УДК 004.421.2
Єрмейчук С. Ю.
Ст. СНУ імені Лесі Українки

АНАЛІЗ АЛГОРИТМІВ ПОШУКУ

Єрмейчук С.Ю. Аналіз алгоритмів пошуку. Важливе місце у сучасні теорії алгоритмів відводиться алгоритмам пошуку, оскільки задача відшукання даних (пошук даних за заданим ключем, пошук підпослідовності у послідовності) є однією із тих задач, що має прикладне застосування. Застосування того чи іншого алгоритму пошуку для вирішення конкретної задачі є досить складною проблемою, вирішення якої потребує не лише досконалого володіння саме цим алгоритмом, але й всебічного розглядання того чи іншого алгоритму, тобто визначення усіх його переваг і недоліків.
Ключові слова: алгоритм; складність алгоритмів; часова характеристика; машина Тьюрінга; асимптотична часова складність; продуктивність алгоритму.

Ермейчук С.Ю. Анализ алгоритмов поиска. Важное место в современной теории алгоритмов отводится алгоритмам поиска.

Задача поиска связана с нахождением заданного значения, называемого ключом поиска ($search\ key$), среди заданного множества. Существует огромное количество алгоритмов поиска, так что есть из чего выбирать. Их сложность варьируется от самых простых алгоритмов поиска методом последовательного сравнения, до чрезвычайно эффективных, но ограниченных алгоритмов бинарного поиска, а также алгоритмов, основанных на представлении базового множества в иной, более подходящей для выполнения поиска форме. Последние из упомянутых здесь алгоритмов имеют особое практическое значение, поскольку применяются в реально действующих приложениях, выполняющих выборку и хранение массивов информации в огромных базах данных.
Ключевые слова: алгоритм; сложность алгоритмов; временная характеристика; машина Тьюринга; асимптотическая временная сложность; производительность алгоритма.

Yermeychuk S. Y. Analysis of the search of algorithms. Important role in the modern theory of algorithms given to algorithms search because the task of finding data (search data for a given key, search subsequence of the sequence) is one of those tasks that has practical applications. The use of a search algorithm to solve a specific problem is quite complex problem whose solution requires not only fluency by this algorithm, but also comprehensive consideration of an algorithm that determine all its advantages and disadvantages. In computer science, a search algorithm is an algorithm for finding an item with specified properties among a collection of items which are coded into a computer program, that look for clues to return what is wanted. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph. Algorithms for searching virtual spaces are used in constraint satisfaction problem, where the goal is to find a set of value assignments to certain variables that will satisfy specific mathematical equations and inequations. They are also used when the goal is to find a variable assignment that will maximize or minimize a certain function of those variables.
Keywords: algorithm; complexity of algorithms; temporal characteristics; Turing machine; asymptotic time complexity; the productivity of the algorithm.

Постановка наукової проблеми та її значення. Важливою проблемою сучасного програмування є його математизація, розробка формалізованих мов проектування алгоритмів і програм, а також їх абстрактних моделей. Засоби проектування, аналізу алгоритмів є особливо актуальними у зв'язку з важливими процесами комп'ютеризації й автоматизації діяльності суспільства. До таких засобів, зокрема, відносяться алгебри алгоритмів, орієнтовані на вирішення проблем формалізації, обґрунтування правильності, покращення алгоритмів (за обраними критеріями).

Мета статті є аналіз алгоритмів відшукання даних за заданим ключем. У відповідності до поставленої мети визначено такі **завдання дослідження**: розглянути алгоритми пошуку даних за заданим ключем; дослідити основні поняття теорії складності алгоритмів; вивчити методику оцінки часової складності алгоритмів; зробити порівняльний аналіз часової складності алгоритмів пошуку за заданим ключем.

Виклад основного матеріалу й обґрунтування отриманих результатів дослідження. Пошук – одна з найпоширеніших у програмуванні дій. Він же являє собою завдання, на якому можна випробовувати різні СТРУКТУРИ даних у міру їх появи. Існує кілька основних варіацій цієї задачі, і для них створено багато різних алгоритмів, наприклад (у статичних структурах даних): лінійний пошук, пошук діленням навпіл, пошук у таблиці, прямий пошук рядка, пошук у рядку (алгоритми Кнута-Моріса-Пратта і Боуера-Мура).

Послідовний (лінійний) пошук. Найпростішим методом пошуку елемента, який знаходиться в неврегульованому наборі даних, за значенням його ключа є послідовний перегляд кожного елемента набору, який продовжується до тих пір, поки не буде знайдений потрібний елемент. Якщо переглянуто весь набір, і елемент не знайдений – значить, шуканий ключ відсутній в наборі. Цей метод ще називають методом повного перебору. Для послідовного пошуку в середньому потрібно *N/2* порівнянь. Таким

чином, порядок алгоритму – лінійний – *O(N)*. Програмна ілюстрація лінійного пошуку в неврегульованому масиві приведена в наступному прикладі, де *a* – початковий масив, key – ключ, який шукається; функція повертає індекс знайденого елемента.

```
int LinSearch(int *a, int key)
{
    int i = 0;
    while ( (i<N) && (a[i] != key) )
        i++;
    return i;
}
```

Якщо елемент знайдено, то він знайдений разом з мінімально можливим індексом, тобто це перший з таких елементів. Рівність *i=N* засвідчує, що елемент відсутній.

Єдина модифікація цього алгоритму, яку можна зробити, – позбавитися перевірки номера елементу масиву в заголовку циклу (*i<N*) за рахунок збільшення масиву на один елемент у кінці, значення якого перед пошуком встановлюють рівним шуканому ключу – key – так званий „бар'єр" [1].

```
int LinSearch(int *a, int key)
{
    a[N] = key;
    i = 0;
    while (a[i] != key)
        i++;
    return i;  }     // i<N – повернення номера елемента
```

Поняття алгоритмічно нерозв'язуваних проблем звичайно важливе і практично значуще. Розв'язування багатьох задач, як це не парадоксально, пов'язане саме з алгоритмічною нерозв'язаністю. Але з розвитком обчислювальної техніки все більше уваги стали приділяти не просто наявності алгоритму рішення деякого класу задач, а й ефективності цих алгоритмів. При використанні алгоритмічних моделей (фінітний комбінаторний процес машини Поста, абстрактну машину Тюрінга чи машини з довільним доступом) не виникає проблем із обмеженням ресурсів (стрічка є нескінченною, а час необмеженим). Але в реальних ЕОМ і пам'ять, і час обмежені. Ось чому замало знати про існування того чи іншого алгоритму – необхідно мати уявлення про необхідні ресурси, а саме: чи може певна програма (алгоритм) розміститися в пам'яті ЕОМ; чи дасть вона результат за належний час.

Дослідженням цих питань і займається розділ теорії алгоритмів – аналіз складності алгоритмів. *Складність алгоритмів* – кількісна характеристика, яка визначає час, що необхідний для виконання алгоритму (часова складність), і об'єм пам'яті, необхідний для його розміщення (ємнісна складність). Часова та ємнісна складність тісно пов'язані між собою. Обидві є функціями від розміру вхідних даних. Складність розглядається, за звичай, для машинних алгоритмічних моделей (ЕОМ), оскільки в них час і пам'ять присутні у явному вигляді. Часова характеристика (фізичний час виконання) складності алгоритму –це величина $\tau*t$, де t – кількість дій алгоритму (елементарних команд), а τ – середній час виконання однієї операції (команди). Кількість команд t визначається описом алгоритму в певній алгоритмічній моделі і не залежить від фізичної реалізації цієї моделі. Середній час τ – величина фізична і залежить від швидкості обробки сигналів в елементах і вузлах ЕОМ. Ось чому об'єктивною математичною характеристикою складності алгоритму в певній моделі є кількість команд t.

Ємкісна характеристика складності алгоритмів визначається кількістю комірок пам'яті, що використовуються в процесі його обчислення. Ця величина не може перевищувати кількість дій t, що перемножена на певну константу (кількість комірок, які використовуються при виконанні однієї команди). В свою чергу, кількість дій t може перевищувати об'єм пам'яті (за рахунок використання повторень в одних і тих же комірках). До того ж проблема пам'яті технічно долається легше, аніж проблема швидкодії, яка має фізичне обмеження – швидкість розповсюдження фізичних сигналів (*300 км/с*). Ось чому часова складність вважається більш суттєвою характеристикою алгоритму. Слід зазначити, що часова складність алгоритму не є постійною величиною і залежить від розмірності задачі (об'єм пам'яті для зберігання даних) – кількість комірок для різних даних. Отже, *складність алгоритму* – функція, значення якої залежить від розмірності n даних задачі.

Зазвичай говорять, що час виконання алгоритму або його часова складність має порядок $T(n)$ від вихідних даних розмірністю n. Одиниця виміру $T(n)$ точно не визначена, її розуміють як кількість кроків, що виконані на ідеалізованому комп'ютері [3,4]. У більшості випадків, при визначенні часової

складності алгоритму *T(n),* розуміють максимальний час виконання алгоритму по всім вхідним даним, так звану часову складність алгоритму *Tmax(n)* у найгіршому випадку. Також розглядають і *Tavg(n)* як середній (в статистичному сенсі) час виконання алгоритму на всіх вихідних даних розмірністю n. Хоча *Tavg(n)* є досить об'єктивною мірою визначення часової складності, проте часто неможливо стверджувати рівнозначність всіх вхідних даних. Таким чином, на практиці середній час виконання алгоритму знайти складніше, ніж час у найгіршому випадку. Ось чому, зазвичай, використовують час у найгіршому випадку як міру часової складності алгоритму *T(n)*. Обчислення часової складності алгоритму *Tmin(n)* у найсприятливішому випадку хоча і буде предметом нашої уваги, проте ми повинні розуміти, що значення цієї функції не повинно суттєво впливати на вибір того чи іншого алгоритму. Це пояснюється декількома причинами: по-перше, знаючи час роботи в найгіршому випадку, можна гарантувати, що виконання алгоритму закінчиться за якийсь час, навіть не знаючи, якого вигляду буде вихідна послідовність; по-друге, на практиці «погані» входи (для яких час роботи близький до максимуму) можуть часто траплятися [2,5].

Формальний опис машини Тьюрінга. Машина Тьюрінга (МТ) складається з: нескінченної стрічки, що поділена на комірки. В кожній комірці може бути записаний один із символів кінцевого алфавіту $A=\{a_0, a_1, a_2, \ldots, a_m\}$, що називається зовнішнім алфавітом (ЗА). Для кожної машини Тьюрінга можна задати власний ЗА; керуючого пристрою, що може знаходитися в одному із внутрішнього станів $Q=\{q_1, q_2, \ldots, q_n\}$. Кількість елементів Q визначає об'єм "внутрішньої пам'яті" машини Тьюрінга. У множині Q вирізняють два спеціальні стани: початковий q_1 та кінцевий q_z (або ! – знак оклику), де z – не числовий індекс, а мнемонічна ознака кінця. Таким чином, машина Тьюрінга починає роботу в стані q_1 та, потрапивши в q_z зупиняється; каретки, що переміщуючись вздовж стрічки, може: записувати в комірку символ зовнішнього алфавіту; зміщуватися на комірку праворуч/ліворуч чи залишатися на місці. Функціонування машини Тьюрінга можна описати так: в залежності від внутрішнього стану машини Тьюрінга (q_i) та символу зовнішнього алфавіту на стрічці (a_j) відбувається запис нового символу зовнішнього алфавіту (a'_j), зміщення каретки (d) та перехід до нового внутрішнього стану (q'_i). Таким чином, робота машини Тьюрінга визначається системою команд вигляду: $q_i\, a_j \rightarrow q'_i\, a'_j\, d$ [2,3,6].

Машина з довільним доступом до пам'яті (RAM) Аналіз алгоритму полягає в тому, щоб передбачити потрібні для його виконання ресурси. Іноді оцінюється потреба в таких ресурсах, як пам'ять, пропускна здатність мережі або необхідне апаратне забезпечення, але найчастіше визначається час обчислення. Шляхом аналізу деяких алгоритмів, призначених для розв'язку однієї та тієї самої задачі, можна легко обрати найбільш ефективний з них. В процесі такого аналізу може також виявитись, що декілька алгоритмів приблизно рівноцінні, а всі інші варто відкинути.

З урахуванням того, що алгоритми реалізуються у вигляді комп'ютерних програм, в більшості випадків в якості технології аналізу алгоритмів буде використовуватись модель узагальненої однопроцесорної машини з довільним доступом до пам'яті (Random-Access Machine – RAM). В цій моделі команди процесора виконуються послідовно; операції, які виконуються одночасно, відсутні. У цю модель входять ті команди, які зазвичай можна знайти в реальних комп'ютерах: арифметичні (додавання, віднімання, добуток, ділення, обчислення остачі від ділення, наближення дійсного числа найближчим більшим чи найближчим меншим цілим числом), операції переміщення даних (завантаження значення в пам'ять, копіювання) та керуючі операції (умовне та безумовне галуження, виклик підпрограми і повернення з неї). Для виконання кожної такої інструкції потрібно певний фіксований проміжок часу. В моделі RAM є цілочисловий тип даних та тип чисел з плаваючою комою. Також передбачається, що є верхня межа розміру одного слова даних. У моделі RAM, яка розглядається, не моделюється ієрархія пристроїв пам'яті, які сьогодні розповсюджені у звичайних комп'ютерах. Таким чином, кеш та віртуальна пам'ять відсутні у RAM. Моделі, які включають в себе таку ієрархію, набагато складніше моделі RAM, тому вони можуть ускладнити роботу. Окрім цього, аналіз, який заснований на моделі RAM, зазвичай чудово прогнозує продуктивність алгоритмів, які виконуються на реальних машинах.

Аналіз навіть простого алгоритму в моделі RAM може вимагати значних зусиль. В число необхідних математичних інструментів може увійти комбінаторика, теорія ймовірностей, алгебраїчні перетворення та здатність ідентифікувати найбільш важливі доданки в формулі. Оскільки поведінка алгоритму може відрізнятись для різних наборів вхідних значень, буде потрібна методика обліку, яка описує поведінку алгоритмів за допомогою простих та зрозумілих формул [1].

Методика оцінки часової складності алгоритмів. Один із способів визначення часової ефективності алгоритмів полягає в наступному: на основі даного алгоритму потрібно написати програму

і виміряти час її виконання на певному комп'ютері для вибраної множини вхідних даних. Хоча такий спосіб популярний і, безумовно, корисний, він породжує певні проблеми. Визначений час виконання програми залежить не тільки від використаного алгоритму, але й від архітектури і набору внутрішніх команд даного комп'ютера, від якості компілятора, і від рівня програміста, який реалізував даний алгоритм. Час виконання також може суттєво залежати від вибраної множини тестових вхідних даних. Ця залежність стає очевидною при реалізації одного й того ж алгоритму з використанням різних комп'ютерів, різних компіляторів, при залучені програмістів різного рівня і при використанні різних тестових даних. Щоб підвищити об'єктивність оцінки алгоритмів, учені, які працюють в галузі комп'ютерних наук, прийняли *асимптотичну часову складність* як основну міру ефективності виконання алгоритму [5]. У більшості випадків алгоритми реалізуються програмами, які записані операторами мов програмування високого рівня. Отже, виникає необхідність запису цих алгоритмів в термінах „кроків", кожен з яких повинен виконуватися за скінчений фіксований час після трансляції їх у машинну мову будь-якої ЕОМ. Проте, точно визначити час виконання будь-якого кроку алгоритму дуже важко, адже цей час залежить не тільки від „природи" самого кроку, а й від процесу трансляції і машинних інструкцій певного комп'ютера. Ось чому замість точної оцінки ефективності алгоритму, що придатна (актуальна, валідна) для певної обчислювальної системи, прийнято використовувати менш точну, але більш загальну асимптотичну часову складність як основну міру ефективності виконання алгоритму.

Для опису швидкості росту часової складності алгоритмів використовується Θ-символіка („тета-символіка"). Наприклад, коли говорять що час виконання $T(n)$ деякого алгоритму має порядок $\Theta(n^2)$, тобто $T(n)=\Theta(n^2)$, то вважають, що існують такі константи c_1, $c_2>0$ і таке число n_0, що $c_1n^2\le T(n)\le c_2n^2$ для всіх $n\ge n_0$. Оскільки, як було зазначено вище, складність алгоритму – це функція, значення якої залежить від розмірності n даних задачі, то запис $T(n)=\Theta(n^2)$ можна подати у вигляді $f(n)=\Theta(n^2)$. Враховуючи різні значення пропорційності для n різних алгоритмів, точнішим буде використання запису $f(n)=\Theta(g(n))$, де $g(n)$ – деяка функція від n. Зміст цього запису полягає у тому , що існують такі константи $c_1,c_2>0$ і таке число n0, що $0\le c_1g(n)\le f(n)\le c_2g(n)$ для всіх $n\ge n_0$. Якщо для деякого алгоритму $f(n)=\Theta(g(n))$, то функцію g(n) вважають асимптотично точною оцінкою його складності.

Для пояснення скористаємося прикладом. Нехай часова складність деякого алгоритму $T(n)=(1/2)n^2-3n$. Перевіримо, що цей алгоритм має швидкість росту $\Theta(n^2)$, тобто $(1/2)n^2-3n=\Theta(n^2)$. Для цього зазначимо константи c_1,c_2 і число n_0, так щоб нерівність $c_1n^2\le(1/2)n^2-3n\le c_2n^2$ виконувалася для всіх $n\ge n_0$. Розділимо нерівність на n^2 і отримаємо $c_1\le 1/2-3/n\le c_2$. Таким чином, для виконання другої нерівності достатньо прийняти $c_2=1/2$. Перша ж є нерівність виконується, наприклад, при $n_0=7$ і $c_1=1/14$.

Зазначимо, що запис $f(n)=\Theta(g(n))$ включає в себе дві оцінки росту швидкості: верхню $O(g(n))$ („о від же від ен") і нижню $\Omega(g(n))$ („омега від же від ен").

Вважають, що: $f(n)=O(g(n))$, якщо знайдеться така константа $c>0$ і таке число n_0, що $0\le f(n)\le cg(n)$ для всіх $n\ge n_0$; $f(n)=\Omega(g(n))$, якщо знайдеться така константа $c>0$ і таке число n_0, що $0\le cg(n)\le f(n)$ для всіх $n\ge n_0$. Із наведеного випливає, що для функції $f(n)$ властивість $f(n)=\Theta(g(n))$ виконується тоді і тільки тоді, коли $f(n)=O(g(n))$ і $f(n)=\Omega(g(n))$. Спрощення, яких ми припускалися при оцінці складності алгоритму $T(n)$, базуються на правилах виконання операцій додавання та множення в Θ-символіці.

Правило сум. Нехай $T_1(n)$ і $T_2(n)$ – час виконання двох програмних фрагментів P_1 і P_2. $T_1(n)$ має швидкість росту $O(f(n))$, $T2(n)$ - $O(g(n))$. Тоді $T_1+T_2(n)$, тобто час послідовного виконання фрагментів P_1 і P_2, має швидкість росту $O(max(f(n),g(n))$. Для доказу цього нагадаємо, що існують константи c_1,c_2, n_1 і n_2 такі, що за умови $n\ge n_1$ виконується нерівність $T_1(n)\le c_1f(n)$, та аналогічно $T_2(n)\le c_2g(n)$, якщо $n\ge n_2$. Нехай $n_0=max(n_1,n_2)$. Якщо $n\ge n_0$, то, очевидно, що і $T_1(n)+T_2(n)\le c_1f(n)+c_2g(n)$. Звідси випливає, що при $n\ge n_0$ справедлива нерівність $T_1(n)+T_2(n)\le(c_1+c_2)max(f(n),g(n))$. Остання нерівність і означає, що $T1(n)+T2(n)$ має порядок росту $O(max(f(n), g(n)))$.

Правило добутку. Якщо $T1(n)$ і $T2(n)$ мають швидкість росту $O(f(n))$ і $O(g(n))$ відповідно, то добуток $T1(n)*T2(n)$ має швидкість росту $O(f(n)*g(n))$. З правила добутку випливає, що $O(n^2/2)$ еквівалентно $O(n^2)$.

Часто говорять, що час виконання алгоритму має порядок $T(N)$ від вхідних даних розміру N. Одиниця вимірювання $T(N)$ точно не визначена, але в більшості випадків розуміють під нею кількість інструкцій, які виконуються на ідеалізованому комп'ютері. Для багатьох програм час виконання дійсно є функцією вхідних даних, а не їх розміру. У цьому випадку визначають $T(N)$ як час виконання в найгіршому випадку, тобто, як максимум часів виконання за всіма вхідними даними розміру N. Поряд з тим розглядають $T_{ср}(N)$ як середній (в статистичному розумінні) час виконання за всіма вхідними

даними розміру *N*. Хоча $T_{ср}(N)$ є достатньо об'єктивною мірою виконання, але часто неможливо передбачити, або обгрунтувати, рівнозначність усіх вхідних даних. На практиці середній час виконання знайти складніше, ніж найгірший час виконання, так як математично це зробити важко і, крім цього, часто не буває простого визначення поняття „середніх" вхідних даних. Тому, в основному, користуються найгіршим часом виконання як міра часової складності алгоритмів [1,2].

Продуктивність алгоритму оцінюють за порядком величини. Говорять, що алгоритм має складність порядку $O\left(f\left(N\right)\right)$, якщо час виконання алгоритму росте пропорційно функції $f\left(N\right)$ із збільшенням розмірності початкових даних N. O – позначає „величина порядку".

Приведемо деякі функції, які часто зустрічаються при оцінці складності алгоритмів. Функції приведемо в порядку зростання обчислювальної складності зверху вниз. Ефективність степеневих алгоритмів звичайно вважається поганою, лінійних – задовільній, логарифмічних – хорошою.

Таблиця 1.1 - Функції, що використовуються при обчислення складності алгоритмів

Функція	Примітка
f(N)=C	C – константа
f(N)=log(log(N))	
f(N)=log(N)	
f(N)=NC	C – константа від нуля до одиниці
f(N)=N	
f(N)=N*log(N)	
$f(N)=N^C$	C – константа більша одиниці
$f(N)=C^N$	C – константа більша одиниці
f(N)=N!	тобто 1*2* … N

Оцінка з точністю до порядку дає верхню межу складності алгоритму. Те, що програма має певний порядок складності, не означає, що алгоритм буде дійсно виконуватися так довго. При певних вхідних даних, багато алгоритмів виконується набагато швидше, ніж можна припустити на підставі їхнього порядку складності. У числових алгоритмах точність і стійкість алгоритмів не менш важлива, ніж їх часова ефективність.

Аналіз складності алгоритму корисний для розуміння особливостей алгоритму і звичайно знаходить частини програми, що витрачають велику частину комп'ютерного часу. Надавши увагу оптимізації коду в цих частинах, можна внести максимальний ефект в збільшення продуктивності програми в цілому. Іноді тестування алгоритмів є найбільш відповідним способом визначити якнайкращого алгоритму. При такому тестуванні важливо, щоб тестові дані були максимально наближені до реальних даних. Якщо тестові дані сильно відрізняються від реальних, результати тестування можуть сильно відрізнятися від реальних .

Опишемо базові правила аналізу складності алгоритмів. У загальному випадку час виконання оператора або групи операторів можна розглядати як функцію з параметрами – розміром вхідних даних і/або одної чи декількох змінних. Але для часу виконання програми в цілому допустимим параметром може бути лише розмір вхідних даних.

Час виконання операторів присвоєння, читання і запису звичайно має порядок $O\left(1\right)$. Час виконання послідовності операторів визначається за правилом сум. Тому міра росту часу виконання послідовності операторів без визначення констант пропорційності співпадає з найбільшим часом виконання оператора в даній послідовності. Час виконання умовних операторів складається з часу виконання умовно виконуваних операторів і часу обчислення самого логічного виразу. Час обчислення логічного виразу часто має порядок $O\left(1\right)$. Час для всієї конструкції if-then-else складається з часу обчислення логічного виразу і найбільшого з часів, який необхідний для виконання операторів, що виконуються при різних значеннях логічного виразу. Час виконання циклу є сумою часів усіх часів виконуваних конструкцій циклу, які в свою чергу складаються з часів виконання операторів тіла циклу і часу обчислення умови завершення циклу (часто має порядок $O\left(1\right)$). Часто час виконання циклу обчислюється, нехтуючи визначенням констант пропорційності, як добуток кількості виконуваних операцій циклу на найбільший можливий час виконання тіла циклу. Час виконання кожного циклу, якщо в програмі їх декілька, повинен визначатися окремо.

Для пояснення методики оцінки часової складності алгоритмів T(n) скористаємося так званими „простими" алгоритмами. До цієї групи відносять алгоритми впорядкування обміном, впорядкування вибором та впорядкування вставками.

Впорядкування обміном. Алгоритм впорядкування обміном базується на принципі порівняння пари сусідніх елементів до тих пір, доки не будуть впорядковані всі елементи. Щоб описати основну ідею цього методу, який іноді називають методом „бульбашки", уявимо, що елементи зберігаються в послідовності (масиві), розташованому вертикально. Елементи, що мають малі значення, є більш „легкішими" і „спливають" нагору подібно бульбашкам. При першому проході уздовж масиву (перегляд починається знизу), береться перший елемент послідовності і його значення по черзі порівнюється зі значеннями наступних елементів. Якщо зустрічається елемент з більш „важким" значенням, то ці елементи міняються місцями. При зустрічі з елементом, що має більш „легше" значення, цей елемент стає „еталоном" для порівняння, і всі наступні елементи порівнюються з цим новим, більш „легшим" елементом. У результаті елемент з найменшим значенням опиниться вгорі послідовності.

Під час другого проходу вздовж масиву знаходиться елемент із другим по величині значенням, що міститься під елементом, який було знайдено при першому перегляді послідовності. Процес повторюється до тих пір, доки не будуть впорядковані всі елементи послідовності. Відзначимо, що під час другого і наступних переглядів послідовності немає необхідності переглядати елементи, знайдені за попередні перегляди, адже вони мають значення менші за значення елементів, що залишилися. Іншими словами, під час і-го перегляду не перевіряються елементи, що знаходяться на позиціях вище і.

Проілюструємо роботу алгоритму впорядкування обміном наступним прикладом, що записаний на псевдокоді.

```
(1) for i←n-1 step -1 until 1 do
(2) for j←1 step 1 until I do
(3)         if a[j] > a[j+1] then
begin
(4)                 swap(a[j], a[j+1])
end
```

Звернемо увагу на те, що підпрограма-процедура *swap* рядку (4) використовується в багатьох алгоритмах впорядкування для перестановки елементів місцями. Код цієї процедури наведено нижче:

```
procedure swap (x,y)
begin
temp ← x
x ← y
y ←temp
end
```

Для визначення часової складності алгоритму виконаємо наступні дії. Біля кожного рядку алгоритму зазначимо його вартість (число операцій) і кількість разів, за яку виконується цей рядок. Зауважимо, що рядки усередині циклу виконуються на один раз менше, ніж перевірка, оскільки остання перевірка виводить з циклу. Для кожного *j* від 1 до 2 і підрахуємо, скільки разів буде виконаний рядок (3), і позначимо це число через *kj*. Аналогічну дію виконаємо і для рядку (4).

Таблиця 1.2 - Визначення часової складності алгоритму впорядкування обміном

Команда	Вартість	Кількість виконань
for i←n-1 *step* -1 *until* 1 *do*	c_1	n
for j←1 *step* 1 *until* i *do*	c_2	n-1

if a[j] > a[j+1] *then*	c_3	$\sum_{j=1}^{i} k_j$
swap (a[j], a[j+1])	c_4	$\sum_{j=1}^{i} t_j$

Рядок вартістю *c*, що повторений *m* разів, дає внесок *cm* в загальну кількість операцій. Склавши внески всіх рядків, одержимо вираз, що позначає часову складність алгоритму впорядкування обміном:

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{j=2}^{i} k_j + c_4 \sum_{j=2}^{i} t_j$$

Обчислимо суму кількості виконань для рядку (3).

$$\sum_{j=1}^{i} k_j = \frac{1+n-1}{2}(n-1) = \frac{n(n-1)}{2}$$

Таким чином, часова складність алгоритму T(n) дорівнює:

$$T(n) = c_1 n + c_2(n-1) + c_3\left(\frac{n(n-1)}{2}\right) + c_4 \sum_{j=2}^{i} t_j =$$

$$= c_1 n + c_2 n - c_2 + \frac{c_3 n^2}{2} - \frac{c_3 n}{2} + c_4 \sum_{j=2}^{i} t_j =$$

Як бачимо, функція T(n) – квадратична, тобто має вигляд $T(n)=an^2+bn+c$, де константи *a*, *b* і *c* визначаються значеннями $c_1, \ldots c_4$.

Слід зауважити, що час роботи алгоритму залежить не тільки від *n*, але і від того, який саме масив розмірністю *n* поданий їй на вхід. Для алгоритму впорядкування обміном найбільш сприятливий випадок, коли послідовність уже впорядкована. Процедура обміну *swap* у такому разі не буде виконана жодного разу, а часова складність алгоритму $T_{min}(n)$ дорівнюватиме:

$$T_{min}(n) = \left(\frac{c_3}{2}\right) n^2 + \left(c_1 + c_2 + \frac{c_3}{2}\right) n - c_2$$

Таким чином, у найбільш сприятливому випадку час $T_{min}(n)$, необхідний для обробки масиву розміру *n*, залишається бути квадратичною функцією від *n*.

Якщо ж масив розташований у зворотному (спадному) порядку, час роботи алгоритму буде максимальним: кожен елемент *a[j]* прийдеться міняти місцями з елементом *a[j+1]* . При цьому $t_j = j$.

Обчислимо суму кількості виконань для рядку (4).

$$T_{max}(n) = \left(\frac{c_3}{2}\right) n^2 + \left(c_1 + c_2 + \frac{c_3}{2}\right) n - c_2 + c_4 \sum_{j=2}^{i} t_{j=}$$

$$= \left(\frac{c_3}{2}\right) n^2 + \left(c_1 + c_2 + \frac{c_3}{2}\right) n - c_2 + c_4\left(\frac{n^2 - n}{2}\right) =$$

$$= \left(\frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + c_2 - \frac{c_3}{2} - \frac{c_4}{2}\right)n - c_2$$

Як бачимо, у гіршому випадку час роботи алгоритму $T_{max}(n)$ також є квадратичною функцією від n.

Аналіз середнього значення часової складності алгоритму впорядкування обміном $T_{avg}(n)$ залежить від обраного розподілу ймовірностей, і на практиці реальний розподіл може відрізнятися від передбачуваного, який, зазвичай, вважають рівномірним. Іноді рівномірний розподіл моделюють, використовуючи генератори випадкових чисел. Проте, можна припустити, що в середньому обмін відбувається приблизно у $i/2$ випадках і його загальна кількість приблизно дорівнює значенню $(1+2+..+n)/2 \approx n^2/4$. У такому випадку середнє значення часової складності алгоритму впорядкування обміном $T_{avg}(n)$ можна показати формулою:

$$T_{avg}(n) = \left(\frac{c_3}{2}\right)n^2 + \left(c_1 + c_2 - \frac{c_3}{2}\right)n - c_2 + c_4\left(\frac{n^2 - n}{4}\right) =$$

$$= \left(\frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + c_2 - \frac{c_3}{2} - \frac{c_4}{2}\right)n - c_2$$

Звичайно, алгоритм впорядкування обміном можна легко модифікувати. Один із шляхів полягає у тому, що можна позбавитися великої кількості непотрібних порівнянь, запам'ятавши чи відбувався обмін на попередньому кроці. Якщо обміну не було, то послідовність вважається впорядкованою, і алгоритм закінчує роботу. Цей процес удосконалення алгоритму можна продовжити, якщо запам'ятовувати не тільки сам факт обміну, але і місце (індекс) останнього обміну. Адже зрозуміло, що всі пари елементів з індексом, меншим від j, вже упорядковані, і наступні перегляди можна закінчувати на цьому індексі. Проте, всі вдосконалення жодним чином не впливають на кількість обмінів – вони лише зменшують кількість надлишкових повторних перевірок. Нажаль, обмін двох елементів – більш ресурсоємна операція, ніж порівняння, тому всі наші вдосконалення дають лише незначний ефект [2].

Таким чином, аналіз алгоритму впорядкування обміном показав, що ніяких переваг, окрім легко запам'ятовуючої назви, цей алгоритм не має. Проте, саме на прикладі цього алгоритму ми розглянули методику визначення часової складності алгоритмів, яку надалі застосуємо при аналізі інших алгоритмів.

Висновки й перспективи подальших досліджень. Застосування того чи іншого алгоритму пошуку для вирішення конкретної задачі є досить складною проблемою, вирішення якої потребує не лише досконалого володіння саме цим алгоритмом, але й всебічного розглядання того чи іншого алгоритму, тобто визначення усіх його переваг і недоліків. Звичайно, необхідність застосування саме простих алгоритмів пошуку очевидна. Адже складні алгоритми пошуку не дають бажаної ефективності в роботі програми. Але завжди треба пам'ятати й про те, що кожний простий алгоритм пошуку поряд із своїми перевагами може містити і деякі недоліки.

В нашій статті ми розглянули деякі алгоритми пошуку та їх реалізацію мовою С++, реалізували програмне використання методів пошуку, а також дослідили переваги алгоритмів пошуку, ефективність їх використання, визначили деякі недоліки окремих алгоритмів.

Програма містить реалізацію деяких методів пошуку даних. Дане дослідження не є завершеним. Надалі буде проводитись модернізація та реструктуризація створеного програмного продукту, задля розширення області його практичного застосування. Оскільки, теорія алгоритмів – молода та не докінця досліджена наука, наповнена великим багажем знань, вдосконалення та розширення предметної області програмного продукту, і надалі буде актуальним питанням для досліджень.

1. Cormen, Thomas H. Introduction to Algorithms (2nd ed.) / [Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford]. – MIT Press and McGraw-Hill. ISBN 0-262-03293-7., 2001. – Глава 12. – 345 с.
2. Ахо А. Структуры данных и алгоритмы / Ахо А., Хопкрофт Дж., Ульман Дж. : [Пер. с англ.] // [Уч. пос.] – М. : Издательский дом «Вильямс», 2000. –384 с. : ил.
3. Ахо А. Построение и анализ вычислительных алгоритмов / Ахо А., Хопкрофт Дж., Ульман Дж. – М. : Мир, 1979. – 536 с.
4. Кормен Т. Алгоритмы : построение и анализ / Кормен Т., Лейзерсон Ч., Ривест Р. – М. : МЦНМО, 2001. – 960 с.
5. Лісовик Л. П. Теорія алгоритмів : [Навч. Посібник] / Л. П. Лісовик, С. С. Шкільняк. – К. : Видавничий поліграфічний центр : Київський університет, 2003. –163 с.
6. Прийма С. М. Теорія алгоритмів: [Навч. Посібник] / Прийма С. М. – Мелітополь : МДПУ, 2004. – 48 с.:іл.