Міністерство освіти і науки України
Кам'янець-Подільський національний університет імені Івана Огієнка
Фізико-математичний факультет
Кафедра математики

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти «магістр»

з теми: "АЛГОРИТМИ ТА МЕТОДИ ЇХ ОПТИМІЗАЦІЇ В ШКІЛЬНИХ ОЛІМПІАДАХ"

Виконав здобувач вищої освіти
освітньої програми Середня освіта (Математика, інформатика)
спеціальності 014 Середня освіта (за предметними спеціальностями)
предметної спеціальності 014.04 Середня освіта (Математика)
денної форми здобуття вищої освіти
Горбатюк Богдан Сергійович

Керівник: **Зеленський О. В.,** кандидат фізико-математичних наук, доцент, доцент кафедри математики

Рецензент: **Кріль С. О.,**
кандидат фізико-математичних наук, доцент

Кам'янець-Подільський – 2025

ЗМІСТ

ВСТУП

Що таке алгоритми та структури даних і навіщо вони потрібні?

Алгоритм — це чітка послідовність дій, яка дозволяє вирішити конкретну задачу або досягти певної мети. Зручна аналогія — кулінарний рецепт: набір інструкцій, які вказують, що робити і в якій послідовності. Комп’ютер, так само як і людина, слідує алгоритму: від простих обчислень до складних операцій над великими масивами даних.

Пояснення алгоритмів на простих прикладах:

1) Алгоритм приготування чаю:

— Набери воду в чайник.
— Закип’яти воду.
— Візьми чашку і поклади в неї чайний пакетик.
— Залий пакетик гарячою водою і зачекай 3–5 хвилин.
— Вийми пакетик і додай цукор або лимон за смаком.

2) Алгоритм переходу вулиці:

— Подивись наліво.
— Подивись направо.
— Якщо немає машин — переходь дорогу.

Формальні властивості алгоритму. У теорії виділяють ключові вимоги: скінченність (алгоритм має завершуватися), детермінованість (кожен крок однозначний), масовість (застосовність до класу вхідних даних), результативність (отримуємо правильний результат за скінченну кількість кроків). Окремо говорять про коректність (доведення того, що алгоритм завжди повертає правильну відповідь) та про стійкість до помилок вхідних даних.

Алгоритми в реальних системах. На практиці алгоритми реалізуються на різних рівнях: від системного програмного забезпечення та баз даних до прикладних сервісів. Добре спроєктований алгоритм дає виграш на порядки, тоді як

оптимізація коду без зміни ідеї часто дає лише відсотки. Тому в олімпіадах ключове — підібрати правильний підхід і відповідну структуру даних.

Поняття складності алгоритму та кількості операцій. Кожен алгоритм має часову й просторову складність — оцінки ресурсів, необхідних для виконання. Для порівняння використовують асимптотичні нотації: O(·) — верхня межа (у гіршому випадку), Ω(·) — нижня межа, Θ(·) — точна асимптотика. Важливі також «середній» та «амортизований» випадки.

Поняття складності алгоритмів O(n)

При роботі з алгоритмами дуже важливо розуміти, скільки часу вони займають при виконанні завдань і як цей час змінюється в залежності від розміру вхідних даних. Для цього використовують поняття асимптотичної складності, або Big O Notation, яка описує швидкість виконання алгоритму відносно кількості елементів у масиві чи іншій структурі даних.
Найчастіше в алгоритмах зустрічаються такі позначення складності:
- O(1) – Константна складність: незалежно від кількості елементів, алгоритм завжди виконується за постійну кількість операцій. Наприклад, доступ до елемента масиву за індексом.
- O(n) – Лінійна складність: час виконання алгоритму пропорційний кількості елементів. Наприклад, перевірка кожного елемента масиву.
- $O(n^2)$ – Квадратична складність: кількість операцій зростає квадратично від кількості елементів. Наприклад, порівняння кожного елемента масиву з кожним іншим елементом.
- $O(\log_2 n)$ – Логарифмічна складність: час виконання зростає набагато повільніше, оскільки на кожному кроці кількість розглянутих елементів зменшується. Наприклад, бінарний пошук.

Приклади:
— Лінійний пошук у невпорядкованому масиві: O(n).
— Бінарний пошук у впорядкованому масиві: O(log n).
— Сортування вставками: $O(n^2)$ у загальному випадку.
— Злиттям/швидке сортування: O(n log n) у середньому.

— Побудова хеш-таблиці дає очікувано O(1) на вставку/пошук, але потребує якісної хеш-функції.

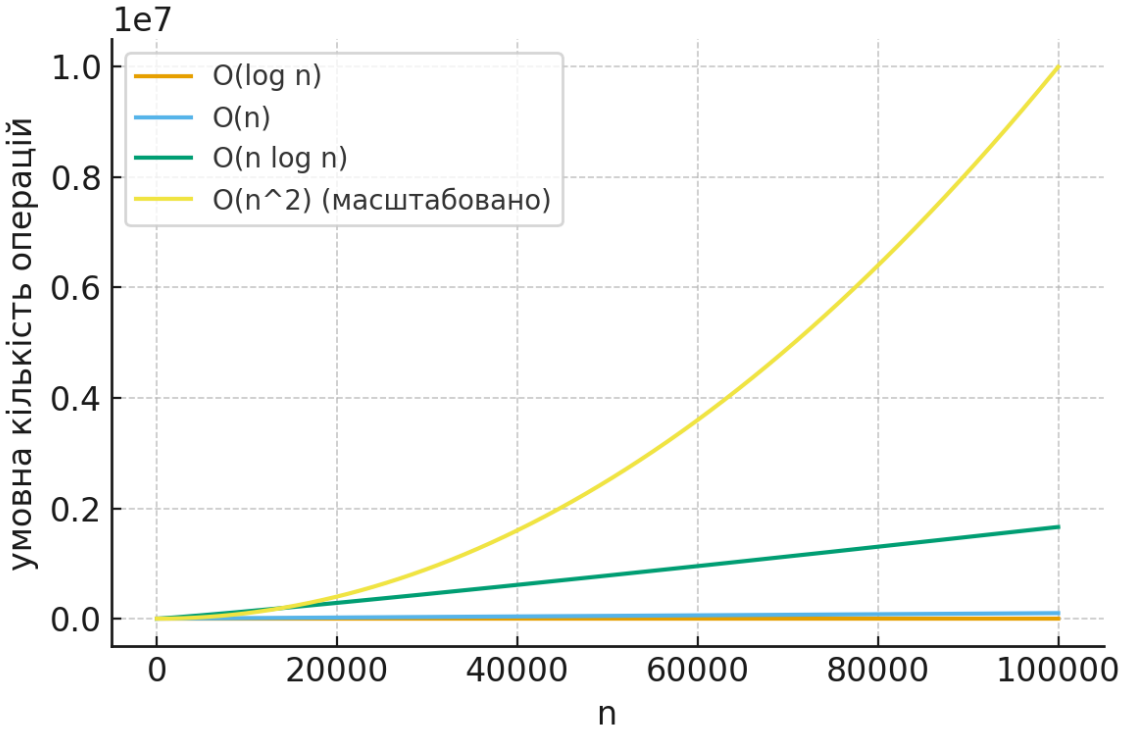

Рис. 1. Асимптотичні криві складності: O(log n), O(n), O(n log n), O(n²) (для наочності O(n²) масштабовано).

Модель обчислень (RAM). В олімпіадних задачах зазвичай припускають модель випадкового доступу до пам’яті, де елементарні операції (арифметика над малими числами, порівняння, доступ за індексом) мають сталу вартість. Це наближення дозволяє порівнювати алгоритми незалежно від конкретної архітектури.

Що таке структура даних? Це спосіб організації та зберігання даних для ефективного виконання базових операцій. Правильний вибір структури зменшує час і пам’ять, спрощує код і підвищує надійність. У цьому вступі розглянемо найуживаніші базові структури: масив, список, двохсторонній список (дек), чергу, стек і бінарне дерево.

Масив

Масив — проста, але потужна структура даних для зберігання фіксованої кількості елементів одного типу. Кожен елемент має індекс, що забезпечує доступ за O(1). Обмеження: фіксований розмір (класичний масив) і повільні вставки у середину (O(n)).

Приклад коду (Python):

```
arr = [1, 2, 3, 4, 5]
print(arr[2])  # 3
```

Список

Список — динамічний контейнер, що змінює розмір під час виконання. У Python список — це динамічний масив: амортизоване додавання в кінець — O(1), але вставки/видалення всередині можуть коштувати O(n). Може зберігати елементи різних типів.

Приклад коду (Python):

```
my_list = [1, "hello", 3.14]
my_list.append(10)
print(my_list)
```

Двохсторонній список (дек)

Дек забезпечує швидке додавання/вилучення елементів з обох кінців за O(1), корисний для «ковзних вікон», двосторонніх черг і деяких алгоритмів пошуку.

Приклад коду (Python):

```
from collections import deque


dq = deque([1, 2, 3, 4, 5])
dq.appendleft(0)
dq.append(6)
print(dq)
```

Черга (queue)

Черга працює за принципом FIFO: першим прийшов — першим вийшов. Використовується у BFS, плануванні, серверних чергах. Операції додавання/вилучення з кінців — O(1).

Приклад коду (Python):

```
from collections import deque


queue = deque([1, 2, 3, 4, 5])
queue.append(6)
queue.popleft()
print(queue)
```

Стек (stack)

Стек працює за принципом LIFO: останнім прийшов — першим обслужений. Використовується в DFS, рекурсії, обробці виразів. Операції push/pop — O(1).

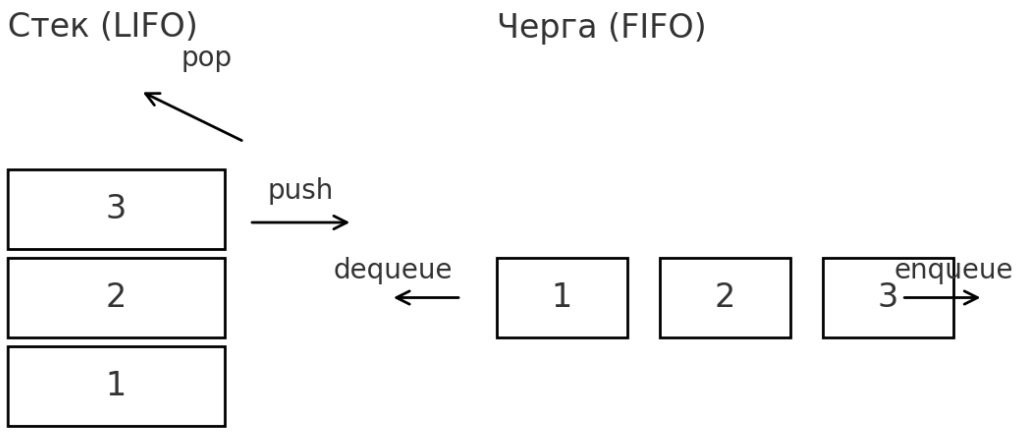

Рис. 2. Порівняння структур «стек (LIFO)» та «черга (FIFO)».

Приклад коду (Python):

```
stack = []
stack.append(1)
stack.append(2)
stack.append(3)
```

```
stack.pop()
print(stack)
```

Бінарне дерево

Бінарне дерево — структура, де кожен вузол має не більше двох нащадків. Найпоширеніший випадок — бінарне дерево пошуку (BST): усі ключі ліворуч менші за батьківський, праворуч — більші. Це дає логарифмічний доступ у збалансованих дерев, але у виродженому випадку — O(n).

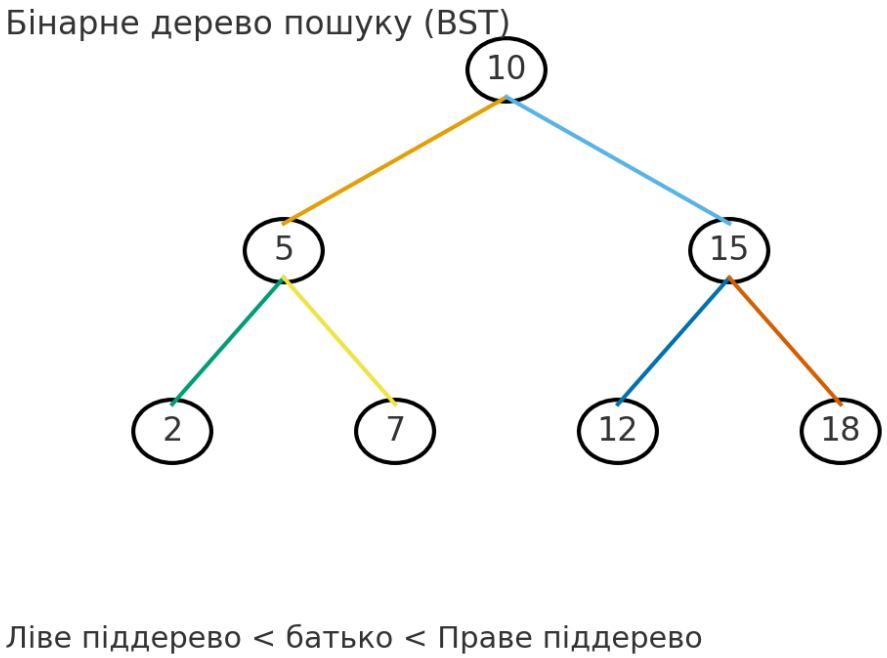

Рис. 3. Схема бінарного дерева пошуку (BST).

Приклад коду (Python) — створення та вставка у BST:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key


def insert(root, key):
```

```
    if root is None:
        return Node(key)
    if key < root.val:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

root = Node(10)
root = insert(root, 20)
root = insert(root, 5)
root = insert(root, 15)
```

Додатково — пошук і обхід (inorder):

```
def search(root, key):
    if not root or root.val == key:
        return root
    if key < root.val:
        return search(root.left, key)
    return search(root.right, key)

def inorder(root):
    if not root: return
    inorder(root.left)
    print(root.val, end=' ')
    inorder(root.right)
```

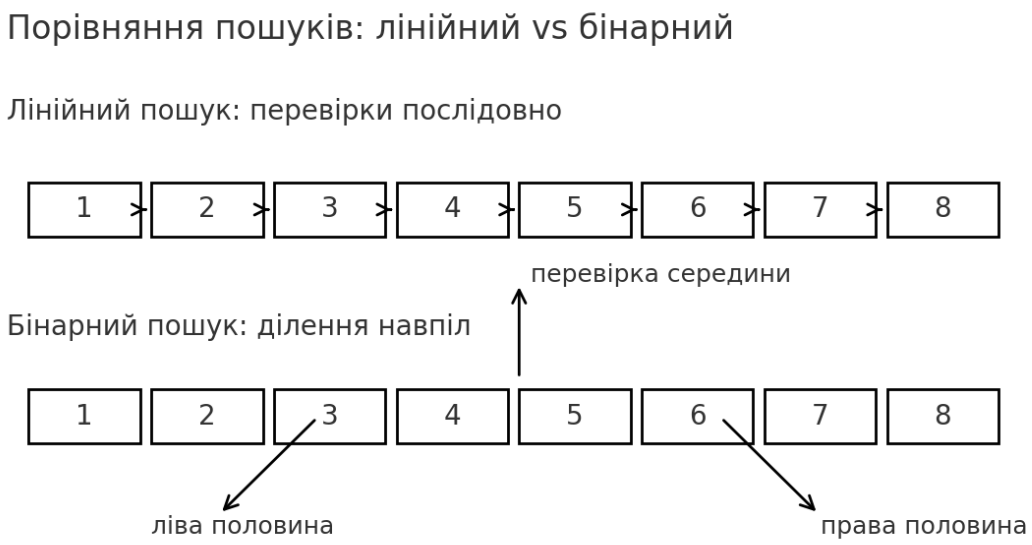

Рис. 4. Лінійний пошук послідовно перевіряє елементи, бінарний — ділить простір навпіл.

Ключові алгоритмічні парадигми, які ми використаємо далі:

— Divide & Conquer (розділяй і володарюй): рекурсивний поділ на підзадачі (приклад — сортування злиттям).
— Жадібні алгоритми: локально оптимальні кроки з доведенням глобальної оптимальності (приклад — вибір інтервалів).
— Динамічне програмування: оптимізація через підзадачі та запам’ятовування результатів (приклад — LCS, рюкзак).
— Пошук у графах: BFS/DFS, найкоротші шляхи (Dijkstra), остовні дерева (Kruskal/Prim).

Типові олімпіадні обмеження та орієнтири складності.

Розмір n	Цільова складність	Приклади алгоритмів
n ≤ 10^3	O(n^2) і швидше	вставки/вибором, підрахунком, прості перебори

n ≤ 10^5	O(n log n), O(n)	швидке/злиттям сортування, мапи/купи, лінійні обходи
n ≤ 10^6	O(n), O(n log n) (на межі)	односканерні рішення, двовказівники, radix
n ≥ 10^7	O(n) (практично), пам’ять критична	стрімінг, компресовані структури

Практичні поради для олімпіадника:

1) Спершу оцінюй порядок складності — лише потім оптимізуй константи.
2) Плануй структури даних: черга для BFS, стек для DFS, купа для задач з пріоритетами, хеш-таблиця для частот.
3) Перевіряй граничні випадки: пусті вхідні дані, мінімальні/максимальні значення, однакові елементи, повтори.
4) Тримай інваріанти: формулюй, що саме завжди істинно на кожному кроці.
5) Амортизований аналіз підкаже, коли середні витрати O(1) прийнятні (динамічні масиви, об’єднання множин).

Початок форми

Кінець форми

Алгоритми — це стратегія, структури даних — це інструменти. Разом вони визначають, чи потрапить розв’язок у часові та пам’ятні ліміти олімпіад. Далі ми послідовно розглянемо: (1) пошук і сортування, (2) жадібні алгоритми, (3) динамічне програмування, (4) поглиблені аспекти пошуку та сортування, (5) графи та алгоритми пошуку. Кожен розділ міститиме як інтуїцію, так і формальний аналіз із прикладами та тренувальними вправами.

ВИСНОВКИ

У цій роботі було систематизовано ключові алгоритмічні парадигми та структури даних, які формують кістяк сучасної алгоритміки і водночас є фундаментом розв'язання задач шкільних олімпіад з інформатики. Ми розглянули базові та ефективні методи пошуку й сортування, жадібні алгоритми, динамічне програмування, а також графові алгоритми. Кожен із цих розділів має самостійну теоретичну цінність, але їхня практична сила розкривається у взаємозв'язку: правильний вибір моделі даних, алгоритмічної парадигми та оцінки складності визначає не лише коректність, а й вкладання у суворі часові й пам'ятні ліміти олімпіадних змагань.

Важливість алгоритмів у «великій» інформатиці та в олімпіадному контексті збігається за суттю, але різниться за акцентами. У прикладних системах часто дозволені довші обчислення або розподілений запуск; в олімпіаді ж маємо типовий горизонт у 1–2 секунди на тест і обмеження пам'яті. Відтак, саме асимптотична ефективність та економність структури даних стають головним критерієм якості розв'язку. Вчитися мислити алгоритмами — це вчитися формулювати інваріанти, робити редукції, оцінювати складності, будувати докази коректності та приймати інженерні компроміси.

Розділи, присвячені пошуку та сортуванню, виконують роль «службового шару», на який спираються майже всі подальші підходи. Сортування є типовим препроцесінгом: воно зменшує складність багатьох задач із $O(n^2)$ до $O(n \log n)$ або навіть $O(n)$. Після впорядкування даних ми отримуємо можливість застосовувати бінарний пошук, метод двох вказівників, двофазні схеми «препроцесинг + пошук», лінійні об'єднання та відсікання дублікатів. Різноманіття алгоритмів сортування — від простих (вибором, вставками, бульбашкою) до ефективних (злиттям, швидким, пірамідальним) — дозволяє підібрати метод із потрібними властивостями: стабільність, in-place, гарантії у гіршому випадку, кеш-дружність, потреби в додатковій пам'яті.

З позиції олімпіадника, грамотний вибір сортування — це не лише про швидкість, а й про «поведінкові» властивості. Якщо важлива стабільність

порядку рівних елементів — доречний MergeSort; якщо критична пам’ять — Heapsort; якщо потрібна висока середня швидкість на практиці — QuickSort із випадковим вибором опорного. У задачах на цілі діапазони ключів працюють лінійні методи (Counting/Radix), які забезпечують O(n + k) і здатні різко зменшити час у порівнянні з методами на порівняннях. Таким чином, знання «портрету» кожного сортування дає тактичну перевагу: ми підбираємо інструмент під конкретні обмеження задачі.

Жадібні алгоритми вкладають у культуру розв’язування задач ідею локальної оптимальності: обирай крок, який тут і зараз виглядає найкраще, та доведи, що це веде до глобально оптимального рішення. В олімпіадах такі задачі трапляються дуже часто: розклад інтервалів, побудова мінімальних кістякових дерев, кодування Гаффмана, вибір мінімального набору купюр тощо. Їхня сила — у простоті, швидкості та прозорості реалізації; їхня слабкість — у необхідності акуратного доказу через «аргумент обміну» або структури на кшталт матроїдів. Уміння впізнати «жадібність» за формою постановки (монотонні виграші, незалежність виборів, підзадачі без «відкатів») — одна з найкорисніших компетенцій олімпіадника.

Динамічне програмування (ДП) систематизує роботу з підзадачами — воно акумулює результати та не перераховує їх повторно. Саме ДП дозволяє переходити від експоненційних переборів до поліноміальних рішень: рюкзак, LCS/LCIS/LIS, розбиття послідовностей, оптимальні розміщення дужок у множенні матриць, ДП по цифрах, по бітмасках і по деревах. Важливими є дві складові: виявлення оптимальної підструктури (яка рекурентна формула «зв’язує» відповідь із меншими підзадачами) і зменшення кількості станів (компресія пам’яті, інваріанти, порядок обходу). Для олімпіадника ДП — це не лише «набір шаблонів», а спосіб мислення: побачити вимір станів, обрати напрям переходів, довести коректність і порахувати межі за часом і пам’яттю.

Графові алгоритми — універсальна мова моделювання. Багато «неграфових» задач зручно зводяться до графів: залежності — це орієнтовані ребра; переміщення — це шляхи; покриття — це паросполучення; ресурси — це потоки. BFS/DFS формують базу для компонент зв’язності, топологічного

впорядкування та пошуку циклів; алгоритми Дейкстри, Беллмана-Форда, Флойда-Воршелла розв’язують задачі найкоротших шляхів; Краскала та Прима — мінімальні кістяки; мережеві потоки — розподіл ресурсів і перевірка досяжності певних станів. Для олімпіад це означає: навчитися «бачити граф» у постановці і підбирати потрібний інструмент без довгих спроб і помилок.

Структури даних — «плече сили» алгоритмів. Масиви дають випадковий доступ O(1), списки зручні для вставок/видалень, дека — для двосторонніх операцій, стек і черга — природна опора для DFS/BFS. Хеш-таблиці забезпечують очікувані O(1) на пошук/вставку й розкривають швидкі рішення задач на підрахунок частот, множини/словники, «бачили — не бачили». Купи/пріоритетні черги дозволяють підтримувати поточні мінімуми/максимуми, що критично в Дейкстрі, жадібних композиціях і потокових сценаріях. Збалансовані дерева (AVL, червоно-чорні) дають логарифмічні гарантії для впорядкованих асоціативних контейнерів. Усе це — будівельні блоки олімпіадних рішень.

Окремо підкреслимо важливість «інженерної дисципліни» у змаганнях. Успіх визначається не лише ідеєю, а й культурою перевірок. Типовий робочий цикл: (1) формалізація — виписати вхід/вихід, інваріанти, обмеження; (2) вибір парадигми та структури даних; (3) оцінка складності з прицілом на ліміти; (4) псевдокод і ручний прогін на крайових випадках (порожні набори, повтори, великі значення, вже відсортовані чи «погано» упорядковані дані); (5) тільки потім — реалізація; (6) локальні тести й аналіз; (7) оптимізація за потреби. Така дисципліна мінімізує дрібні помилки і економить дорогоцінний змагальний час.

Ще один вимір — правильна стратегія навчання. Стійкий прогрес забезпечують (а) систематична практика за «драбинкою» складності, (б) розбір розв’язків і власних помилок, (в) тематика «від загального до спеціального»: від масивів і сортувань — до ДП на бітмасках і ТСП-наближень, (г) ведення власного «щоденника рішень» або міні-бібліотеки шаблонів. Джерела на кшталт CP-Algorithms, USACO Guide, AtCoder Educational DP Contest і KACTL полегшують формування інтуїції, але не замінюють власного мислення — важливо не лише «знати код», а розуміти, чому саме він працює.

Змістовний внесок цієї роботи для олімпіадника полягає у виробленні «чуття адекватної моделі». Звичка починати з оцінки порядку складності, підбирати структури даних під операції (пошук, вставка, мінімум/максимум, обхід), розпізнавати ознаки жадібності, бачити стани для ДП, переводити описані процеси у графові моделі — це інструменти, що безпосередньо конвертуються у бали на змаганнях. Додатково, практики на кшталт попереднього сортування, двох вказівників, префіксних сум, «ковзного вікна», двофазних алгоритмів (preprocess → query) формують «стандартні прийоми», які зменшують час на пошук ідеї під тиском.

Водночас важливо пам’ятати про межі підходів. Жадібність не завжди дає оптимум, швидке сортування має погані сценарії, Дейкстра не працює з від’ємними вагами, а класичні хеш-таблиці деградують за поганих хешів. Здорова критика і коректність — понад усе: доводити оптимальність, перевіряти інваріанти, тестувати крайові випадки, підбирати стабільні реалізації там, де це істотно. Знання цих «підводних каменів» часто відрізняє силу рішення від удачі.

Практичне значення описаних алгоритмів виходить далеко за рамки олімпіад. Вони лежать в основі пошукових систем, мережевих протоколів, компіляторів, системних інструментів, фінансових транзакцій, біоінформатики, комп’ютерної графіки, логістики та промислової автоматизації. Навички оцінювання складності й побудови коректних алгоритмів формують універсальну компетентність, потрібну в інженерії, науці та аналітиці. Олімпіади ж створюють концентроване середовище для швидкого розвитку цієї компетентності: чіткі обмеження, негайний зворотній зв’язок і велика різноманітність постановок.

Підсумовуючи, можна виділити кілька ключових висновків. По-перше, алгоритми пошуку і сортування — базова платформа, без якої не працює більшість інших підходів; їх потрібно знати не лише «по назвах», а й у деталях реалізації та з урахуванням властивостей (стабільність, пам’ять, гірші випадки). По-друге, жадібні стратегії дають прості й швидкі рішення там, де виконуються умови коректності; тут необхідний навик доведення через інваріанти та «аргумент обміну». По-третє, динамічне програмування — найпотужніший

інструмент зниження складності при наявності оптимальної підструктури і перекриття підзадач; освоєння ДП означає навчитися бачити стани й переходи. По-четверте, графові алгоритми — універсальна мова моделювання, яка дозволяє «переписати» складні сюжети на прості операції над вершинами і ребрами. Нарешті, правильний вибір і комбінування структур даних роблять алгоритмічну ідею життєздатною у змагальних лімітах.

Головною цінністю роботи є не перелік назв, а сформована «методична оптика»: як читати постановку, у що перекладати, яку парадигму випробувати першою, як довести коректність і оцінити ресурси, коли та як застосувати препроцесинг, як організувати дані. Розвиток цієї оптики — це шлях до впевненості на олімпіадах і до професійної інженерної майстерності поза ними. Алгоритми, розглянуті у роботі, — це не лише теорія, а набір перевірених практик, що дають відчутний виграш у швидкості, надійності та ясності мислення. Саме тому їхня важливість у шкільних олімпіадах і в реальних проєктах — принципова й довготривала.

Список використаних джерел

1. Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. Introduction to Algorithms. 4th ed. Cambridge, MA: MIT Press, 2022.
2. Sedgewick, R.; Wayne, K. Algorithms. 4th ed. Addison-Wesley (Pearson), 2011.
3. Morin, P. Open Data Structures: An Introduction. Athabasca University Press, 2013.
4. Roughgarden, T. Algorithms Illuminated, Part 1: The Basics. Soundlikeyourself Publishing, 2017.
5. Roughgarden, T. Algorithms Illuminated, Part 2: Graph Algorithms and Data Structures. Soundlikeyourself Publishing, 2018.
6. Roughgarden, T. Algorithms Illuminated, Part 3: Greedy Algorithms and Dynamic Programming. Soundlikeyourself Publishing, 2019.
7. Knuth, D. E. The Art of Computer Programming, Vol. 3: Sorting and Searching. 2nd ed. Addison-Wesley, 1998.
8. Kleinberg, J.; Tardos, É. Algorithm Design. Addison-Wesley, 2005.
9. Skiena, S. S. The Algorithm Design Manual. 3rd ed. Springer, 2020.
10. Dasgupta, S.; Papadimitriou, C.; Vazirani, U. Algorithms. McGraw-Hill, 2006.
11. Ahuja, R. K.; Magnanti, T. L.; Orlin, J. B. Network Flows: Theory, Algorithms, and Applications. Prentice Hall, 1993.
12. CP-Algorithms (англомовна версія). Довідник з алгоритмів і структур даних для олімпіад. URL: https://cp-algorithms.com/
13. USACO Guide. Структурований курс і тренувальні задачі. URL: https://usaco.guide/
14. AtCoder Educational DP Contest (редакторіали та завдання з динамічного програмування). URL: https://atcoder.jp/contests/dp
15. KACTL — KTH ICPC Team Reference (довідник реалізацій для змагань). URL: https://github.com/kth-competitive-programming/kactl
16. Laaksonen, A. Guide to Competitive Programming: Learning and Improving Algorithms Through Contests. 2nd ed. Springer, 2020.