1. АЛГОРИТМИ ТА ОБЧИСЛЮВАЛЬНА СКЛАДНІСТЬ

1.1. Основи структури даних і алгоритми

1.1.1. Поняття алгоритму. Визначення його часової та просторової (за обсягом пам'яті) складності.

Сложность алгоритмов: [YouTube](https://www.youtube.com/watch?v=cXCuXNwzdfY)

Мега корисна Книга (приклади на Python): [АЛГОРИТМИ І СТРУКТУРИ ДАНИХ](https://www.mechmat.univ.kiev.ua/wp-content/uploads/2021/09/pidruchnyk-alhorytmy-i-struktury-danykh.pdf)

1.1.2. Поняття абстрактного типу даних. Абстрактні типи даних: стеки, списки, вектори, словники, множини, мультимножини, черги, черги з пріоритетами.

Плейліст де коротко розповідають основні структури даних: [YouTube](https://youtube.com/playlist?list=PLQOaTSbfxUtAIipl4136nwb4ISyFk8oI4&si=BXlMlEnfCMIUGSgB)

Мега корисна Книга (приклади на Python): [АЛГОРИТМИ І СТРУКТУРИ ДАНИХ](https://www.mechmat.univ.kiev.ua/wp-content/uploads/2021/09/pidruchnyk-alhorytmy-i-struktury-danykh.pdf)

1.1.3. Кортежі, множини, словники, одно- та двобічнозв'язні списки. Реалізація абстрактних типів даних з оцінюванням складності операцій.

Мега корисна Книга (приклади на Python): [АЛГОРИТМИ І СТРУКТУРИ ДАНИХ](https://www.mechmat.univ.kiev.ua/wp-content/uploads/2021/09/pidruchnyk-alhorytmy-i-struktury-danykh.pdf)

1.1.4. Базові алгоритми та їх складність: пошук, сортування (прості сортування вибором, вставками, обмінами та удосконалені сортування деревом, сортування Шелла, швидке сортування).

Переглянути ролик для ознайомлення або повтору теми (Сортування): [YouTube](https://www.youtube.com/watch?v=PF7AqefS4MU)

1.1.5. Алгоритми на графах та їх складність:

* пошук в ширину і глибину;
* пошук зв'язних компонентів;
* побудова кістякового дерева;
* побудова найкоротших шляхів з виділеної вершини;
* побудова найкоротших шляхів між двома вершинами;

Переглянути плейліст для ознайомлення або повтору теми : [YouTube](https://www.youtube.com/watch?v=VehB3eglQMQ)

Мега корисна Книга (приклади на Python): [АЛГОРИТМИ І СТРУКТУРИ ДАНИХ](https://www.mechmat.univ.kiev.ua/wp-content/uploads/2021/09/pidruchnyk-alhorytmy-i-struktury-danykh.pdf)

1.2. Стратегії розроблення алгоритмів.

1.2.1. Стратегія «розділяй та володарюй» та приклади застосування.

«Розділя́й та володарюй» 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.

<https://www.youtube.com/watch?v=SMzo4zPZiko>

Типовий приклад — алгоритм сортування злиттям. Щоб відсортувати масив чисел за зростанням, його розбивають на дві рівні частини; кожну сортують, потім відсортовані частини зливають в одну. Ця процедура застосовується до кожної з частин доти, поки сортовані частини масиву містять хоча б два елементи (щоб можна було її розбити на дві частини).

Приклади стратегії розділяй та володарюй:

Двійковий пошук:

* Розділяйте впорядкований масив на дві частини і порівнюйте ключ, який потрібно знайти, з серединнім елементом.
* Виберіть половину масиву, в якій можливо міститься ключ, і повторюйте процес пошуку у цій підмасиві.
* Цей процес триває до тих пір, поки ключ не буде знайдений або діапазон пошуку стане порожнім.

Метод бісекції:

* Розділіть інтервал, на якому ви шукаєте корінь або розв'язок, на дві частини.
* Обчисліть значення функції у середині цього інтервалу.
* Залежно від знаку значення функції визначте, у якій половині інтервалу знаходиться корінь або розв'язок.
* Повторюйте цей процес для нового, меншого інтервалу, доки не буде досягнута потрібна точність.

Швидке сортування:

* Розділіть масив на дві частини, використовуючи опорний елемент.
* Упорядкуйте кожну з половинок масиву рекурсивно.
* Об'єднайте відсортовані підмасиви разом.

Швидке перетворення Фур'є:

* Розділіть послідовність сигналу на підпослідовності.
* Використовуйте рекурсивне розкладання кожної підпослідовності на складові, що складаються.
* Об'єднайте результати, використовуючи алгоритм злиття Фур'є.

Множення Карацуби: <https://www.youtube.com/watch?v=Z7EPXlGVYgA>

* Розділіть числа, які ви хочете перемножити, на половини.
* Використовуйте рекурсивне множення для кожної пари половин.
* Об'єднайте результати, використовуючи алгоритм злиття.

1.2.2. Стратегія «Балансування» та приклади застосування.

Балансування навантаження (англ. load balancing) — це процес розподілення множини задач на множині ресурсів (обчислювальних одиниць), з метою зробити загальний час їхнього опрацювання більш дієвим. Балансування навантаження може оптимізувати час відповідь і запобігти нерівномірному навантаженню на окремі обчислювальні вузли тоді як інші обчислювальні вузли бездіють.

Балансування навантаження це предмет досліджень у галузі паралельних обчислень. Головні два підходи — статичні алгоритми, які не беруть до уваги стан різних машин, динамічні алгоритми, які зазвичай загальніші і дієвіші, але вимагають обміну інформацією між різними обчислювальними одиницями, що створює ризик падіння дієвості.

Ось декілька прикладів застосування стратегії балансування:

AVL-дерева:

* AVL-дерева є видом самобалансуючих бінарних дерев пошуку.
* Під час вставки або видалення вузлів AVL-дерево автоматично перебалансовується для забезпечення оптимальної висоти дерева.
* Це дозволяє забезпечити швидкий доступ до даних усіх операцій (вставка, видалення, пошук).

Розподілені системи:

* У розподілених системах рівновага може бути досягнута шляхом розподілу навантаження між різними вузлами або серверами.
* Механізми такі, як розподілене навантаження, можуть використовуватися для автоматичного розподілу запитів між серверами в залежності від їх поточного стану.

Оптимізація алгоритмів:

* В більш загальному сенсі, балансування може бути досягнуто шляхом оптимізації алгоритмів та структур даних для забезпечення рівномірного використання ресурсів.
* Наприклад, у випадку алгоритмів розподілених обчислень, можна розробити стратегії, які автоматично розподіляють завдання між різними вузлами з урахуванням їх потужності та завдань.

Методи балансування навантаження:

* У веб-програмуванні, методи балансування навантаження використовуються для розподілу запитів між різними серверами у залежності від їх поточного стану.
* Це допомагає уникнути перевантаження окремих серверів та забезпечити кращу продуктивність та доступність системи.

1.2.3. Динамічне програмування та приклади застосування.

Динамічне програмування - це метод розв'язання складних задач, який базується на рекурсії та зберіганні проміжних результатів для запобігання повторних обчислень. Він може бути успішно застосований до різних алгоритмів пошуку, сортування та методів структур даних для оптимізації часу виконання.

Ось декілька прикладів застосування динамічного програмування:

Найкоротший шлях в графі (Dynamic Programming for Shortest Path):

* У задачі пошуку найкоротшого шляху в графі, можна застосовувати динамічне програмування для обчислення найкращого шляху між двома вершинами.
* Проміжні результати можуть бути збережені, щоб уникнути повторного обчислення шляху між тими ж вершинами.

Рюкзакові задача (Dynamic Programming for Knapsack Problem):

* У задачі вибору найцінніших предметів для наповнення рюкзака обмеженого об'єму, можна використовувати динамічне програмування для знаходження оптимального розв'язку.
* Тут проміжні результати можуть включати найкращу вартість, яку можна досягнути для кожного обсягу рюкзака.

Найбільша зростаюча підпослідовність (Dynamic Programming for Longest Increasing Subsequence):

* У задачі пошуку найбільшої зростаючої підпослідовності можна використовувати динамічне програмування для знаходження найкращого розв'язку.
* Проміжні результати можуть зберігатися для кожного під масиву, щоб уникнути повторних обчислень.

Оптимальне сортування (Dynamic Programming for Optimal Sorting):

* Динамічне програмування може бути використане для знаходження оптимальної послідовності операцій сортування (наприклад, обміну, вставки тощо), щоб досягти мінімальної кількості порівнянь чи обмінів.

<https://www.youtube.com/watch?v=5NmE1gcL_kY>

1.3. Моделі обчислень.

Моделі обчислень - це абстрактні концепції або формальні системи, які дозволяють описувати, аналізувати та розуміти процеси обчислень. Ці моделі можуть бути математичними, теоретичними або апаратними, і вони допомагають вивчати та вдосконалювати алгоритми, програми та системи обчислень.

Наприклад, модель Тьюрінга - одна з найвідоміших та базових моделей обчислень. Вона складається з набору правил, які визначають, як мають виконуватися обчислення на абстрактному пристрої (так званій машині Тьюрінга). Ця модель допомагає вивчати можливості та обмеження обчислювальних процесів.

Існують і інші моделі, такі як модель хмарних обчислень, модель вищих рівнів абстракції програмування, модель взаємодії процесів у мережах, або навіть фізичні моделі квантових обчислень. Вони всі відображають певні аспекти та підходи до обчислень.

1.3.1. Імперативний та декларативний підходи до праграмування.

Імперативний та декларативний підходи - це два різних способи вирішення завдань у програмуванні.

**Імперативний підхід:** У цьому підході програміст вказує комп'ютеру точні кроки, які потрібно виконати для досягнення певного результату. Основними конструкціями є команди присвоєння, цикли, умовні вирази та процедури. Прикладами мов програмування, що використовують імперативний підхід, є C, C++, Java та Python у деяких аспектах.

**Декларативний підхід:** У цьому підході програміст зосереджується на тому, що потрібно зробити, а не на тому, як це робити. Замість того, щоб описувати послідовність дій, програміст надає опис того, який результат потрібно отримати. Основними конструкціями є правила, обмеження та запити до баз даних. Мови програмування, що підтримують декларативний підхід, включають SQL (Structured Query Language), Prolog та HTML/CSS (у випадку веб-розробки).

Імперативний підхід часто використовується для оптимізації продуктивності та точного контролю над виконанням програм. Декларативний підхід, натомість, часто дозволяє вищому рівню абстракції та полегшує розуміння програмного коду, що може призвести до більш ефективного розвитку та підтримки програмного забезпечення.

<https://foxminded.ua/imperatyvne-ta-deklaratyvne-prohramuvannia/>

<https://dou.ua/forums/topic/37548/#:~:text=Імперативний%20і%20декларативний%20підходи%20у,виконання%20команд%20(імперативний%20підхід)>.

1.3.2. Розв'язні, напіврозв'язні та нерозв'язні проблеми. Проблема зупинки.

Терміни "розв'язні", "напіврозв'язні" та "нерозв'язні" використовуються для характеристики можливості вирішення або невирішення певних проблем у контексті обчислювальної теорії.

**Розв'язні проблеми:** Це проблеми, для яких існує алгоритм або метод, який дозволяє знайти точний розв'язок за скінчений час. Найпростіший приклад - це додавання двох чисел.

**Напіврозв'язні проблеми:** Це проблеми, для яких існує алгоритм або метод, який може виявити наявність розв'язку, але не завжди може дати відповідь "так" або "ні" за скінчений час. Такі проблеми можуть мати розв'язок, але алгоритм не завжди зможе його знайти. Прикладом напіврозв'язних проблем є проблема зупинки.

**Нерозв'язні проблеми:** Це проблеми, для яких не існує жодного алгоритму або методу, який може знайти розв'язок за скінчений час. Одним із класичних прикладів є проблема зупинки.

**Проблема зупинки:** Проблема зупинки - це класична нерозв'язна проблема в обчислювальній теорії. Вона полягає в тому, щоб написати програму, яка могла б визначити, чи зупиниться інша програма при введенні в неї деякого коду. Математик Алан Тьюрінг довів, що ця проблема нерозв'язна: не існує жодного алгоритму, який би гарантовано вирішив її для всіх можливих вхідних даних. Це означає, що існує певні програми, для яких неможливо визначити, чи вони будуть працювати завжди, чи вони "зацикляться" у безкінечному циклі.

Дуже складний матеріал додав щоб шось було (не читайте це): <https://studfile.net/preview/7328750/>

Відео с того же розряду: <https://www.youtube.com/watch?v=F3eaPfzCO_A>