**1. Основи структури даних і алгоритми**

**1.1 Поняття алгоритму. Визначення його часової та просторової (за обсягом пам’яті) складності.**

*Алгоритм* — це скінчена послідовність указівок на виконання дій, спрямованих на розв'язування задачі. Алгоритм складається з окремих кроків, які потрібно виконати в певному порядку.

*Складність алгоритму* – це кількісна характеристика, що відображує споживані алгоритмом ресурси під час свого виконання. Складність алгоритмів зазвичай оцінюють за часом виконання або по використовуваній пам'яті.

***Часова складність алгоритму* -**  Це кількість операцій або час, який потрібно для виконання алгоритму в залежності від розміру вхідних даних. Вимірюється в часових одиницях, таких як кількість операцій (наприклад, порівняння або обмін) або в часових одиницях, таких як мілісекунди, секунди тощо.

***Просторова складність алгоритму* -** Це кількість пам'яті, необхідної для виконання алгоритму, в залежності від розміру вхідних даних. Вимірюється в байтах або кількості пам'яті, використовуваної для зберігання даних та додаткових структур даних.

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

*Абстрактний тип даних* - це математична модель, яка визначає набір даних разом з набором операцій, які можуть бути виконані над цими даними. Він визначає лише операції, але не конкретну реалізацію цих операцій. Це дозволяє розділити інтерфейс користувача від реалізації даних, що спрощує розробку та підтримку програм.

*Стек* - це структура даних, що працює за принципом "Last In, First Out" (LIFO).

Операції:

* **push()**: Додає елемент на верхівку стека.
* **pop()**: Видаляє і повертає верхній елемент зі стека.
* **peek()** або **top()**: Повертає верхній елемент, не видаляючи його.
* **isEmpty()**: Перевіряє, чи є стек порожнім.
* **size()**: Повертає кількість елементів у стеці.

*Список* - це послідовність елементів, де кожен елемент має свій індекс.

Операції:

* **append(item)**: Додає елемент в кінець списку.
* **insert(index, item)**: Додає елемент на конкретну позицію в списку.
* **remove(item)**: Видаляє перше входження елемента зі списку.
* **pop(index)**: Видаляє і повертає елемент за вказаним індексом, або останній елемент, якщо індекс не заданий.
* **clear()**: Видаляє всі елементи зі списку.
* **index(item)**: Повертає індекс першого входження елемента.
* **count(item)**: Повертає кількість входжень елемента у списку.
* **access by index**: Доступ до елементів за індексом, наприклад, list[0].
* **extend(other\_list)**: Додає всі елементи іншого списку.
* **copy()**: Створює копію списку.
* **sort()**: Сортує елементи списку на місці.
* **sorted(list)**: Повертає новий відсортований список.
* **reverse()**: Змінює порядок елементів на зворотний.
* **reversed(list)**: Повертає ітератор для зворотного перебору.

*Вектор* - це динамічний масив, який може збільшуватися або зменшуватися у розмірі. Складність операцій O(1), хоча може бути O(n).

Операції:

* **push\_back(item)**: Додає елемент в кінець вектора.
* **insert(index, item)**: Додає елемент на вказану позицію у векторі.
* **pop\_back()**: Видаляє останній елемент вектора.
* **erase(index)**: Видаляє елемент за вказаним індексом.
* **clear()**: Видаляє всі елементи з вектора.
* **at(index)**: Повертає елемент за вказаним індексом, з перевіркою меж.
* **[index]**: Доступ до елемента за вказаним індексом без перевірки меж.
* **Призначення** нового значення за вказаним індексом, наприклад, vec[index] = new\_value.
* **size()**: Повертає кількість елементів у векторі.
* **capacity()**: Повертає поточну місткість вектора (скільки елементів він може вмістити без перевиділення пам'яті).
* **reserve(n)**: Збільшує місткість вектора до щонайменше n елементів.
* **assign(range)**: Замінює поточні елементи вектора елементами з іншого діапазону.
* **copy()**: Створює копію вектора.
* **sort(begin, end)**: Сортує елементи у векторі.
* **reverse(begin, end)**: Змінює порядок елементів на зворотний.

*Словник* - (Map, HashMap) абстрактний типо даних, що представляє собою колекцію пар "ключ-значення". Кожен ключ у словнику унікальний, а значення може бути будь-якого типу. Основною особливістю словників є швидкий доступ до значення за заданим ключем.

*Множина* (Set) — це абстрактний тип даних, який представляє собою колекцію унікальних елементів без конкретного порядку.   
Операції:

* **Додавання елементів:** Множини дозволяють додавати нові елементи. Якщо елемент вже існує, він не додається вдруге.
* **Видалення елементів:** Множини дозволяють видаляти існуючі елементи. Це можна робити явно або видаляти випадковий елемент, якщо такий метод підтримується.  
  **Перевірка наявності** Можна швидко перевірити, чи існує певний елемент у множині. Це особливо корисно для операцій пошуку.
* **Об'єднання:** Об'єднання множин дозволяє створити нову множину, що містить усі унікальні елементи з обох множин.
* **Перетин:** Перетин множин створює нову множину, що містить тільки ті елементи, які є спільними для обох множин.
* **Різниця:** Різниця множин створює нову множину, яка містить елементи однієї множини, що не входять в іншу множину.
* **Симетрична різниця:** Симетрична різниця створює нову множину, що містить елементи, які входять у будь-яку з множин, але не в обидві одночасно.

*Мультимножина* - це колекція, яка може містити дубльовані елементи. **Унікальність проти Кратності**: У звичайній множині кожен елемент може бути представлений лише один раз, тоді як у мультимножині кожен елемент може зустрічатися кілька разів. **Операції з Повтореннями**: Операції над мультимножинами враховують кількість повторень елементів, що додає складності в порівнянні з операціями над звичайними множинами.

*Черга* - це структура даних, що працює за принципом "First In, First Out" (FIFO). Основні операції - це додавання елементу в кінець черги (enqueue) та видалення елементу з початку черги (dequeue).

Операції:

* **Додавання елемента (enqueue):** Додавання нового елемента до кінця черги.
* **Видалення елемента (dequeue):** Видалення елемента з початку черги. Якщо черга порожня, операція зазвичай повертає помилку або спеціальне значення.
* **Перевірка, чи порожня черга (is\_empty):** Повертає логічне значення, яке вказує, чи містить черга елементи.
* **Перегляд першого елемента (peek or front):** Повертає перший елемент черги без його видалення. Якщо черга порожня, операція зазвичай повертає помилку або спеціальне значення.
* **Отримання розміру черги (size):** Повертає кількість елементів у черзі.
* **Очистка черги (clear):** Видаляє всі елементи з черги, залишаючи її порожньою.
* **Перевірка переповнення (is\_full):** Зазвичай використовується для черг з фіксованою довжиною, щоб визначити, чи досягнуто максимальну кількість елементів.
* **Обхід черги (traverse):** Проходить через всі елементи черги, зазвичай для виконання певної операції над кожним з них.
* **Реверсування черги (reverse): Змінює порядок елементів у черзі на зворотний.**

*Черга з пріоритетами* - це черга, де кожен елемент має пріоритет. Елементи з більшим пріоритетом обробляються першими. Операції можуть включати додавання елементу з пріоритетом, видалення елементу з найбільшим пріоритетом тощо.

**1.3 Кортежі, множини, словники, одно-тадвобічнозв'язні списки. Реалізація абстрактних типів даних з оцінюванням складності операцій.**

*Кортеж* - це упорядкована колекція об'єктів різних типів. Основна властивість кортежу - це незмінність (immutable), тобто одержане значення не може бути змінене після створення кортежу.

*Множина* - це набір унікальних елементів без порядку. Основні операції множин включають додавання елементу, видалення елементу та перевірку належності.

*Словник* - це колекція пар ключ-значення, де кожен ключ унікальний. Операції словників включають додавання нової пари ключ-значення, видалення, зміну значення за ключем та пошук значення за ключем.

*Одно/двобічно-зв'язні списки* - це структури даних, які складаються з вузлів, кожен з яких містить дані та посилання на наступний (у випадку одностороннього списку) або на попередній та наступний (у випадку двостороннього списку) вузли.

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

*Алгоритми* – це набір чітко визначених інструкцій чи кроків, які виконуються у порядку для вирішення конкретної завдання чи досягнення певної мети. Їх застосування не обмежується програмуванням, вони широко використовуються у фізиці, математиці та інформатиці загалом. Алгоритм дозволяє точно описати процес розв'язання задачі, який не зможуть вплинути вхідні дані.

**Пошук** – потрібні для пошуку конкретної інформації

***Лінійний пошук*** – проходить по всіх елементах масиву послідовно до знаходження шуканого елемента.

**Просторова складність:** O(1).

**Часова складність:** У найгіршому випадку O(n), де n - кількість елементів у вхідній послідовності. У найкращому випадку складність може бути O(1). В середньому O(n).

function linearSearch(arr, target) {

for (let i = 0; i < arr.length; i++) {

if (arr[i] === target) return i; // повертаємо індекс, якщо елемент знайдено

}

return -1; // повертаємо -1, якщо елемент не знайдено

}

**Бінарний(швидкий)** – застосовується до відсортованих масивів, рекурсивно ділить масив навпіл і шукає елемент у відповідній половині.

**Просторова складність:** O(1).

**Часова складність:** У найгіршому O(log n). У найкращому випадку O(log n).

function binarySearch(arr, target) {

let left = 0;

let right = arr.length - 1;

while (left <= right) {

let mid = Math.floor((left + right) / 2);

if (arr[mid] === target) {

return mid; // повертаємо індекс, якщо елемент знайдено

} else if (arr[mid] < target) {

left = mid + 1; // зміщуємо ліву границю

} else {

right = mid - 1; // зміщуємо праву границю

}

}

return -1; // повертаємо -1, якщо елемент не знайдено

}

**Сортування** – призначені для ранжування даних.

***Вибором*** – знаходить найменший елемент у масиві і поміщає його на початок, а потім шукає наступний найменший елемент і поміщає його слідом, і так далі;

****Часова складність:**** O(n²) в найгіршому, середньому і найкращому випадках.

****Просторова складність:**** O(1), оскільки використовується постійна кількість додаткової пам’яті.

function selectionSort(arr) {

const len = arr.length;

for (let i = 0; i < len - 1; i++) {

let minIndex = i;

for (let j = i + 1; j < len; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

if (minIndex !== i) {

[arr[i], arr[minIndex]] = [arr[minIndex], arr[i]]; // обмін значень, якщо знайдено менший елемент

}

}

return arr;

}

***Вставками* (Insertion Sort)** – поступово будує відсортований масив, вставляючи елементи у правильне положення у вже впорядкованій частині масиву.

****Часова складність:****  O(n²) / O(n²) / O(n) — виникає, коли масив вже відсортований.

****Просторова складність:**** O(1), оскільки використовується постійна кількість додаткової пам’яті.

function insertionSort(arr) {

const len = arr.length;

for (let i = 1; i < len; i++) {

let current = arr[i];

let j = i - 1;

while (j >= 0 && arr[j] > current) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = current;

}

return arr;

}

***Бульбашкою(обміном) (*Bubble Sort*)*** – послідовно порівнює та змінює сусідні елементи масиву доти, доки весь масив не буде впорядкований;

****Часова складність:**** O(n²) / O(n²) / O(n) — виникає, коли масив вже відсортований (з використанням оптимізованого варіанту).

****Просторова складність:**** O(1), оскільки використовується постійна кількість додаткової пам’яті.

function bubbleSort(arr) {

const len = arr.length;

for (let i = 0; i < len - 1; i++) {

for (let j = 0; j < len - 1 - i; j++) {

if (arr[j] > arr[j + 1]) {

// обмін значень, якщо порядок неправильний

[arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];

}

}

}

return arr;

}

*Сортування (двійковим/бінарним) деревом* (*Binary Tree Sort*)— [алгоритм сортування](https://uk.wikipedia.org/wiki/Алгоритм_сортування), що полягає в побудові двійкового дерева пошуку за ключами [масиву](https://uk.wikipedia.org/wiki/Масив_(структура_даних)), а далі, в створенні результуючого масиву впорядокованих елементів виконуючи [обхід дерева](https://uk.wikipedia.org/wiki/Обхід_дерева).

*****Часова складність:***** O(n²) — виникає, коли дерево вироджується в ланцюг (це може статися, якщо елементи вставляються в відсортованому порядку). O(n log n) — для випадкових даних.

****Просторова складність:**** O(n) — для зберігання вузлів дерева.

class Node {

constructor(data) {

this.data = data;

this.left = null;

this.right = null;

}

}

class BinaryTree {

constructor() {

this.root = null;

}

insert(data) {

let newNode = new Node(data);

if (this.root === null) {

this.root = newNode;

} else {

this.insertNode(this.root, newNode);

}

}

insertNode(node, newNode) {

if (newNode.data < node.data) {

if (node.left === null) {

node.left = newNode;

} else {

this.insertNode(node.left, newNode);

}

} else {

if (node.right === null) {

node.right = newNode;

} else {

this.insertNode(node.right, newNode);

}

}

}

inOrderTraversal(node, sortedArray) {

if (node !== null) {

this.inOrderTraversal(node.left, sortedArray);

sortedArray.push(node.data);

this.inOrderTraversal(node.right, sortedArray);

}

}

}

function binaryTreeSort(array) {

let tree = new BinaryTree();

array.forEach((item) => {

tree.insert(item);

});

let sortedArray = [];

tree.inOrderTraversal(tree.root, sortedArray);

return sortedArray;

}

*Сортування Шелла* — це вдосконалений алгоритм сортування вставками, який працює шляхом порівняння елементів, що знаходяться на певній відстані один від одного, а не лише сусідніх елементів. Відстань поступово зменшується, і в кінцевому підсумку алгоритм переходить до звичайного сортування вставками. Цей підхід дозволяє ефективно сортувати частини масиву, що зменшує кількість необхідних зсувів.

****Часова складність:**** Залежить від вибору послідовності розривів. O(n^2) / O(n^(3/2)) / O(n log^2 n)

****Просторова складність:**** O(1), оскільки алгоритм сортує масив на місці.

function shellSort(arr) {

const len = arr.length;

let gap = Math.floor(len / 2); // Ініціалізуємо початковий крок

while (gap > 0) {

for (let i = gap; i < len; i++) {

let temp = arr[i];

let j = i;

while (j >= gap && arr[j - gap] > temp) {

arr[j] = arr[j - gap];

j -= gap;

}

arr[j] = temp;

}

gap = Math.floor(gap / 2); // Зменшуємо крок

}

return arr;

}

*Швидке сортування* (Quicksort)- це ефективний алгоритм сортування, який використовує стратегію "розділяй і володарюй". Він розбиває масив на менші частини, сортує кожну з них окремо, а потім об'єднує їх в один відсортований масив. Основна ідея полягає в тому, щоб обирати елемент, який називається опорним, і розміщувати всі елементи менше опорного ліворуч, а всі більше - праворуч. Далі цей процес повторюється для кожної з двох отриманих підмасивів. Швидке сортування є одним з найшвидших алгоритмів сортування, із середнім часом виконання O(n log n), де n - кількість елементів у масиві.

****Часова складність:**** O(n²) / O(n log n).

****Просторова складність:**** O(log n), оскільки рекурсивний стек займає пам'ять.

function quickSort(arr) {

if (arr.length <= 1) {

return arr; // базовий випадок - масив з одного або менше елементів вже відсортований

}

const pivot = arr[0]; // обираємо перший елемент як опорний

const left = [];

const right = [];

for (let i = 1; i < arr.length; i++) {

if (arr[i] < pivot) {

left.push(arr[i]); // додаємо менші елементи до лівого підмасиву

} else {

right.push(arr[i]); // додаємо більші елементи до правого підмасиву

}

}

return [...quickSort(left), pivot, ...quickSort(right)]; // рекурсивно сортуємо лівий та правий підмасиви та об'єднуємо їх разом

}

**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.**

*Пошук в ширину (обхід в ширину)* - це один з основних алгоритмів на графах. В результаті пошуку в ширину знаходиться шлях найкоротшої довжини у незваженому графі, тобто шлях, що містить найменшу кількість ребер. Алгоритм працює за *O*(*n*+*m*), де *n* - кількість вершин, *m* - кількість ребер.

*Пошук в глибину (обхід в глибину)* - це один з основних алгоритмів на графах. В результаті пошуку в глибину знаходиться лексикографічно перший шлях в графі. Алгоритм працює за *O*(*n*+*m*), де *n* - кількість вершин, *m* - кількість ребер.

***Кістякове дерево*** *(іноді називають каркасним деревом)* - у зв’язаному неорієнтованому графі є ациклічним підграфом, який містить усі вершини графа і не має циклів. Іншими словами, це підграф, який можна побудувати, видаляючи деякі ребра з вихідного графа, так щоб залишилася можливість дійти від будь-якої вершини до будь-якої іншої.

Властивості кістякового дерева:

* Будь-яке кістякове дерево у графі з n вершинами містить рівно n — 1 ребер.
* Кількість кістякових дерев у повному графі з n вершинами визначається формулою Келі:
* Кістякове дерево можна побудувати різними алгоритмами, такими як **алгоритм Прима**, **алгоритм Крускала** або обхід графа (наприклад, пошук в глибину або у ширину).

*Побудова найкоротших шляхів з виділеної вершини у графі* - це задача знаходження найкоротшого шляху від однієї конкретної вершини до всіх інших вершин у графі. Це може бути зручним для визначення найбільш ефективних маршрутів у транспортних мережах, в маршрутизації даних у комп'ютерних мережах, або в багатьох інших варіантах. Одним з найвідоміших алгоритмів для цієї задачі є алгоритм Дейкстри. Інший популярний алгоритм - це алгоритм Беллмана-Форда.

*Побудова найкоротших шляхів між двома вершинами у графі* - це задача знаходження найкоротшого шляху (або шляхів) між двома заданими вершинами. Це одна з ключових задач у теорії графів та має безліч застосувань у різних областях, включаючи транспортні системи, маршрутизацію даних у комп'ютерних мережах, планування маршрутів для рухомих роботів тощо.

**2. Стратегії розроблення алгоритмів**

**2.1 Стратегія «розділяй та володарюй» та приклади застосування.**

*Розділяй та володарюй* (*divide and conquer*) в інформатиці — важлива парадигма розробки [алгоритмів](https://uk.wikipedia.org/wiki/Алгоритм), 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.

Приклади:

* Алгоритм сортування злиттям (merge sort) - 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).
* Двійковий пошук — алгоритм знаходження заданого значення у впорядкованому [масиві](https://uk.wikipedia.org/wiki/Масив_(структура_даних)), який полягає у порівнянні серединного елемента масиву з шуканим значенням, і повторенням алгоритму для тієї або іншої половини, залежно від результату порівняння.

**2.2 Стратегія балансування та приклади застосування.**

При проектуванні деяких алгоритмів доводитися йти на раз-особисті компроміси, тобто по можливості збалансувати обчислювальні витрати на використання різних частин алгоритми. Метод балансування розглядається як балансування дерев. Дерево - важлива структура даних, вживана для зберігання, обробки і представлення інформації. Дерево складається з елементів (**вершин)** і зв'язків між ними (**дуг**). Серед вершин виділяється одна, яка називається коренем. Вона є батьківською по відношенню до інших пов'язаних з нею вершин. Усі вершини, пов'язані з коренем дугами, називаються нащадками. Кожна вершина в дереві, окрім кореня, має точно одну батьківську вершину і більше нащадків.

**2.3 Динамічне програмування та приклади застосування.**

*Динамічне програмування* - це метод розв'язання складних задач, який полягає в розбитті проблеми на менші підзадачі, розв'язок кожної з яких зберігається, щоб уникнути повторних обчислень. Потім результати підзадач комбінуються для отримання остаточного рішення.

Приклади:

1. Задача про найбільший підмасив: знаходження найбільшої суми елементів у підмасиві довільної довжини в масиві цілих чисел.

2. Задача про рюкзак: вибір підмножини предметів з обмеженим обсягом та вагою для максимізації загальної вартості предметів, що поміщаються в рюкзак.

**2.4 Оцінювання складності алгоритму під час застосування кожної стратегії.**

1. *Розділяй та господарюй (Divide and Conquer)*: У цій стратегії алгоритм розбиває задачу на менші підзадачі, розв'язуючи їх окремо, а потім комбінує їх рішення. Оцінка складності в основному залежить від кількості підзадач та способу їх об'єднання. Зазвичай, якщо кожна підзадача має розмір n, а об'єднання вимагає O(n) операцій, загальна складність буде O(n log n).

2. *Стратегія балансування (Balanced Strategy)*: Використовується, коли потрібно розподілити завдання рівномірно між обчислювальними ресурсами. Оцінка складності зазвичай враховує розмір простору, який потрібно розподілити, та кількість доступних ресурсів.

3. *Динамічне програмування (Dynamic Programming)*: В цій стратегії алгоритм розв'язує проблему, розбиваючи її на менші підзадачі та зберігаючи результати цих підзадач для подальшого використання. Оцінка складності зазвичай залежить від кількості підзадач та відносної складності обчислень для кожної з них. Якщо для кожної підзадачі потрібно здійснити O(1) операцій та кількість підзадач (n), загальна складність буде O(n).

**3. Моделі обчислень**

**3.1 Імперативний та декларативний підходи до програмування.**

*Імперативний підхід до програмування* - це спосіб опису програми через послідовність інструкцій, які вказують комп'ютеру, як виконувати певне завдання. У цьому підході програміст конкретно вказує, як програма має працювати, крок за кроком.

*Декларативний підхід до програмування* - це спосіб опису програми через опис потрібного результату, не вказуючи конкретних кроків для його досягнення. Програміст описує, що потрібно зробити, а не як саме це робити. Функціональне програмування є одним із видів декларативного підходу, де програма описується як множина функцій та їх взаємозв'язки.

**3.2 Розв'язні, напіврозв'язні та нерозв'язні проблеми. Проблема зупинки.**

*Роз'язні проблеми* - це ті, для яких існує алгоритм, який завжди знаходить правильний відповідь за скінченну кількість кроків. Наприклад, сортування масиву чисел або знаходження найменшого елемента у ньому.

*Напіврозв'язні проблеми* - це ті, для яких існує алгоритм, який знаходить правильну відповідь тільки для певного підмножини вхідних даних або для обмеженої кількості кроків. Проте для інших вхідних даних відповідь може бути неправильною або алгоритм може не зупинятися.

*Нерозв'язні проблеми* - це ті, для яких не існує жодного алгоритму, який може знайти правильну відповідь для будь-якого можливого вводу за скінченну кількість кроків. Один з прикладів - проблема зупинки. У загальному випадку, неможливо написати алгоритм, який би визначав, чи зупиниться довільна програма для будь-яких вхідних даних. Це питання, пов'язане з теорією обчислювальності та має глибокі математичні корені.