин або соняшникове насіння розташовуються в спіральній формі, утворюючи рекурсивний шаблон



Приклад рекурсії - Віршик про індіанців

One little, two little, three little Indians
Four little, five little, six little Indians
Seven little, eight little, nine little Indians
Ten little Indian boys.

Ten little, nine little, eight little Indians
Seven little, six little, five little Indians
Four little, three little, two little Indians
One little Indian boy



Базовий випадок (Base Case): Існує умова зупинки, яка не вимагає подальшого повторення. У віршику це «один маленький індіанець», після якого рахунок не зменшується

Рекурсивний крок (Recursive Step): Виконується дія, яка повторює саму себе, але з меншим значенням. Це послідовний рахунок від десяти до одного

Проста рекурсія

```
def countdown(n):
    if n <= 0:
        print("Stop!")
    else:
        print(n)
        countdown(n - 1)

countdown(5)
```

Функція викликає себе з аргументом, зменшеним на одиницю, доки не досягне базового випадку (n <= 0), який завершує рекурсію.

Результат:

5
4
3
2
1
Stop!

Повернення суми елементів списку

```python
def list_sum(nums):
    if len(nums) == 0:
        return 0
    else:
        return nums[0] + list_sum(nums[1:])

numbers = [1, 2, 3, 4, 5]
print("Summa:", list_sum(numbers))
```

Замість використання циклу, функція викликає сама себе, щоб розбити задачу на простіші частини

Базовий випадок (умова виходу):
Рядок if len(nums) == 0: return 0 є **умовою зупинки**. Коли функція отримує порожній список ([]), вона повертає 0. Це запобігає нескінченній рекурсії

Рекурсивний випадок:
Рядок else: return nums[0] + list_sum(nums[1:]) є **рекурсивним викликом**

Функція додає **перший елемент списку** (nums[0]) до результату, який повернеться після виклику list_sum для **решти списку** (nums[1:])
nums[1:] створює новий список, який містить усі елементи, крім першого. Наприклад, якщо nums був [1, 2, 3], то nums[1:] буде [2, 3]

Факторіал числа

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

Базовий випадок (Умова зупинки):

Рядок if n == 0: є критично важливим. Він перевіряє, чи досягли ми найпростішої підзадачі. Факторіал нуля дорівнює 1. Коли n стає 0, функція повертає 1, і рекурсивні виклики починають «розкручуватися» назад, обчислюючи кінцевий результат.

Рекурсивний випадок (Виклик самої себе):

Рядок return n * factorial(n - 1) показує суть рекурсії. Функція повертає n, помножене на результат виклику тієї ж функції, але з аргументом n-1

Це створює ланцюжок викликів: factorial(5) → factorial(4) → factorial(3) → factorial(2) → factorial(1) → factorial(0).

Пошук максимального елементу у списку

```python
def find_max(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        max_rest = find_max(lst[1:])
        return lst[0] if lst[0] > max_rest else max_rest

numbers = [3, 7, 2, 8, 5]
print("Max:", find_max(numbers))
```

Функція розбиває задачу на простіші частини, порівнюючи перший елемент списку з максимальним значенням з решти елементів

Базовий випадок (умова зупинки):

Рядок if len(lst) == 1: return lst[0] - це **умова зупинки**. Коли функція отримує список, що містить лише один елемент, вона повертає цей елемент, оскільки він, очевидно, є максимальним у цьому списку. Це запобігає нескінченній рекурсії.

Рекурсивний випадок:

Рядок else: ... - це **рекурсивний випадок**.

max_rest = find_max(lst[1:]) - тут функція рекурсивно викликає саму себе, щоб знайти максимальне значення в **решті списку** (lst[1:])

return lst[0] if lst[0] > max_rest else max_rest - цей рядок порівнює **перший елемент списку** (lst[0]) з максимальним значенням, знайденим у решті списку (max_rest). Функція повертає з цих двох значень те, що є більшим

Аспекти рекурсії

Проста рекурсія. Функція викликає себе безпосередньо без додаткової обробки результату

Рекурсія із поверненням значення. Функція викликає саму себе і використовує значення, що повертається для подальших обчислень

Рекурсія із базовим випадком. Усі приклади мають базовий випадок, який завершує рекурсію та запобігає безкінечному виконанню

Рекурсія з розподілом на підзадачі. Кожен приклад розбиває задачу на більш дрібні підзадачі, які потім вирішуються рекурсивно