

Міністерство освіти і науки України  
Державний заклад  
«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій

Кафедра математики та інформатики

**Щучінов Михайло Леонідович**

**РОЗРОБКА ТА ДОСЛІДЖЕННЯ СИСТЕМИ ТВОРЧИХ ЗАВДАНЬ ДЛЯ  
ФОРМУВАННЯ ЗДАТНОСТІ СКЛАДАТИ РЕКУРСИВНІ  
АЛГОРИТМИ**

**кваліфікаційна робота  
здобувача вищої освіти другого (магістерського) рівня  
освітньої програми «Інформатика»  
за спеціальністю 014.09 .Середня освіта (Інформатика)**

Особистий підпис \_\_\_\_\_ Михайло ЩУЧІНОВ

Науковий керівник \_\_\_\_\_ Ольга СМАГІНА,  
кандидат педагогічних наук, доцент  
кафедри математики та інформатики

В.о. завідувача кафедри \_\_\_\_\_ Юрій КОЗУБ,  
доктор технічних наук, професор  
кафедри математики та інформатики

## АНОТАЦІЯ

**Щучінов М. Л.**

**Тема:** Розробка та дослідження системи творчих завдань для формування здатності складати рекурсивні алгоритми.

**Спеціальність:** 014.09 «Середня освіта (Інформатика)».

**Установа:** ЛНУ імені Тараса Шевченка, 2025р.

**Магістерська робота містить:** 107 с., 17 рис. 8 таб., 33 джерела.

**Об'єкт дослідження** – рекурсивні алгоритми.

**Предмет дослідження** – творчі завдання для формування здатності складати рекурсивні алгоритми.

**Мета дослідження** – розробка та дослідження системи творчих завдань для формування здатності складати рекурсивні алгоритми.

**Результати роботи.** На основі аналізу науково-методичної літератури та мережних джерел проведено аналіз мов програмування які можливо використовувати в закладах загальної середньої освіти. Проаналізовано програми та підручники з теми «Алгоритми та програми» в закладах загальної середньої освіти. Проведено аналіз існуючих підходів до методики викладання теми «Рекурсія». Розглянуто та досліджено значення рекурсії. Розглянуто категорії задач, що дозволяють рекурсивні визначення. Проведено аналіз реалізації рекурсивних підпрограм та трудомісткості рекурсивних алгоритмів методом підрахунку вершин дерева рекурсії.

В роботі наведено методичні рекомендації щодо опису рекурсивних функцій. Розглянуто використання творчих завдань для формування здатності складати рекурсії у числових задачах, рекурсію та послідовно організовані дані, рекурсивне оброблення двійкових дерев.

**Ключові слова:** ІНФОРМАТИКА, МЕТОДИКА ВИКЛАДАННЯ ІНФОРМАТИКИ, РЕКУРСІЯ, РЕКУРСИВНІ АЛГОРИТМИ.

## ANNOTATION

**Shchuchinov Mikhail**

**Theme Use of** Development and research of a system of creative tasks for the formation of the ability to compose recursive algorithms.

**Speciality:** 014.09 "Secondary Education (Informatics)".

**Institution:** Luhansk Taras Shevchenko National University (LTSNU), 2025 year.

**Master's work of:** 107 p., 17 im, 33 sources.

**A research object of** - recursive algorithms.

**The article of research** - creative tasks for the formation of the ability to compose recursive algorithms.

**An aim of research is** - development and research of a system of creative tasks for the formation of the ability to compose recursive algorithms.

**Job performanes.** Based on the analysis of scientific and methodological literature and online sources, an analysis of programming languages that can be used in general secondary education. Programs and textbooks on the topic "Algorithms and programs" in general secondary education are analyzed. An analysis of existing approaches to the methodology of teaching the topic "Recursion". The value of recursion is considered and investigated. Categories of problems that allow recursive definitions are considered. The analysis of implementation of recursive subroutines and complexity of recursive algorithms by the method of counting the vertices of the recursion tree is carried out.

The paper presents methodological recommendations for the description of recursive functions. The use of creative tasks for the formation of the ability to compose recursions in numerical problems, recursion and sequentially organized data, recursive processing of binary trees is considered.

**Keywords:** INFORMATICS, METHODS OF TEACHING INFORMATICS, RECURSION, RECURSIVE ALGORITHMS.

## ЗМІСТ

<b>ВСТУП.....</b>	<b>6</b>
<b>РОЗДІЛ І АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ В ШКІЛЬНОМУ КУРСІ ІНФОРМАТИКИ .....</b>	<b>10</b>
1.1. Вивчення алгоритмізації та програмування в школі .....	10
1.2. Проблеми вибору мови програмування для вивчення в шкільному курсі інформатики .....	11
1.3. Аналіз програм та підручників з теми «Алгоритми та програми» .....	21
1.4. Методика викладання теми «Рекурсія» .....	34
Висновки до розділу.....	51
<b>РОЗДІЛ II РЕКУРСІЯ ТА РЕКУРСИВНІ АЛГОРИТМИ.....</b>	<b>52</b>
2.1. Рекурсивні функції та алгоритми .....	52
2.1.1. Рекурсивні функції .....	52
2.1.2. Рекурсивні процедури і функції.....	54
2.1.3. Аналіз трудомісткості рекурсивних алгоритмів методом підрахунку вершин дерева рекурсії .....	57
2.1.4. Рекурсивна реалізація алгоритмів .....	59
2.1.5. Аналіз трудомісткості алгоритму обчислення факторіала .....	61
2.2. Рекурсивні алгоритми і методи їх аналізу .....	62
2.2.1. Логарифмічні тотожності .....	62
2.2.2. Методи рішення рекурсивних співвідношень.....	63
2.2.3. Рекурсивні алгоритми .....	64
2.2.4. Основна теорема про рекурентні співвідношення.....	65
Висновки до розділу.....	65
<b>РОЗДІЛ III РОЗРОБКА СИСТЕМИ ТВОРЧИХ ЗАВДАНЬ ДЛЯ ФОРМУВАННЯ ЗДАТНОСТІ СКЛАДАТИ РЕКУРСИВНІ АЛГОРИТМИ .....</b>	<b>66</b>
3.1. Методичні рекомендації щодо опису рекурсивних функцій.....	66
3.2. Використання творчих завдань для формування здатності складати рекурсії у числових задачах .....	69

3.2.1. Рекурсія за величиною числа .....	70
3.2.2. Рекурсія за записом числа у позиційній системі числення.....	74
3.2.3. Більш складні завдання.....	77
3.3. Використання творчих завдань для формування здатності складати рекурсію та послідовно організовані дані .....	80
3.3.1. Рекурсивна робота з одновимірними масивами.....	80
3.3.2. Рекурсивна робота з файлами .....	84
3.3.3. Рекурсивна робота зі списками.....	87
3.4. Використання творчих завдань для формування здатності складати рекурсивне оброблення двійкових дерев.....	90
Висновки до розділу.....	100
<b>ВИСНОВКИ .....</b>	<b>101</b>
<b>СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ .....</b>	<b>104</b>

## ВСТУП

Яку сферу життєдіяльності людини ми б не взяли: маркетинг, політику, медицину, освіту, – без залучення комп'ютерних технологій в сучасному світі нічого не обходиться. Для кожної з цих сфер розробляються відповідні програми.

Отже, сьогодні спеціалісти ІТ – галузі є затребуваними. А саме, дедалі популярнішими стають спеціалісти в сфері розробки та просування сайтів, фахівці з напрямку кібербезпеки, штучного інтелекту, віртуальної реальності та інші.

Завдяки таким спеціалістам створюються веб-сторінки, забезпечується цілісність даних та їх безпека, розробляється різноманітне програмне забезпечення. Кожен фахівець повинен володіти такими якостями: здатність швидко навчатись, аналітичний склад розуму.

Щодо професійних навичок, актуальними є: володіння технічною англійською мовою не нижче Upper Intermediate, знання мов програмування (найчастіше це: Java, C, C++, JavaScript, PHP, Python тощо).

Щоб оволодіти даними якостями, вміннями та навичками необхідно починати як можна раніше вивчати основи програмування.

Нагальною проблемою стало вивчення шкільного курсу інформатики, зокрема питань навчання програмування у школі. Дослідники, які розглядають програмування як спосіб спілкування з комп'ютером на зрозумілій йому мові, завжди підтримували ідею навчання учнів програмування з раннього віку [1].

В Україні учні 5 – 9 класів мають засвоювати деякі елементарні знання з цієї дисципліни, згідно з навчальною програмою Міністерства освіти і науки України, а саме тему, «Алгоритми та програми». Учні засвоюють базові поняття та виконують елементарні задачі з програмування[2]. За допомогою такого середовища, як "Scratch", вчителі інформатики готують учнів до навчання алгоритмізації і програмування якісно, доступно і легко на практичних прикладах, підвищивши при цьому рівень мотивації учнів до

навчання за рахунок використання різних мультимедійних можливостей середовища та ігрових компонентів.

А завдяки ознайомленню учня з базовими поняттями, принципами алгоритмізації, полегшується подальше знайомство з мовами програмування.

За умови правильної організації вивчення інформатики, а саме тематичних модулів «Алгоритми та програми» можна зацікавити, та мотивувати підростаюче покоління в подальшому освоєнні ІТ-спеціальностей. Тема «Алгоритми та програми» дозволяє вчителю не тільки навчати учнів навичкам програмування, а й розвивати алгоритмічне мислення, яке є складовою креативного мислення.

Щоб досягти успіху, людина повинна вміти планувати свою діяльність, встановлювати пріоритети у поставлених завданнях.

За словами О. Пасічник, «в основі навчання лежить не вивчення комп'ютера, а вивчення того, як використати його можливості для розв'язання тих задач, які постають – це розуміння як потреб людини, так і перспектив і обмежень обчислювальної техніки. У результаті набуваються не рутинні навички, а фундаментальні уявлення, створюються математичні моделі, які взаємодіють з реальним світом. Дуже важливо, що набуті знання і навички мають практичну застосовність у повсякденному житті, і можуть бути пов'язані з різноманітними сферами: від медицини до мистецтва, від історичних досліджень до космічних польотів»[2, с. 11].

Використання творчих задач на заняттях з програмування значно підвищує ефективність навчального процесу. З умови задач такого типу безпосередньо не можна сказати про те, які саме знання знадобляться учням для їх розв'язання, а тому їх діяльність спрямована на виявлення шляхів розв'язання і підбір потрібних даних, відомостей та закономірностей. При вирішенні творчих задач можуть знадобитися різноманітні знання з інших предметів, наприклад з алгебри, геометрії, теорії ймовірностей тощо. Основною ознакою того, що учні знаходяться у творчому процесі є відмова від традиційних підходів до інтерпретації існуючих відомостей. До проблем

розв'язування творчих задач у різних галузях зверталось багато науковців, серед них: Е. Григорова [2], А. Давиденко [3], С. Даниленко [4], К. Кноп [7], І. Лернер [9], Ю. Мурашковський [11], С. Притуляк [13] та інші. Але сам термін "творча задача" вводили лише деякі учені, наприклад І. Лернер [9]. Що ж до терміну "творча задача з програмування", то до нього звертались багато вчених, але жоден не давав конкретного означення і не наводив їх класифікацію.

Для того, щоб перейти до означення творчої задачі з програмування, необхідно спочатку звернутись до означень таких понять, як "задача", "творча задача", "задача з програмування". Термін "задача" у тлумачному словнику російської мови трактується у різних значеннях: 1) у загальному – те, що потребує виконання, рішення; 2) у математичному – вправа, яка виконується за допомогою умовиводу, обчислення; 3) у науковому – складне питання, проблема, що вимагають дослідження і розв'язання [12].

Аналогічне розуміння даного поняття ми знаходимо і у тлумачному словнику Ушакова [16]. До терміну "задача з програмування" зверталось чимало вчених таких, як Д. Златопольський [6], С. Окулов [5], Ф. Меньшиков [10], А. Юркін [17] та інші.

Ми будемо вважати, що задача з програмування – це така задача, яка передбачає пошук алгоритму рішення задачі засобами деякої мови програмування. До терміну "творча задача" в різних галузях зверталось багато науковців, які наведені вище. Але дане поняття вводили лише деякі вчені, наприклад: "Творчою, – пише І. Лернер, – вважається задача, дії по розв'язуванні якої не детермінуються або не повністю (неоднозначно) детермінуються якимись прописами, тобто якщо розв'язуючому невідомий алгоритм розв'язання й необхідно здійснити пошук, кроки якого наперед не дані" [9, с. 81].

**Об'єкт дослідження** – рекурсивні алгоритми.

**Предмет дослідження** – творчі завдання для формування здатності складати рекурсивні алгоритми.



**Мета дослідження** – розробка та дослідження системи творчих завдань для формування здатності складати рекурсивні алгоритми.

У відповідності до мети дослідження поставлено такі завдання:

- 1) Вивчити наукову, методичну літературу з програмування та методики його навчання в закладах загальної середньої освіти.
- 2) Провести аналіз документів та підручників з інформатики з теми «Алгоритми та програми».
- 3) Провести аналіз існуючих підходів до методики викладання теми «Рекурсія».
- 4) розробити систему творчих завдань для формування здатності складати рекурсивні алгоритми.

Для розв’язування поставлених задач застосовувались такі **методи дослідження**:

- теоретичні: аналіз науково-педагогічних джерел щодо впровадження ІКТ в навчальний процес; аналіз державних нормативних документів, навчальних програм, веб-орієнтованих ресурсів, програмного забезпечення; порівняння, вивчення та узагальнення педагогічного досвіду щодо покращення процесу навчання інформатики у закладах загальної середньої освіти.

**Практичне значення одержаних результатів** дослідження полягає в тому, що:

- проведено огляд та аналіз програм і підручників з теми «Алгоритми та програми»;
- розроблено методичні рекомендації щодо опису рекурсивних функцій;
- розроблено систему творчих завдань для формування здатності складати рекурсивні алгоритми.

# РОЗДІЛ I

## АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ В ШКІЛЬНОМУ КУРСІ ІНФОРМАТИКИ

### 1.1. Вивчення алгоритмізації та програмування в школі

Оволодіння фундаментальними знаннями та практичними навичками в галузі алгоритмізації та програмування сприяє формуванню у школярів комплексу компетентностей, необхідних для успішної участі в інноваційних процесах у сфері інформаційних технологій. Засвоєння основ програмування дозволяє не лише створювати власні програмні продукти, а й ефективно використовувати існуючі програмні рішення, вносячи до них необхідні модифікації.

Впровадження вивчення алгоритмізації та програмування в освітній процес відповідає сучасним вимогам до підготовки конкурентоспроможних фахівців. Згідно з оновленими навчальними програмами, основи програмування викладаються вже з початкової школи, що сприяє ранньому розвитку алгоритмічного мислення учнів.

Виділяють два основні напрями вивчення алгоритмізації в школі:

1. Розвиток алгоритмічного мислення: формування умінь розробляти алгоритми розв'язання задач, логічно мислити та структурувати інформацію.
2. Набуття практичних навичок програмування: вивчення основних понять мов програмування, засвоєння технології створення програмних кодів та їхньої реалізації в середовищі програмування.

Останній напрям передбачає:

- *посилення фундаментальної підготовки*: формування уявлення про структуру мов програмування, принципи роботи компіляторів та інтерпретаторів, а також особливості створення програм різного рівня складності.
- *профорієнтаційну складову*: ознайомлення учнів з різноманітними професіями в галузі інформаційних технологій та

формування в них мотивації до подальшого вивчення програмування.

Дидактичні аспекти вивчення алгоритмізації в школі передбачають послідовне оволодіння учнями різноманітними алгоритмічними конструкціями. Починаючи з простих лінійних алгоритмів, учні поступово переходять до складніших циклічних алгоритмів та алгоритмів з розгалуженням. Такий підхід дозволяє сформувати у школярів міцну теоретичну базу та практичні навички програмування.

Використання візуальних засобів навчання, таких як блок-схеми, є ефективним методом для подання алгоритмів у доступній для учнів формі. Візуалізація алгоритмічних структур сприяє кращому розумінню їх логіки та взаємозв'язків між окремими елементами.

Застосування середовищ програмування з виконавцями дозволяє учням перевіряти правильність розроблених алгоритмів на практиці, аналізувати отримані результати та вносити необхідні корективи. Таким чином, учні набувають досвіду самостійного вирішення програмних задач.

Аналіз навчального процесу показав, що використання онлайн-платформи code.org є більш ефективним для вивчення основ програмування у порівнянні з традиційними інструментами, такими як Scratch. Структурована система завдань різного рівня складності, адаптованих до вікових особливостей учнів, дозволяє забезпечити індивідуальний підхід до навчання кожного учня. Вбудована система зворотного зв'язку надає можливість учням отримувати оперативну допомогу та коректувати свої помилки в процесі виконання завдань. Функціонал створення груп та відстеження прогресу кожного учня дозволяє вчителю ефективно організувати навчальний процес та забезпечити індивідуальний підхід до навчання.

## **1.2. Проблеми вибору мови програмування для вивчення в шкільному курсі інформатики**

Проблема вибору мови програмування для навчання в загальноосвітній школі є актуальною в контексті постійного розвитку інформаційних

технологій. Незважаючи на різноманітність існуючих мов програмування, вибір оптимального інструменту для навчання школярів залишається дискусійним питанням.

Основною метою викладання програмування в школі є формування алгоритмічного мислення та ознайомлення учнів з принципами побудови програмних систем. При цьому, багато дослідників підкреслюють, що не стільки сама мова програмування, скільки методичні прийоми та якість навчальних матеріалів відіграють вирішальну роль у успішності навчання.

Вибір мови програмування для шкільного курсу інформатики залежить від низки факторів, таких як:

- Простота синтаксису: мова повинна бути інтуїтивно зрозумілою для учнів.
- Наявність інструментів для налагодження: це дозволяє учням швидко виявляти та виправляти помилки у своїх програмах.
- Якість документації та навчальних матеріалів: доступність якісних підручників, посібників та онлайн-ресурсів є важливим фактором успішного навчання.
- Актуальність: мова повинна бути затребуваною на сучасному ринку праці.

З одного боку, традиційні мови програмування, такі як Паскаль, дозволяють ефективно викладати базові поняття алгоритмізації. З іншого боку, швидкий розвиток інформаційних технологій вимагає постійного оновлення навчальних програм та використання більш сучасних мов програмування, таких як Python, Java та C#.

Таким чином, вибір мови програмування для шкільного курсу інформатики є складним завданням, яке вимагає комплексного підходу та врахування як дидактичних, так і практичних аспектів.

Вибір мови програмування для навчання учнів є відповідальним завданням, яке вимагає ретельного аналізу та врахування низки факторів.

Згідно з дослідженням «Комп'ютер у школі та сім'ї» (2013), до ключових критеріїв відбору мови програмування належать:

- Технічні характеристики: транслятор мови повинен бути кросплатформним та безкоштовно доступним. Синтаксис мови має бути інтуїтивно зрозумілим, а її можливості – достатніми для розв'язання широкого кола задач.
- Зручність використання: програми, написані на обраній мові, повинні бути компактними, зрозумілими для читання та легко піддаватися модифікації.
- Сучасність та поширеність: мова програмування має бути актуальною та широко використовуватися в реальних проектах.
- Підтримка різних парадигм програмування: мова повинна дозволяти реалізовувати алгоритми в рамках структурного, функціонального та об'єктно-орієнтованого програмування.
- Просте середовище розробки: середовище програмування має бути інтуїтивно зрозумілим та не перевантаженим додатковими функціями, які можуть ускладнити процес навчання.
- Наявність консольного транслятора: консольний інтерфейс дозволяє зосередитися на основних принципах програмування та не відволікатися на графічний інтерфейс.

На сьогоднішній день існує великий вибір мов програмування — починаючи від найпростіших (тих, на яких навчаються діти молодших класів), і закінчуючи такими потужними інструментами розробки, як C ++, C #, OpenGL, Java і т. д. Перерахуємо деякі з них.

**Scratch** — це інтерактивна візуальна мова програмування, заснована на платформі Squeak, яка спеціально розроблена для навчання дітей основам програмування. Завдяки своєму динамічному характеру, Scratch дозволяє змінювати код «на льоту», що робить процес навчання більш інтерактивним та захопливим. Використовуючи Scratch, діти можуть створювати

різноманітні інтерактивні проекти, такі як анімації, ігри та презентації, що сприяє розвитку їхньої творчості та логічного мислення. Ними можна обмінюватися всередині міжнародної спільноти, яка сформувалася в мережі Інтернет»[17].

Середовище програмування Scratch є безкоштовним, його можна завантажити із сайту розробників <http://scratch.mit.edu/> і вільно використовувати з ціллю навчання.

Особливістю Scratch являється блочне програмування: для того, щоб скласти програму потрібно переміщувати, та з'єднувати графічні блоки у стеках. Кожен блок має свою форму, що виключає можливість допущення синтаксичної помилки в алгоритмічних структурах.

До основних переваг середовища програмування Скретч варта віднести:

- мультиплатформенність (Скретч коректно працює на Windows, Linux, Mac);
- легкість і зрозумілість середовища, можливість використовувати її вже в початковій школі (достатньо, щоб діти вміли читати);
- можливість вирішення творчих задач, створення інтерактивних проектів;
- інтеграція з різними предметними областями;
- орієнтація на колективну роботу через співтовариство на офіційному сайті;
- різноманітність та повнота функцій, що дає перспективи вивчення візуального програмування [18, с. 278].

**Free Pascal**, або FPC (free pascal compiler), а раніше як FPK - вільно у доступі в початкових текстах кроссплатформний 32-розрядний компілятор мови Pascal»[14, с. 215].

Копняк Н. наводить наступні переваги використання середовища Free Pascal у загальноосвітній школі:

1. Два варіанти середовища – вільно розповсюджуване (безкоштовне програмне забезпечення) та платне (Turbo Pascal, Borland Pascal).

2. Можливість одночасного використання декількох вікон введення (зручно у випадку організації введення-виведення даних через файли).
3. Можливість реалізації як процедурного, так і об'єктно-орієнтованого програмування Lazarus - середовище швидкої розробки програмного забезпечення для компілятора Free Pascal[16, с. 219].

До недоліків використання середовища Free Pascal з навчальною метою відносять:

1. Дещо обмежені можливості реалізації як структурного програмування, так і об'єктно-орієнтованого.
2. Текстовий користувацький інтерфейс (консольна програма) – неможливість використання стандартних «гарячих» клавіш ОС Windows.
3. Вікна введення (для тексту програми) та виведення (для результатів виконання програми) за замовчуванням одночасно не використовуються.
4. Незручне (громіздке) створення графічного інтерфейсу для програм користувача.
5. Англomовний інтерфейс, який учні розуміють гірше за україномовний»[16, с. 219].

**Lazarus** – вільне середовище розробки програмного забезпечення для компілятора коду на мові Pascal. Василенко О. відзначає, що «однією з переваг цього середовища програмування є те, що Lazarus є безкоштовним і вільно поширюваним на основі ліцензій GNUGeneral Public License (GNUGPL) і GNULesser General Public License (GNUL GPL), що є важливим, особливо для використання в державних освітніх установах, дозволяючи закладам освіти уникнути використання піратського програмного забезпечення»[15, с. 32].

Можна виділити деякі основні характеристики середовища програмування Lazarus, а саме:

- інтегрований редактор коду, із набором підказок, гіпертекстовою навігацією та автоматичною регенерацією коду;
- присутній редактор форм та вікно інспектора об'єктів, які дуже схожі на відповідні елементи комерційного середовища програмування Delphi;
- наявність вбудованого компілятора помилок;
- підтримка різних типів синтаксису Pascal: TurboPascal, Object Pascal, Delphi, MacPascal;
- присутній власний формат керування пакетами даних;
- наявність локалізації інтерфейсу українською мовою;
- середовище функціонування
- операційні системи Microsoft Windows (Win32, Win64), Linux, WinCE, FreeBSD, OS/2[3], MacOS X.

**Python IDLE** –безкоштовне середовище програмування з підтримкою мови Python. Це середовище працює в режимі командного рядку. Мова інтерфейсу – англійська. Це середовище підходить для використання в навчальному процесі. «Системні вимоги IDLE залежать від версії мови Python, з якою поставляються.

Так, для Python версії 3.9 системні вимоги такі:

- тактова частота процесора: 80 МГц або більшою;
- оперативна пам'ять: 256 МБ або більше;
- вільне місце на жорсткому диску: 256 МБ;
- архітектура з розрядністю 32 біт або 64 біт;
- операційна система: Windows XP/Vista/7/8/10»[19].

PyCharm – це безкоштовне середовище програмування для мови Python. Відрізняється від Python IDLE наявністю віконним інтерфейсом та вищими системними вимогами. Мова інтерфейсу – англійська.

Системні вимоги:

1. операційна система: Microsoft Windows 10/8/7/Vista/2003/XP;
2. оперативна пам'ять: мінімум 1 ГБ, рекомендовано 2 ГБ;



3. мінімальна роздільна здатність екрану: 1024x768; 4. мова програмування Python: версії 2.4 або вище»[20].

**Мова програмування C#.** Компанія Microsoft розробила мову C# в кінці 1990-х років. Автором мови є Андерс Хейльсберг. Мова C# є складовою програмної платформи Microsoft.NET. У середині 2000 року була випущена альфа-версія мови. Хейльсберг, так само як автори мов C++ і Java, не створював нових наборів команд і правил побудови синтаксису, а використав як фундамент існуючі мови, зосередився на покращеннях й інноваціях. C# має багато спільного з широко використовуваними і популярними мовами програмування C, C++, Java та PHP. Нині практично всі професійні програмісти знають ці мови, тому перехід до C# відбувається без особливих труднощів. Поряд із цим, знання, набуті у процесі оволодіння мовою C#, безперечно знадобляться під час опанування іншими мовами програмування. В останні роки мова C# стала активно використовуватись для навчання програмування. Існує надзвичайно багато on-line сервісів, що використовуються для надання допомоги всім бажаючим у вивченні цієї мови програмування.

**LightBot**— середовище програмування для школярів початкових класів (можна використовувати з 5–6 років). Необхідно запрограмувати рух «віртуального» робота. Незважаючи на простоту, за допомогою LightBot можна не тільки формувати логічне мислення учнів, але й вивчати такі поняття як «підпрограми» і «процедури».

**RoboMind**— просте середовище програмування, яке дозволяє програмувати поведінку «машинки-робота». Тут у доступній формі вивчаються популярні методи програмування та основи штучного інтелекту. Робот може програмуватися на різних мовах.

**LittleWizzard** — середовище програмування для дітей, призначене для вивчення основних елементів програмування в початковій школі. Використовуючи тільки мишку, учні отримують можливість складати програми і вивчати такі поняття, як «змінні», «вирази», «розгалуження»,

«умови» і «логічні блоки». Кожен елемент мови є інтуїтивно зрозумілим символом.

**Karel, Karel++, Karel J. Robot** — мови для початківців, вони використовуються для складання програм управління «роботом». Karel використовує власну мову програмування, Karel++ — мову програмування C++ , Karel J. Robot — версію Karel на Java.

**Logo** — мова, яка була спеціально розроблена з метою навчання дітей програмування. У сучасних реалізаціях Logo віртуальний агент під назвою «черепашка», переміщенням якого можна програмно керувати, використовується для того, щоб зробити програмування привабливим для дітей, зосередити їх увагу на побудові зображень за допомогою «черепашки». Logo створювали, з одного боку, для того, щоб допомогти дітям вивчити основні поняття програмування, а з іншого, — для їх інтелектуального розвитку у світі, де все більше використовуються інформаційно-комунікаційні технології. Logo успішно застосовується в різних сферах — від початкового знайомства з комп'ютером і програмуванням у школі до вивчення проблем штучного інтелекту і моделювання екологічних систем в університетах.

**Squeak** — сучасна, відкрита, повнофункціональна реалізація середовища і об'єктно орієнтованої мови програмування Smalltalk. Squeak використовується як засіб для створення надзвичайно широкого діапазону проектів — від мультимедійних додатків і різноманітних освітніх платформ до розробки веб-сайтів. Програмні засоби, розроблені за допомогою Squeak, надзвичайно просто «переносяться» в середовище будь-якої операційної системи, оскільки код програми виконується (інтерпретується) «віртуальною машиною» Squeak. Аналогічна технологія була використана в процесі розробки мови програмування Java.

**Середовище програмування Robolab.** Даний програмний продукт з успіхом можна використовувати для програмування конструкторів-роботів компанії Lego. Це середовище засноване на мові графічного програмування

LabVIEW— потужного середовища програмування, яке використовується інженерами й ученими в дослідницьких інститутах і промисловості. LabVIEW— це провідний інструмент для вимірювання і контролю. Він буде корисним у процесі аналізу реальних результатів у біомедицині, космічних дослідженнях, енергетичних дослідженнях і має ще багато сфер використання. Однією з основних переваг середовища Robolab є можливість наочної демонстрації результатів роботи програми шляхом її завантаження в мікропроцесор, який керує роботом. Робот у свою чергу складається з конструктора Lego [3]. Ідея створення цього робота з'явилася в Массачусетському технологічному інституті (США), але дуже швидко поширилась в інші навчальні заклади як наочний навчальний матеріал. Учень у процесі складання робота опановує основи конструювання, фундаментальні фізичні принципи і розвиває технічне мислення. На етапі програмування і виконання програми учень може відстежити помилки, які, можливо, були допущені у процесі збирання робота і можливі помилки в алгоритмі програми. Це середовище програмування чудово підходить для фахівців практично всіх категорій, оскільки має можливості як «блокового» програмування, так і програмування на таких мовах, як C, C++, Assembler і т. д. Ця особливість обумовлена тим, що мікроконтролер, що входить в набір LegoMindstorms має універсальну систему команд, до якої, використовуючи трансляцію, можна звести програму, написану на інших мовах програмування.

За статистичними даними ресурсу TIOBE ([tiobe.com](http://tiobe.com)) нами досліджено рейтинг мов програмування, які є найбільш популярними і які можна вивчати у старших класах ЗЗСО.

Індекс TIOBE (рейтинг мов програмування) – показник популярності мов програмування, що створюється і підтримується компанією TIOBE, заснованою в Ейндховені, Нідерланди ([tiobe.com](http://tiobe.com)).

Рейтинг мов програмування розраховується, виходячи з кількості результатів запитів до пошукових систем, що містять назву мови. Охоплює пошукові системи в Google, MSN, Yahoo, Baidu, Вікіпедії і Youtube. Рейтинг

оновлюється один раз на місяць. Розглянемо статистичні дані рейтингу мов програмування TIOBE станом на квітень 2024р. (рис. 1.1). Згідно з даними на рис. 1.1. можна бачити, що лідером рейтингу є мова Java (16,9%), третє місце посіла мова Python (9,1%), четверте – C++ (6,2%), Ruby (1,3%) і Visual Basic (1,2%) займають 13 та 16 місця відповідно. Мова Object Pascal (1,05%) утримує 18 місце. Мова Scratch посідає 25 місце. Станом на квітень 2024 р. та вересень 2023 р. мови Python, Ruby та Visual Basic піднялися у рейтингу, а мови C++ та Object Pascal знизили свої рейтинги. Java уже декілька років утримує свої позиції на першому місці незмінно.

Apr 2019	Oct 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.884%	-0.92%
2	2		C	16.180%	+0.80%
3	4	^	Python	9.089%	+1.93%
4	3	v	C++	6.229%	-1.36%
5	6	^	C#	3.860%	+0.37%
6	5	v	Visual Basic .NET	3.745%	-2.14%
7	8	^	JavaScript	2.076%	-0.20%
8	9	^	SQL	1.935%	-0.10%
9	7	v	PHP	1.909%	-0.89%
10	15	^	Objective-C	1.501%	+0.30%
11	28	^	Groovy	1.394%	+0.96%
12	10	v	Swift	1.362%	-0.14%
13	18	^	Ruby	1.318%	+0.21%
14	13	v	Assembly language	1.307%	+0.06%
15	14	v	R	1.261%	+0.05%
16	20	^	Visual Basic	1.234%	+0.58%
17	12	v	Go	1.100%	-0.15%
18	17	v	Delphi/Object Pascal	1.046%	-0.11%
19	16	v	Perl	1.023%	-0.14%
20	11	v	MATLAB	0.924%	-0.39%

Рис. 1.1. Рейтинг популярності мов програмування за даними індексу TIOBE станом на квітень 2024

На рис. 1.2. відображено графік популярності деяких мов програмування за рейтинговою статистикою ресурсу TIOBE. Бачимо, що постійним лідером серед мов програмування, починаючи з 2002 року, є мова Java. Не настільки популярними, але також рейтинговими є мови Python та C++. Отже,

перспективним сьогодні у старшій школі вивчати популярні мови Java, Python та C++.

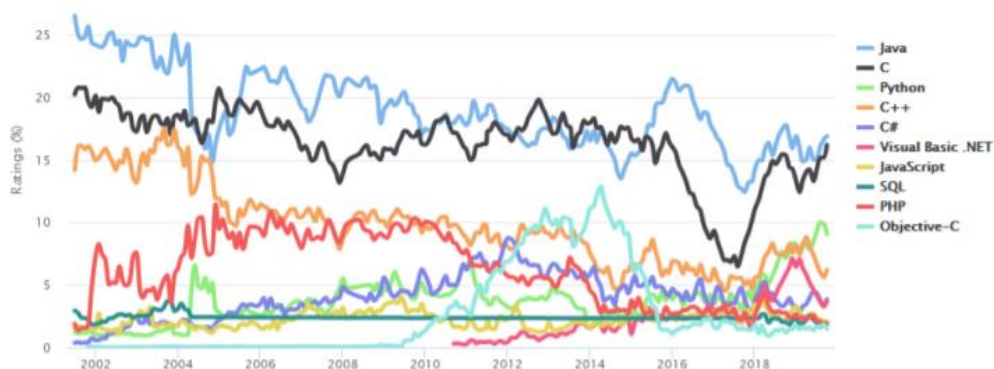


Рис. 1.2. Популярність найреєнтинговіших мов програмування (2002-2024 роки)

Весь процес навчання програмування в школі розбивається на кілька етапів. Перед початком навчання вчитель стикається з проблемою вибору мови програмування для вивчення.

Одна група вчителів розглядає тему «Алгоритмізація та програмування» на основі формальних алгоритмів, побудувавши навчання учнів на мові блок-схем. Інша група вчителів інформатики навчають учнів тієї мови програмування, за допомогою якої вміють вирішувати завдання, знають основи відповідної мови і використовують правильну методику вивчення обраної ними мови програмування. Цим втрачається єдиний освітній інформаційний простір не тільки країни, а й окремих регіонів.

Таким чином, на початку вивчення програмування вчителю необхідно вибрати мову програмування з урахуванням власної компетентності, інтересів учнів та структури навчання інформатики в школі.

### 1.3. Аналіз програм та підручників з теми «Алгоритми та програми»

Нормативною базою для вивчення інформатики в 5-9 класах загальноосвітніх навчальних закладів у 2023/2024 навчальному році слугують навчальні програми, затверджені Міністерством освіти і науки України та розміщені на офіційному веб-сайті відомства (<http://mon.gov.ua/>). Ці програми визначають цілі, завдання та зміст навчання інформатики.

Зміст навчального предмета «Інформатика» містить фундаментальну складову, що реалізується шляхом вивчення основ науки «Інформатика», має прикладну спрямованість, яка реалізується в процесі виконання учнями практичних завдань з використанням комп'ютера у формі, яку добирає вчитель: вправ, практичних, контрольних чи тематичних робіт, розв'язування компетентнісних задач, виконання індивідуальних і групових навчальних проектів тощо, а також застосування інших організаційних форм діяльності учнів й інноваційних методів навчання.

Курс «Інформатика» вибудовується за такими *предметними змістовими лініями*:

- інформація, інформаційні процеси, системи, технології;
- комп'ютер як універсальний пристрій для опрацювання даних;
- телекомунікаційні технології;
- інформаційні технології створення й опрацювання інформаційних об'єктів;
- моделювання, алгоритмізація й програмування.

З метою дотримання принципів науковості і доступності програмою передбачено послідовне ускладнення навчального матеріалу кожної з названих вище змістових ліній та умовне виокремлення двох змістових рівнів.

*Перший рівень* (5–7 класи) – продовження розпочатого в початковій школі ознайомлення з базовими поняттями курсу (*табл. 1.1.*). На цьому рівні не ставиться завдання глибокого та вичерпного вивчення ІКТ, а зроблено акцент на набутті навичок їх практичного застосування, а також на розвивальній спрямованості навчання. З метою врахування вікових особливостей учнів допускається використання навчально-імітаційних програмних засобів і середовищ, зокрема для підтримки вивчення розділу «Алгоритми і програми».

**Розділи курсу в 5–7 класах**

5 клас	6 клас	7 клас
<ul style="list-style-type: none"> <li>● Інформаційні процеси та системи</li> <li>● Мережеві технології та Інтернет</li> <li>● Опрацювання текстових даних</li> <li>● Алгоритми та програми</li> </ul>	<ul style="list-style-type: none"> <li>● Комп'ютерні презентації</li> <li>● Комп'ютерна графіка</li> <li>● Алгоритми та програми</li> </ul>	<ul style="list-style-type: none"> <li>● Служби Інтернету</li> <li>● Опрацювання табличних даних</li> <li>● Алгоритми та програми</li> </ul>

*Другий рівень* (8–9 класи) — повноцінне формування ключових та предметних ІТ-компетентностей (табл. 4). На цьому рівні, зокрема, має формуватися понятійний апарат, достатній для набуття вищезазначених компетентностей. Для цього рекомендується використовувати повнофункціональні, а не імітаційні, програмні засоби та середовища.

Таблиця 1.2

**Розділи курсу у 8–9 класах**

8 клас	9 клас
<ul style="list-style-type: none"> <li>● Кодування даних та апаратне забезпечення</li> <li>● Опрацювання текстових даних</li> <li>● Створення та публікація веб-ресурсів</li> <li>● Опрацювання мультимедійних об'єктів</li> <li>● Алгоритми та програми</li> </ul>	<ul style="list-style-type: none"> <li>● Програмне забезпечення та інформаційна безпека</li> <li>● 3D-графіка</li> <li>● Опрацювання табличних даних</li> <li>● Бази даних. Системи керування базами даних</li> <li>● Алгоритми та програми</li> </ul>

Очікувані результати навчання вказано у змістовому розділі програми для кожної теми курсу в кожному класі. Час, що необхідний для досягнення цих результатів, визначається вчителем залежно від рівня попередньої підготовки учнів, обраної методики навчання, наявного обладнання тощо. Однак на опанування тем змістової лінії «Моделювання, алгоритмізація та

програмування» має приділятися не менше 40 % навчального часу в 5–8 класах і не менше 30 % у 9 класі. За необхідності вчитель може змінювати порядок вивчення тем, не порушуючи змістових зв'язків між ними.

Очікувані результати навчання та зміст навчального матеріалу з теми «Алгоритми та програми»

Таблиця 1.3

### Алгоритми та програми у 5 класі

Очікувані результати навчально-пізнавальної діяльності учнів	Зміст навчального матеріалу
<b>Алгоритми та програми</b>	
<p><b>Учень/учениця</b>  <b>Знаннєва складова</b>  Пояснює поняття алгоритму та програми.  Наводить приклади виконавців та команд, які вони виконують.  Пояснює сутність алгоритмічних структур.</p> <p><b>Діяльнісна складова</b>  Складає прості алгоритми.  Розрізняє алгоритмічні структури.  Використовує середовище для опису та виконання алгоритмів.  Обирає алгоритмічні структури для розв'язування поставленої задачі.  За необхідності коригує алгоритми.  Виконує алгоритми, подані у формальному вигляді</p> <p><b>Ціннісна складова</b>  Усвідомлює значущість алгоритмів у житті.  Робить висновки про відповідність результату виконання алгоритму поставленій задачі</p>	<p>Виконавці алгоритмів та їхні системи команд.</p> <p>Способи опису алгоритму.  Програма.</p> <p>Середовище опису й виконання алгоритмів.  Лінійні алгоритми.</p> <p>Алгоритми з розгалуженнями.</p> <p>Алгоритми з повтореннями</p>



## Алгоритми та програми у 6 класі

Очікувані результати навчально-пізнавальної діяльності учнів	Зміст навчального матеріалу
Алгоритми та програми	
<p><b>Учень/учениця</b></p> <p><b>Знаннєва складова</b></p> <p><i>Знає і розуміє</i> поняття об'єкта в програмуванні. Наводить приклади властивостей об'єктів та їх значень.</p> <p><i>Пояснює</i> поняття події та наводить приклади подій та їх опрацювання.</p> <p><i>Знає і розуміє</i> поняття вкладених алгоритмічних структур, наводить приклади їх застосування</p> <p><b>Діяльнісна складова</b></p> <p><i>Розкладає</i> задачу на підзадачі і розв'язує їх (здійснює декомпозицію задачі).</p> <p><i>Додає</i> об'єкти до програмного проекту.</p> <p><i>Уміє</i> змінювати значення властивостей об'єктів, у тому числі програмно.</p> <p><i>Уміє</i> перевіряти результат виконання програми на відповідність умові задачі.</p> <p><i>Програмує</i> опрацювання подій.</p> <p><i>Застосовує</i> вкладені алгоритмічні структури повторення та розгалуження</p> <p><b>Ціннісна складова</b></p> <p><i>Усвідомлює</i> доцільність застосування подійного програмування для розв'язання конкретної задачі.</p> <p><i>Обґрунтовує</i> необхідність застосування вкладених алгоритмічних структур.</p>	<p>Поняття про об'єкт у програмуванні. Властивості об'єкта. Створення програмних об'єктів.</p> <p>Поняття події. Види подій. Програмне опрацювання події.</p> <p>Змінювання значень властивостей об'єкта в програмі.</p> <p>Вкладені алгоритмічні структури повторення та розгалуження.</p> <p>Розв'язання задачі методом поділу на підзадачі</p>

## Алгоритми та програми у 7 класі

Очікувані результати навчально-пізнавальної діяльності учнів	Зміст навчального матеріалу
<b>Алгоритми та програми</b>	
<p><b>Учень/учениця</b>  <b>Знаннєва складова</b>  Пояснює поняття величини, змінної та операції присвоювання.  Знає базові алгоритми роботи зі змінними: обмін значеннями, визначення найбільшого й найменшого з двох значень</p> <p><b>Діяльнісна складова</b>  Використовує різні алгоритмічні структури та змінні для розв'язання навчальних і життєвих задач.  Застосовує засоби програмування для побудови моделей</p> <p><b>Ціннісна складова</b>  Усвідомлює роль програмування та моделювання для розв'язання навчальних і життєвих задач</p>	<p>Величини. Змінні. Вказівка присвоювання.</p> <p>Створення алгоритмів і програм з використанням змінних і різних алгоритмічних структур: лінійних, розгалужень і повторень.</p> <p>Опис моделей у середовищі програмування</p>

## Алгоритми та програми у 8 класі

Очікувані результати навчально-пізнавальної діяльності учнів	Зміст навчального матеріалу
<b>Алгоритми та програми</b>	
<p><b>Учень/учениця</b>  <b>Знаннєва складова</b>  Розуміє призначення мови програмування та основних її елементів. Наводить приклади сучасних мов програмування.  Знає відмінність між змінними та константами.  Порівнює особливості різних середовищ програмування.</p>	<p>Сучасні мови програмування.</p> <p>Поняття об'єкта в мові програмування, його властивостей і методів. Графічний інтерфейс, основні компоненти програми з графічним інтерфейсом. Поняття елемента керування. Обробники подій, пов'язаних з елементами керування. Властивості та методи</p>

Очікувані результати навчально-пізнавальної діяльності учнів	Зміст навчального матеріалу
<p><i>Розуміє</i> поняття об'єкта в мові програмування, його властивостей і методів.</p> <p><i>Пояснює</i> структуру програми.</p> <p><i>Пояснює</i> функції елементів графічного інтерфейсу та користується ними.</p> <p><i>Розрізняє</i> властивості і методи елементів управління</p> <p><b>Діяльнісна складова</b></p> <p><i>Планує</i> процес розв'язування задачі з використанням програмування.</p> <p><i>Створює і налагоджує</i> програми, зокрема подійно- й об'єктно-орієнтовані.</p> <p><i>Використовує</i> в програмах вирази, коректно добирає типи даних.</p> <p><i>Розв'язує</i> задачі з використанням усіх базових алгоритмічних структур, змінних та констант.</p> <p><i>Обґрунтовує</i> вибір типів даних для розв'язування задачі</p> <p><b>Ціннісна складова</b></p> <p><i>Оцінює</i> відповідність результатів виконання програми поставленим задачам.</p> <p><i>Розпізнає</i> задачі, для розв'язання яких доцільно використовувати засоби програмування</p>	<p>елементів керування.</p> <p>Типи даних у програмуванні. Структура програми. Введення й виведення даних. Вирази. Логічні вирази та змінні й операції над ними. Умовні оператори (коротка та повна форма). Складені умови. Оператори циклу. Вкладені цикли. Пошук найбільшого та найменшого серед кількох значень</p>

Таблиця 1.7

### Алгоритми та програми у 9 класі

Очікувані результати навчально-пізнавальної діяльності учнів	Зміст навчального матеріалу
<b>Алгоритми та програми</b>	
<p><b>Учень/учениця</b></p> <p><b>Знаннєва складова</b></p> <p><i>Пояснює</i> принцип організації даних за допомогою одновимірних масивів.</p> <p><i>Пояснює</i> поняття масиву, елемента</p>	<p>Поняття одновимірного масиву. Введення й виведення значень елементів масиву.</p> <p>Алгоритми опрацювання масивів:</p>

Очікувані результати навчально-пізнавальної діяльності учнів	Зміст навчального матеріалу
<p>масиву, індексу та значення елемента.  <i>Описує</i> алгоритми опрацювання елементів масиву, що задовольняють певній умові.  <i>Описує</i> алгоритм знаходження підсумкових величин у масиві.  <i>Описує</i> принаймні один алгоритм впорядкування масиву  <b>Діяльнісна складова</b>  <i>Складає й описує</i> мовою програмування алгоритми для опрацювання елементів масиву, що задовольняють певну умову, знаходження підсумкових величин у масиві та його впорядкування  <b>Ціннісна складова</b>  <i>Оцінює</i> часову та ємнісну складність алгоритмів.  <i>Усвідомлює</i> важливість застосування ефективних методів для опрацювання великих наборів даних</p>	<p>знаходження підсумкових величин, зокрема для елементів, що задовольняють задані умови, а також пошук у масиві за певними критеріями.</p> <p>Алгоритми впорядкування масиву.</p> <p>Поняття складності алгоритмів</p>

В Україні, починаючи з 2018 року, є чинними дві навчальні програми вибірково-обов'язкового предмету «Інформатика» для учнів старших класів закладів загальної середньої освіти (ЗЗСО) ([mon.gov.ua](http://mon.gov.ua)):

- Інформатика (рівень стандарту);
- Інформатика (профільний рівень).

Навчальна програма складається з двох частин – базового та вибіркового (варіативного) модулів. Основним для вивчення у старших класах є базовий модуль, зміст якого може бути розширеним за рахунок вибірових модулів.

Саме при вивченні розділів базового модуля відбувається завершення формування в учнів предметних і ключових компетентностей щодо використання сучасних інформаційно-комунікаційних технологій на рівні, визначеному чинним

Державним стандартом базової і повної загальної середньої освіти ([zakon.rada.gov.ua](http://zakon.rada.gov.ua)).

Базовий модуль є тим мінімумом, при вивченні якого є недопустимим рознесення навчального матеріалу на два роки. Вибіркові модулі слугують для розширення курсу інформатики, і учитель має можливість обирати їх самостійно, спираючись на профіль навчання ЗЗСО, запитів, індивідуальних інтересів і здібностей учнів, особливостей регіону проживання, наявності у кабінеті інформатики матеріально-технічної бази та необхідного програмного забезпечення.

За допомогою поєднання базового та вибіркового модулів вивчення інформатики у старшій школі забезпечує необхідну гнучкість та свободу у відборі і комплектації необхідного навчального матеріалу для навчання учнів і реалізації дидактичних цілей.

Розглянемо особливості навчальних програм з інформатики стандартного та профільного рівнів для учнів старших класів (табл. 1.8).

Таблиця 1.8

### Характеристика навчальних програм рівня стандарт та профільного рівня

Характеристика	Рівень стандарту	Профільний рівень
Загальний обсяг навчального плану	105 годин	350 годин
Обсяг базового модуля	35 годин	350 годин
Обсяг вибірових модулів	70 годин (2 модулі по 35 годин)	-
Тематики базового модуля	<ul style="list-style-type: none"> <li>Інформаційні технології в суспільстві.</li> <li>Моделі і моделювання.</li> <li>Аналіз та візуалізація даних.</li> <li>Системи керування базами даних.</li> <li>Мультимедійні та гіпертекстові документи</li> </ul>	<ul style="list-style-type: none"> <li>Мова програмування та структури даних</li> <li>Сучасні інформаційні технології</li> <li>Аналіз і візуалізація даних</li> <li>Графіка\мультимедіа</li> <li>Електронні публікації</li> <li>Бази даних</li> <li>Алгоритми</li> <li>Веб-технології</li> </ul>

Характеристика	Рівень стандарту	Профільний рівень
		<ul style="list-style-type: none"> <li>Парадигми та технології програмування</li> </ul>
Розділи вибіркового модуля	<ul style="list-style-type: none"> <li>Графічний дизайн</li> <li>Комп'ютерна анімація</li> <li>Тривимірне моделювання</li> <li>Математичні основи інформатики</li> <li>Інформаційна безпека</li> <li>Веб-технології</li> <li>Основи електронного документообігу</li> <li>Бази даних</li> <li>Формальна логіка</li> <li>Комп'ютерні технології опрацювання звукової інформації</li> <li>Креативне програмування</li> </ul>	

Згідно з таблицею 1.7 бачимо, що обидва рівні мають істотну різницю в обсязі. Це пояснюється тим, що для профільного рівня інформатика вивчається по 5 годин на тиждень, 175 годин на рік, у той час, коли на рівні стандарту вивчення інформатики займає тільки 1 годину на тиждень у 10-х класах та 2 години на тиждень у 11-х класах.

Варто зазначити, що для профільного рівня відсутній вибіркового модуль, і всі основні розділи інформатики вивчаються на базовому рівні. Стосовно переліку тем двох навчальних програм зазначимо, що окремі теми співпадають, а деякі взагалі не розглядаються.

Наприклад, розділ «Інформаційні технології в суспільстві» базового модуля рівня стандарт має певну схожість з розділом «Сучасні інформаційні технології» профільного рівня; розділ «Аналіз та візуалізація даних»

присутній взагалі в обох навчальних програмах; «Системи керування базами даних» базового модуля рівня стандарт є певним відбитком вибіркового модуля «Бази даних» цього ж рівня і розділу «Бази даних» профільного рівня. Зв'язки тем навчальних програм між собою представлені на рис. 1.3

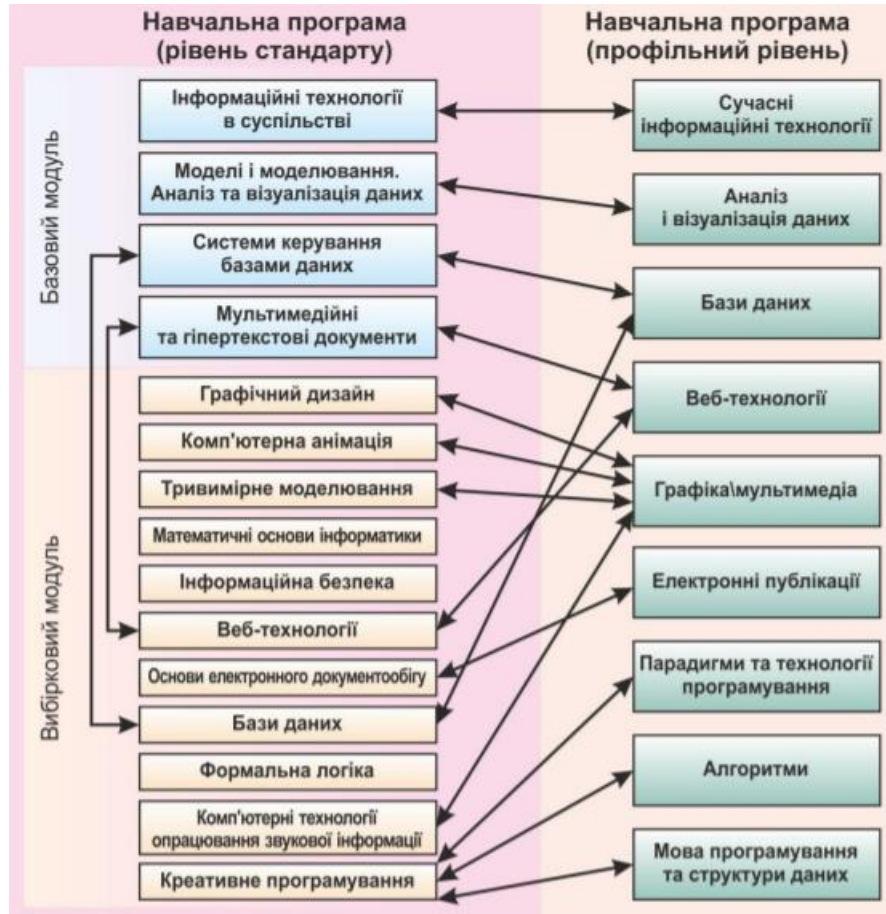


Рис. 1.3. Зв'язок розділів інформатики між рівнем стандарту та профільним рівнем

Як бачимо, програмування вивчається як у звичайних класах, так і в профільних. Але відмінності є у змісті самих тем.

Так, на рівні стандарту програмування вивчається у вибіркому модулі «Креативне програмування». Як зазначено у навчальній програмі (mon.gov.ua), цей розділ передбачає вивчення наступних тем (рис. 1.4).

На профільному рівні (mon.gov.ua) вивчення програмування переплітається відразу з трьома розділами: «Мова програмування та структури даних», «Алгоритми», «Парадигми та технології програмування». Але основи

програмування, мова та синтаксис, структура та правила написання програм вивчаються саме в першому розділі.



Рис. 1.4. Теми вибіркового модуля «Креативне програмування»

Варто зазначити, що у змісті навчального матеріалу розділу, що стосується вивчення програмування, не зазначено конкретної мови програмування для розгляду. Це значить, що вибір тієї чи іншої мови залишається за вчителем, і вже він розробляє конспекти уроків та інструкції до лабораторних робіт відповідно до обраної мови та змісту навчального матеріалу, який зазначено у навчальних програмах (Глинський & Палюшок, 2013).

Зазначимо, що вивчення програмування є однією з найбільш складних тем курсу інформатики в старшій школі. Більшість вчителів на початку XXI століття викладали мову програмування Basic. Потім у багатьох школах вивчалася мова Pascal, яка вважалася більш прийнятною з методичної точки зору для вивчення основних принципів програмування. Мова Pascal є навчальною структурною мовою програмування, яка передбачає не тільки вивчення алгоритмічних конструкцій, формування логічного і алгоритмічного



мислення в учнів, а й вирішення складних технологічних і виробничих завдань. Наразі вже більшого поширення набувають мови програмування C, C++, Visual C ++, Delphi, Java, Python та інші. Найчастіше для вивчення в старшій школі обирають наступні мови програмування: Scratch, Visual Basic, Pascal, Object Pascal, C++, Python, Java та Ruby.

Динаміка використання мов програмування у шкільних підручниках з інформатики вказує на зростаючу роль Python у навчальному процесі.

Якщо у підручниках для 5-6 класів традиційно використовувалася мова Scratch, то для старших класів (7-9) до 2020 року основною мовою була Free Pascal в середовищі Lazarus. Однак, починаючи з 2020 року, спостерігається тенденція до переходу на мову Python. Зокрема, в підручниках під редакцією Н.В. Морзе та її колег, Python стала основною мовою програмування, починаючи з 8-го класу.

Цікаво відзначити, що деякі автори поєднують вивчення різних мов програмування. Наприклад, у підручнику для 7-го класу (Морзе&Барна, 2020) паралельно вивчаються Python і Scratch, що дозволяє учням ознайомитися з різними парадигмами програмування.

Якщо брати до уваги старшу школу, то, як вже було сказано, на рівні стандарту вивчення програмування можливе тільки у вибіркового модулі «Креативне програмування». Відповідно підручників, де були відображені навчальні матеріали з даного модуля нема.

Зокрема, у підручнику Руденка, Речича та Потієнко (2023) для 10-го класу профільного рівня Python використовується для вивчення основ алгоритмізації, об'єктно-орієнтованого програмування та створення графічних інтерфейсів користувача. Автори підручника для 11-го класу того ж авторського колективу пропонують використовувати Python для реалізації складніших алгоритмів, таких як алгоритми сортування, пошуку та обробки графів.

Варто зазначити, що вибір Python як основної мови програмування для старшої школи є тенденцією, яка спостерігається не тільки в Україні. Багато

досліджень та практиків освіти в галузі інформатики відзначають переваги Python, такі як простота синтаксису, широкий спектр застосувань та активна спільнота розробників.

Підсумовуючи, можна зробити висновок, що усі аргументи на користь тієї чи іншої мови програмування є важливими, але останнє слово завжди залишається за вчителем. На думку авторів статті (Юрченко&Семеніхіна&Хворостіна&Удовиченко&Петренко, 2019), перспективним сьогодні у старшій школі вивчати популярні мови Java, Python та C++.

#### **1.4. Методика викладання теми «Рекурсія»**

Рекурсія вважається одним із фундаментальних інструментів програміста, але досить складним. Тому вона потребує ретельного вивчення та пояснення. Необхідно зазначити, що ознайомлення учнів навіть із найпростішими рекурсивними алгоритмами викликає ряд ускладнень при їх вивченні та засвоєнні.

Методичною помилкою є намагання пояснити рекурсивні процеси на прикладі факторіалу (або інших математичних задач), з поняттям якого більшість учнів до цього часу або зовсім не зустрічались, або не розуміють його математичної сутності й не вміють на практиці його застосовувати. Це означає, що пояснення поняття рекурсії в цьому випадку зводиться до пояснення поняття факторіалу, і таким чином основне поняття не формується. Тому слід особливу увагу звернути на добір прикладів для початкового пояснення поняття рекурсії. Вони повинні бути наочними, природно зрозумілими та чітко відображаючими суттєві ознаки рекурсії.

Другою помилкою, якої часто припускаються, є те, що учням не демонструють процес виконання рекурсивних алгоритмів, це призводить до нерозуміння учнями основних ознак рекурсії та правил її використання.

Перше ознайомлення з поняттям рекурсії можна почати з демонстрації деякого зображення відповідної структури, яке потім разом з учнями слід проаналізувати.

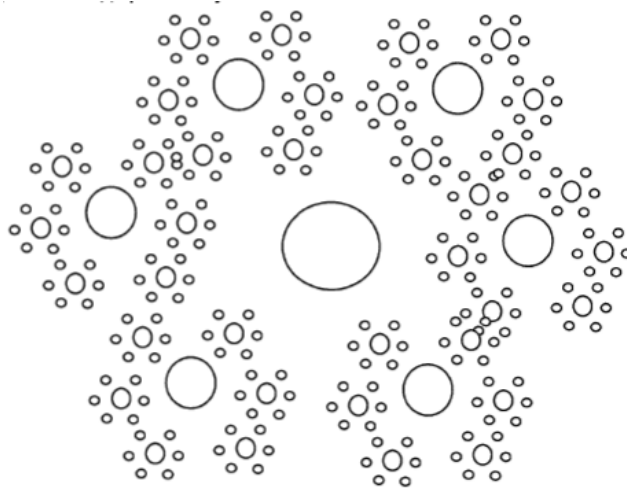


Рис. 1.5. Коло із шістьма супутніми колами

На рис. 1.5 подано коло із шістьма супутніми колами, у кожного з яких є свої шість супутніх кіл, у кожного з останніх — знову шість супутніх кіл і т.д., причому співвідношення між радіусами кола та супутніх кіл, а також між відстанями від центра кола до центрів супутніх кіл увесь час залишаються сталими.

Тому, якщо скласти алгоритм, за допомогою якого можна зобразити на екрані деяке коло із шістьма супутніми колами, а для малювання кожного із супутніх кіл також використовувати цей же алгоритм (зрозуміло, з іншими параметрами — координатами центрів кіл та радіусами), то можна одержати зображення, подане на рис. 1.5. Таким чином приходимо до ідеї про можливість звернення в деякому алгоритмі до нього ж самого.

Аналогічну задачу можна було б сформулювати так: задано деякий правильний шестикутник. Далі побудовано 6 правильних шестикутників з центрами у вершинах попереднього, сторонами, паралельними до сторін попереднього, причому відношення довжини сторін цих шести шестикутників до довжини сторони попереднього дорівнює  $a$ , ( $a$  може бути меншим за 1, більшим за 1, дорівнювати 1). З кожним із цих шести шестикутників знову будується шість шестикутників за тими ж правилами, що і раніше, і т.д. Так можна розглядати і трикутники, чотирикутники, п'ятикутники і т.ін., до кожного з них вміщувати коло з центром у центрі правильного багатокутника

і радіусом, довжина якого в  $k$  разів більша або менша (чи дорівнює) від радіуса вписаного чи описаного кола, і т.д.

Можна запропонувати також й інші рисунки, які будуються з використанням так званих рекурсивних процедур — це відомі килими Серпінського, сніжинки Коха тощо.

Далі доцільно повідомити учням, що вимога (вказівка) в алгоритмі про виконання цього ж алгоритму називається рекурсією (від латинського *recursion* — повернення), а алгоритми, в яких є вказівки про виконання їх же самих, називаються рекурсивними. Можна пригадати, що ситуація, яка нагадує рекурсію, зображена на фантику цукерки «Ану відніми!», на якому намальована собака, яка намагається відняти у дівчинки цукерки, а цукерка на фантику також називається «Ану відніми!» і т.д.

Крім того, як приклад рекурсії можна навести текст «історії» про попа та його собаку: «У попа був собака — він його любив. Він викрав шматок сала — той його убив. Убив, закопав, на камені написав: «У попа був собака ..... Таким чином, у пісні ця кількість повторень нескінченна, якщо її спеціально не обмежити деякою величиною, значення якої при повторенні пісні буде зменшуватися на 1, а при виконанні такої пісні кожного разу перевірятися, чи не дорівнює ця величина нулю.

**Приклад 1.** Можна також запропонувати учням проаналізувати вже готовий алгоритм побудови сніжинки, розробити на цьому кроці його деталі. Головне—демонстрація та розуміння рекурсивного процесу.

Нехай сніжинка створюється таким чином: із центра виростає 6 кристаликів — відрізків довжиною  $L$ , кути між сусідніми відрізками дорівнюють  $60^\circ$ ; з їх «вільних» кінців виростає по 5 відрізків; сусідні відрізки утворюють кути по  $60^\circ$ , довжини цих відрізків у  $K$  разів менші за  $L$ ; з їх «вільних» кінців аналогічно виростає по 5 нових відрізків, довжина яких ще в  $K$  разів менша, і так росте  $N$  «рівнів» сніжинки. Довжина кристалика на кожному рівні в  $K$  разів менша від довжини кристалика на попередньому рівні.

### Схема розв'язування задачі

Сніжинка складається із 6 гілок глибини  $N$  (які містять  $N$  рівнів), які повернуті одна відносно іншої на 60 градусів; кожна гілка глибини  $M$  ( $M=N, N-1, \dots, 3, 2$ ) складається з одного «стовбура» — відрізка довжиною  $L/K$  і п'яти гілок глибини  $M-1$ , які повернуті одна відносно іншої на 60 градусів. Такий опис сніжинки дозволяє розробити рекурсивний алгоритм її побудови.

Опис алгоритму навчальною алгоритмічною мовою

**алг** СНІЖИНКА (*ціл*  $N$ . *дійсн*  $L, K$ )

*арг*  $M, L, K$

*поч* *ціл*  $i$

*для*  $i$  *від* 1 до 6

*пц*

        ГІЛКА( $N, L, K$ )

        НАПРАВО(60)

    кц

кін

**алг** ГІЛКА (*ціл*  $N$ . *дійсн*  $L, K$ )

*арг*  $N, L, K$

*поч* *ціл*  $i$

    МАЛЮЙ

    ВПЕРЕД( $L$ )

*якщо*  $N > 1$

*то*                   {намалювати 5 гілок глибиною  $N-1$ }

        НАЛІВО(120)

*для*  $i$  *від* 1 до 5

*пц*

                ГІЛКА( $N-1, L/K, K$ )

                НАПРАВО(60)

        кц

        інакше НАЛІВО(180)

    все

    НЕ МАЛЮЙ

    НАЗАД( $L$ )

кін

Можна запропонувати учням намалювати таку сніжинку.

Таким чином, уводиться поняття **рекурсивного** алгоритму, при виконанні якого зустрічається вказівка про виконання його самого.

Необхідно з'ясувати з учнями, як виконується алгоритм побудови сніжинки. Важливо, щоб учні зрозуміли, що для того, щоб такий алгоритм

зробити скінченним, слід використати в ньому як аргумент деяку величину, яка при кожному новому виклику алгоритму буде збільшуватися або зменшуватися на 1, а до вказівок алгоритму слід включити вказівку про перевірку умови, за якою вказівки алгоритму повинні виконуватися лише тоді, коли ця умова істинна (вміст лічильника кількості звернень до алгоритму менший за наперед задану кількість  $N$ , або дорівнює нулю і т.ін.).

При проектуванні алгоритму зверху вниз для розв'язування деякої задачі може статися, що однією з визначених підзадач є вихідна задача, але з іншими даними, і за допомогою контролю за зміною даних задачу можна звести до випадку, який легко розв'язується. Доцільно скористатися такою ситуацією для одержання повного розв'язку задачі.

Можна дати учням описове тлумачення рекурсії: **зведення задачі до неї ж самої, але із зміненими вхідними даними, називається рекурсією.**

Рекурсія може бути *прямою (безпосередньою)* — якщо в алгоритмі є вказівки про виконання цього ж алгоритму, або *опосередкованою* — в алгоритмі немає прямих вказівок про виконання цього ж алгоритму, але є вказівки про виконання інших алгоритмів, в яких є вказівки про виконання даного алгоритму. В такий спосіб в алгоритмі можуть бути опосередковані посилання через інші алгоритми на цей самий алгоритм.

Після проведення етапу мотивації вивчення рекурсивних алгоритмів можна перейти до пояснення правил опису таких алгоритмів. При цьому пояснення доцільно будувати індуктивним способом, спираючись на добре відомі учням приклади.

**Приклад 2.** Практика свідчить, що методично виправданим є розгляд спочатку рекурсивного алгоритму обчислення багатократним додаванням добутку двох цілих чисел.

При цьому слід пояснити учням, що обчислення добутку двох цілих чисел  $A$  і  $B$  можна подати як обчислення суми, де як доданок використовується одне із заданих чисел, наприклад,  $A$ ; друге ж число (в даному прикладі число  $B$ ) буде визначати, скільки разів перше число слід додати до суми раніше

знайдених доданків, початкове значення якої дорівнює нулеві. Чергове значення знаходять шляхом збільшення біжучого значення суми на величину доданка  $S:=S+A$ . Такий процес продовжується до тих пір, поки не буде  $B$  разів додано до біжучого проміжного значення суми  $S$  доданок  $A$ .

Таким чином, для того, щоб знайти значення суми із  $B$  доданків  $A$ , необхідно додати доданок  $A$  до суми  $S$  із  $(B-1)$  доданків  $A$ , тобто використати рекурсивне звернення до аналогічних дій. Щоб знайти суму із  $(B-1)$  доданків  $A$ , треба додати доданок  $A$  до суми  $S$  із  $(B-2)$  доданків  $A$ . Такий зворотний процес продовжуватиметься доти, поки не дійдемо до суми  $S$  із 0 доданків  $A$ , значення якої дорівнює нулеві. Після цього здійснюється прямий хід (процес) — від суми 0 доданків переходять до суми із одного доданка, від суми із одного доданка—до суми із двох доданків і т.д., поки не буде отримана сума із  $B$  доданків.

Обов'язковим елементом рекурсивного описання процесу є умова *завершення рекурсії*. В цій умові повинно бути задано деяке фіксоване значення, якого обов'язково буде досягнуто в процесі рекурсивних обчислень, і контроль за яким дозволяє забезпечити своєчасне припинення процесу. Для розглядуваного прикладу це буде рівність  $k=B$ , де  $k$ — кількість кроків зворотного, а пізніше — прямого процесу.

Таким чином, важливо наголосити, що будь-яка *рекурсія містить три елементи*:

1. початкове значення проміжного результату (на нульовому кроці) перед початком прямого ходу;
2. спосіб одержання проміжного результату на  $i$ -тому кроці прямого ходу через проміжний результат, одержаний на  $(i-1)$ -му кроці;
3. умову завершення процесу.

**Приклад 3.** Розглянемо описаний вище алгоритм на конкретному прикладі. Нехай необхідно знайти добуток  $7 \times 5$ , який зводиться до суми

$$S=\underbrace{7+7+7+7+7}_{5 \text{ доданків}}$$

У свою чергу цю суму можна подати так:

$$\begin{aligned}
 S &= \underbrace{7+7+7+7+7}_{5 \text{ доданків}} = 7 + \underbrace{(7+7+7+7)}_{4 \text{ доданки}} = 7 + (7 + \underbrace{(7+7+7)}_{3 \text{ доданки}}) = 7 + (7 + (7 + \underbrace{(7+7)}_{2 \text{ доданки}})) = 7 + (7 + (7 + (7 + \underbrace{(7)}_{1 \text{ доданок}}))) = \\
 &= 7 + (7 + (7 + (7 + (7 + \underbrace{(0)}_{0 \text{ доданків}}))))
 \end{aligned}$$

Таким чином, у даному прикладі:  $S(5)=7+S(5-1)$ , а для довільних  $A$  і  $B$  формула знаходження чергового елемента буде виглядати так:  $S(B)=A+S(B-1)$ . Отже визначено спосіб одержання проміжного результату на кожному наступному кроці через проміжний результат на попередньому кроці. Процес **рекурсивний**. Це означає, що крім способу одержання наступного проміжного результату через попередній необхідно при опису такого процесу ввести для розгляду вказівку розгалуження, за якою буде визначатися момент закінчення (кількість) дій, що повторюються. При цьому природно використати значення величини  $B$ .

Тоді рекурсивний алгоритм-функцію можна записати навчальною алгоритмічною мовою так:

```

алг ціл сума(ціл A, B)
поч
  якщо B=0
    то знач:=0
    інакше знач:=сума(A, B-1 )+A
все
кін

```

При виконанні рекурсивних алгоритмів для кожного модуля будується своя таблиця виконання. В рекурсивних алгоритмах окрема таблиця виконання будується при кожному черговому виконанні алгоритму для самого себе. При виході з алгоритму здійснюється повернення до попередньої таблиці виконання (таблиці алгоритму, який викликаний даним), а біжуча таблиця вилучається.

Після створення такого алгоритму необхідно разом з учнями побудувати таблиці виконання та виконати описаний алгоритм.

На рис. 1.6 наведено таблиці виконання рекурсивного алгоритму, де суцільні лінії вказують напрями прямого ходу, пунктирні — напрями



зворотного ходу.

Крім того, для унаочнення схеми виклику рекурсивної функції можна скористатися схемами звернення до алгоритму з деякого блоку (рис. 1.7).

Використовуючи таку схему виклику алгоритму за допомогою спеціальної вказівки попередній рекурсивний алгоритм можна наочно зобразити так, як показано на рис. 1.8.

Зворотний хід одержується автоматично за рахунок того, що кожний алгоритм, який викликається, після свого завершення повертає управління на вказівку, яка йде за вказівкою виконання алгоритму. Таким чином, дійшовши до алгоритму, який вже не буде викликати даний і видасть певний результат, із цього управління повернеться до попереднього на вказівку, яка розташована за вказівкою «виконати алгоритм» (у прикладі 5 на «все»). Після завершення виконання цього попереднього алгоритму управління буде передано на вказівку, яка йде наступною за вказівкою виклику попереднього алгоритму і т. ін., до тих пір, поки управління не повернеться на вказівку, яка слідує після самої першої вказівки про виконання алгоритму.

Рекурсивне (як і нерекурсивне) багатократне виконання алгоритму (вкладання впорядкованого набору алгоритмів послідовно один в інший) можна графічно подати так, як на рис. 1.9.

Набір алгоритмів  $S(A,B)$ ,  $S(A,B-1)$ ,  $S(A,B-2)$ ,  $S(A,2)$ ,  $S(A,1)$ ,  $S(A,0)$  є впорядкованим (за спаданням значення  $B-K$ ,  $K=0,1,2 \dots B$ ), кожна наступна вказівка вложена в попередню, тобто в кожному з алгоритмів  $S(A,B-K)$  є вказівка про виконання алгоритму  $S(A,B-K-1)$ , причому після виконання вказівки про виконання алгоритму  $S(A,B-K-1)$  буде виконуватися вказівка алгоритму  $S(A,B-K)$ , що слідує одразу за вказівкою про виконання алгоритму  $S(A, B-K-1)$ .

сума (7,5)				
Крок	Аргументи		Результати	Перевірка умови
	A	B	знач	
	7	5	В	
1.				5=0 (-)
2.			сума (7,4)+7	
3.			35	

сума (7,4)				
Крок	Аргументи		Результати	Перевірка умови
	A	B-1	знач	
	7	4		
1.				4=0 (-)
2.			сума (7,3)+7	
3.			28	

сума (7,3)				
Крок	Аргументи		Результати	Перевірка умови
	A	B-2	знач	
	7	3		
1.				3=0 (-)
2.			сума (7,2)+7	
3.			21	

сума (7,2)				
Крок	Аргументи		Результати	Перевірка умови
	A	B-3	знач	
	7	2		
1.				2=0 (-)
2.			сума (7,1)+7	
3.			14	

сума (7,1)				
Крок	Аргументи		Результати	Перевірка умови
	A	B-4	знач	
	7	1		
1.				1=0 (-)
2.			сума (7,0)+7	
3.			7	

сума (7,0)				
Крок	Аргументи		Результати	Перевірка умови
	A	B-5	знач	
	7*	0		
1.				0=0 (+)
2.			0	

Рис 1.6. Таблиці виконання рекурсивного алгоритму



Рис. 1.7. Блок схема алгоритму

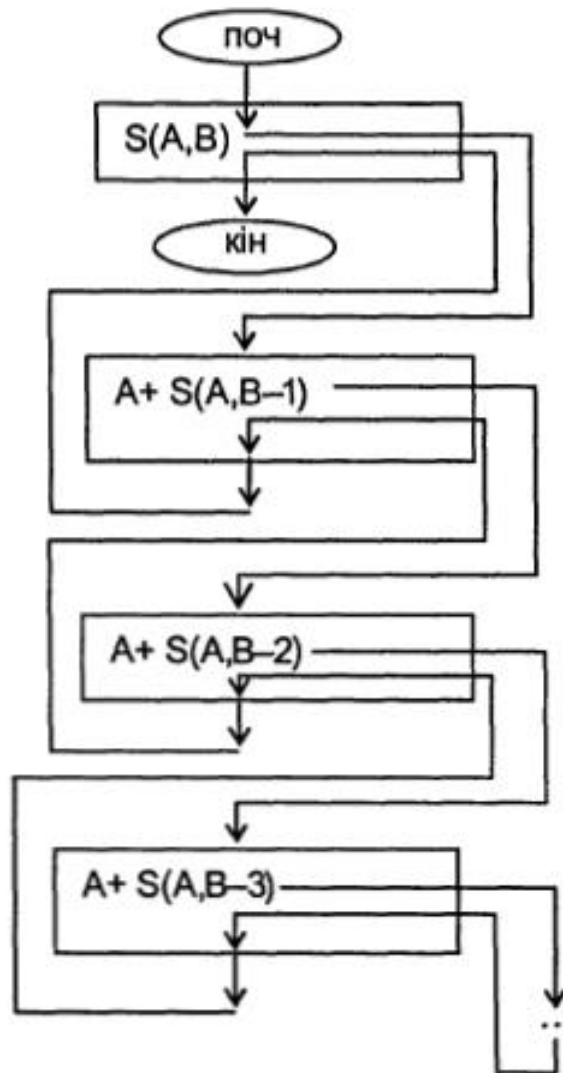


Рис. 1.8. Блок схема алгоритму

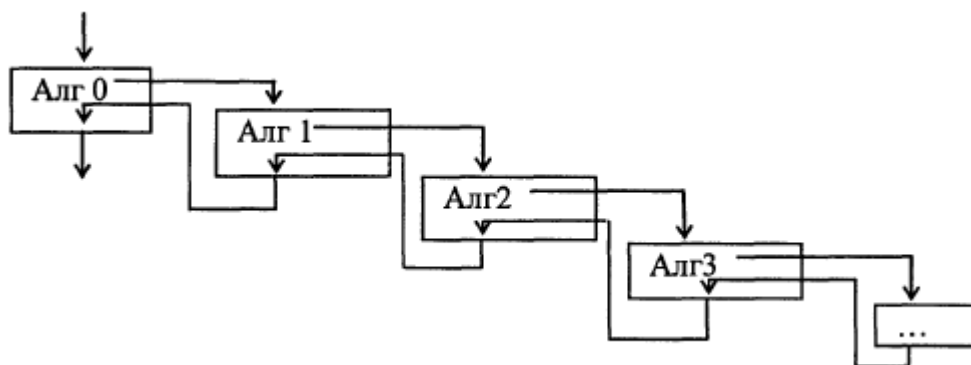


Рис. 1.9. Рекурсія

**Приклад 4.** Далі можна за аналогією проаналізувати рекурсивний алгоритм обчислення степеня числа  $a^b$ , де  $b$  — ціле число.

Міркування доцільно побудувати так: алгоритм обчислення степеня  $a^b$

можна подати у вигляді добутку на кожному кроці числа  $a$  на добуток, отриманий на  $(k-1)$ - му кроці, повторивши таке множення  $b$  разів, взявши як початкове значення для добутків число 1:  $(\dots(((1*a)*a)*a)*\dots*a) \} b$  разів.

Спочатку значення такого добутку дорівнює 1, а наступне його значення визначається як добуток попереднього значення добутку на задане число  $a$ . Кількість повторень такого процесу дорівнює  $b$ . Очевидно, що значення степеню  $a^b$  знаходиться таким чином:

$$\begin{aligned} a^b &= a * a^{b-1} = a * (a * a^{b-2}) = a * (a * (a * a^{b-3})) = a * (a * (a * \dots * (a * a^2))) = a * (a * (a * \dots * a * (a^1))) = \\ &= a * a * a * a * \dots * a^0 = \underbrace{a * a * a * \dots * a * 1}_{b \text{ разів}} \end{aligned}$$

Таким чином, цей алгоритм можна описати рекурсивно. Умовою виходу із рекурсивного процесу буде рівність  $b-k=0$ ,  $k=0,1,2, \dots, b$ , а спосіб одержання одного значення через інше можна виразити за допомогою формули  $\text{ступінь}(a,b) := \text{ступінь}(a,b-1) * a$ .

Лишилось розглянути випадок, коли показник степеня — ціле від'ємне число. В цьому випадку значення степеня знаходиться як обернене значення степеня  $a^{|b|}$ , тобто  $1 / a^{|b|}$ .

Опис алгоритму навчальною алгоритмічною мовою

**алг ціл степені (ціл основа, показник)**

**поч**

**якщо** показник  $< 0$

**то знач:**  $= 1 / \text{ступінь}(\text{основа}, \text{abs}(\text{показник}))$

**інакше**

**якщо** показник  $= 0$

**то знач:**  $= 1$

**інакше знач:**  $= \text{ступінь}(\text{основа}, \text{показник}-1) * \text{основа}$

**все**

**все**

**кін**

Виконання такого алгоритму за допомогою відповідних таблиць доцільно запропонувати учням як самостійну роботу.

Далі можна разом з учнями розв'язати такі задачі:

**Приклад 5.** Дано натуральне число. Скласти алгоритм знаходження факторіалу цього числа, використовуючи рекурсію.

Схема розв'язування задачі Будемо використовувати рекурентне визначення факторіалу:  $N! = (N - 1)! * N$ . Умовою завершення рекурсії буде рівність  $1! = 1$ . У випадку, коли  $N$  дорівнює одиниці, функція факторіал одержує значення 1. Якщо ж  $N$  більше за одиницю, значення факторіал визначається рекурсивно: знаходиться добуток значення  $N$  і факторіалу попереднього до  $N$  числа, тобто факторіалу числа  $(N-1)$ .

Опис алгоритму навчальною алгоритмічною мовою

```

алг ціл факторіал (ціл N)
поч
    якщо N=1
        то знач:=1
    інакше знач:=N*факторіал(N-1)
все
кін

```

На рис. 1.10 наведено таблиці виконання рекурсивного алгоритму факторіал, який викликається із такого головного алгоритму:

```

алг Задача 1 (нат F)
    рез F
    поч
         $P := \text{факторіал}(4)$ 
    кін

```

Як видно з таблиць, при виконанні рекурсивного алгоритму, спочатку відбувається своєрідне «заглиблення» на все глибші рівні рекурсивного алгоритму, а потім вихід (піднімання) аж до повернення до зовнішньої програми, в якій є вказівка про виконання алгоритму.

Заповнення таблиць виконання рекурсивного алгоритму: кожне нове виконання збільшує рівень рекурсивного заглиблення, а повернення результату і вихід із алгоритму—зменшує «глибину» рекурсивного заглиблення. Стрілками позначено передавання аргументів і повернення значень результатів між алгоритмами.

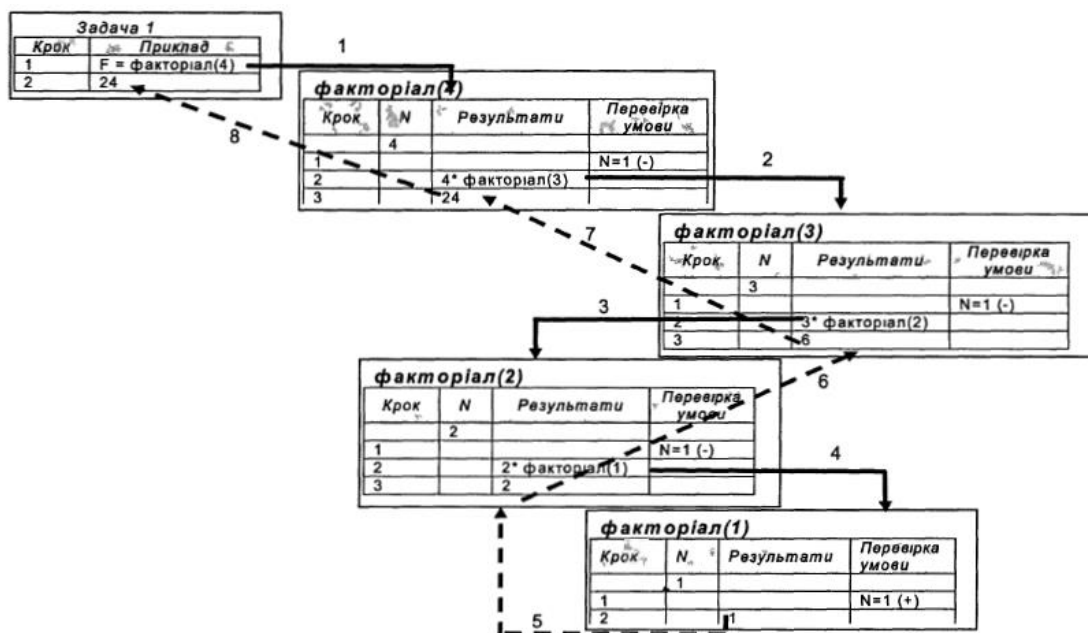


Рис. 1.10. Таблиці виконання рекурсивного алгоритму факторіал

**Приклад 6.** Скласти алгоритм для знаходження членів арифметичної і геометричної прогресій та їх суми, використовуючи при цьому рекурсивні алгоритми.

Схема розв'язування задачі

Відомо, що при заданому значенні першого члена  $a_1$  і різниці  $d$  членів арифметичної прогресії значення наступних членів знаходяться за формулами:

$$a_2 = a_1 + d$$

$$a_3 = a_2 + d$$

.....

$$a_n = a_{n-1} + d$$

Формула знаходження членів арифметичної прогресії є рекурентною, тобто дозволяє знаходити наступне значення через попереднє, і вказує на можливість описання рекурсивного алгоритму. Залишається визначити умову закінчення рекурсивного процесу.

Умовою закінчення рекурсії є рівність  $N-K=1$  для номера  $N$  членів прогресії. Очевидно, що якщо номер шуканого члена збігається з одиницею, то значення рекурсивної функції збігається із значенням першого члена прогресії. Значення будь-якого  $N$ -го члена арифметичної прогресії знаходиться як сума різниці  $d$  і значення  $(N-1)$ -го члена, який в свою чергу знаходиться як

сума різниці  $d$  і значення  $(N-2)$ -го члена і т.д.

Суму членів арифметичної прогресії також можна знайти на основі рекурсивного процесу. Чергове її значення для  $N$  членів подається через суму  $N-1$  членів, яка в свою чергу подається за допомогою вже описаного рекурсивного алгоритму функції знаходження суми.

Аналогічно на основі рекурсивного процесу описується алгоритм знаходження членів геометричної прогресії та їх суми.

Опис алгоритму навчальною алгоритмічною мовою

```
алг дійсн АРИФМ (дійсн  $a_1$ ,  $d$ , ціл  $n$ )
поч
    ЯКЩО  $n=1$ 
        то знач:= $a_1$ 
        інакше знач:=АРИФМ( $a_1, d, n-1$ )+ $d$ 
    все
алг дійсн СУМА_АРИФМ (дійсн  $a_1$ ,  $d$ , ціл  $n$ )
поч
    якщо  $n=1$ 
        то знач:= $a_1$ 
        інакше
            знач:=СУМА_АРИФМ( $a_1, d, n-1$ )+АРИФМ( $a_1, d, n$ )
    все
кін
```

Аналогічно і черговий член і суму членів геометричної прогресії можна знайти на основі рекурсивного процесу. Учня можна запропонувати самотійно скласти такі алгоритми та продемонструвати їх правильність за допомогою виконання для конкретних вхідних даних.

Опис алгоритму навчальною алгоритмічною мовою

```
алг дійсн ГЕОМ (дійсн  $b_1, q$ , ціл  $n$ )
поч
    якщо  $n=1$ 
        то знач:= $b_1$ 
        інакше знач:=ГЕОМ( $b_1, q, n-1$ )* $q$ 
    все
кін
алг дійсн СУМА_ГЕОМ (дійсн  $b_1, q$ , ціл  $n$ )
поч
    якщо  $n=1$ 
        то знач:= $b_1$ 
```

інакше

знач:=СУМА\_ГЕОМ(b1,q,n-1)+ГЕОМ(b1,q,n)

все

кін

**Приклад 7.** Скласти алгоритм для визначення мінімального вмісту клітинок лінійної таблиці, використовуючи рекурсивну функцію для знаходження мінімального вмісту серед решти вмістів клітинок таблиці.

### Схема розв'язування задачі

Для побудови алгоритму розглянемо конкретну таблицю A[1:5]:

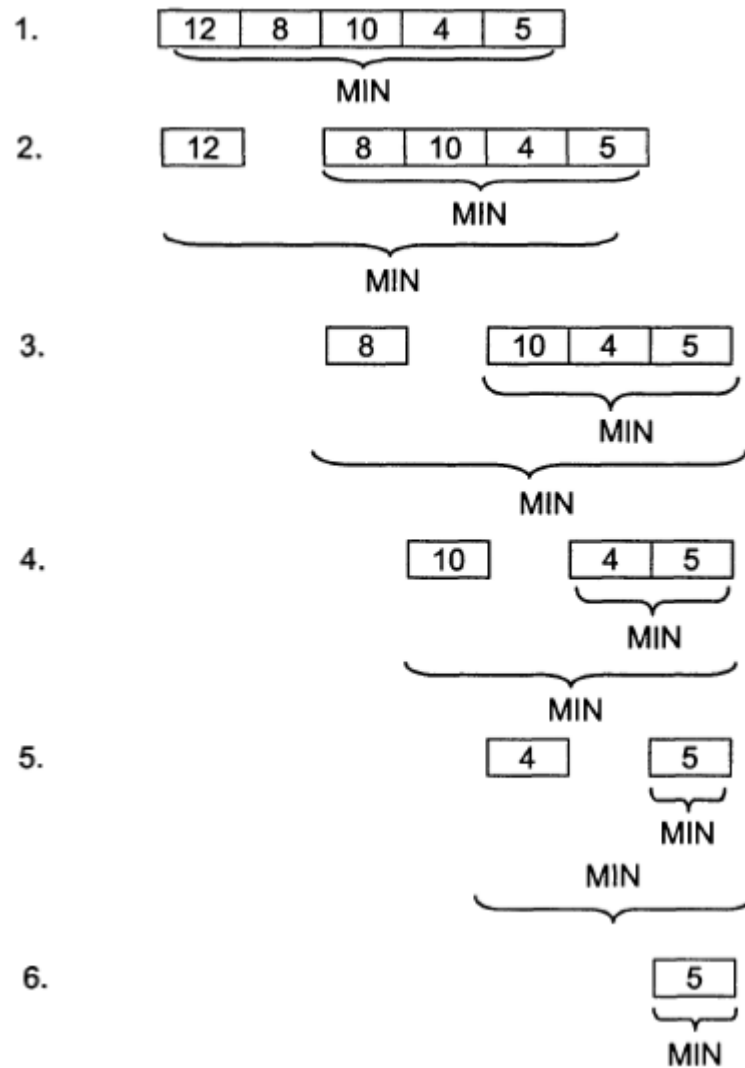
1	2	3	4	5
12	8	10	4	5

Мінімальний елемент у цій таблиці будемо знаходити як мінімальний елемент із двох: першого елемента A[1] і мінімального елемента таблиці A[2:5]. У свою чергу для знаходження мінімального елемента в таблиці A[2:5] будемо знаходити мінімальний елемент із двох: першого елемента цієї таблиці A[2] і мінімального елемента таблиці A[3:5]. Аналогічно для знаходження мінімального елемента таблиці A[3:5] скористаємося функцією знаходження мінімального з двох: першого елемента цієї таблиці A[3] і мінімального елемента таблиці A[4:5], для цієї таблиці знайдемо мінімальний елемент, як мінімальний з двох A[4] і мінімального в таблиці A[5:5]. Як бачимо, процес знаходження мінімального вмісту клітинок у конкретній таблиці закінчився. Лишилося визначити умову закінчення рекурсивного процесу та спосіб одержання наступного значення проміжного результату через попереднє.

У такій умові може бути використаний номер розглядуваної клітинки таблиці: якщо номер розглядуваної клітинки таблиці збігається з номером останньої клітинки в таблиці, то за мінімальне слід прийняти значення розглядуваного, тобто вмісту останньої клітинки таблиці.

За цим алгоритмом із вмістом клітинок таблиці виконуються наступні дії:





### Опис алгоритму навчальною алгоритмічною мовою

*алг* НАЙМЕНТАБ(ціл N, *дійсн таб* A[1:N], *дійсн* мінтаб)

арг N,A

рез мінтаб

*поч*

мінтаб:=MIN(1)

*кін*

У цьому алгоритмі міститься вказівка про надання значення, яка передбачає виконання рекурсивного алгоритму для знаходження мінімального елемента таблиці.

*алг дійсн MIN*(ціл K)

*поч* ціл N, *дійсн таб* A[1 :N]

*якщо* K=N

то знач:=A[N]

інакше

*якщо* A[K]<MIN(K+1)

```

        то знач::=A[K]
        інакше знач:=MIN(K+1)
    все
все
кін

```

При зверненні до рекурсивної функції спочатку записується умова закінчення рекурсії: якщо номер клітинки, з якої вибирається аргумент функції, збігається з номером останньої клітинки таблиці, то за значення функції приймається значення вмісту останньої клітинки таблиці. Якщо ж номер клітинки, з якої вибирається аргумент рекурсивної функції, не збігається з номером останньої клітинки, то порівнюється значення біжучого аргументу із значенням мінімального в таблиці, номер першого елемента якої на 1 більший значення номера біжучого елемента. Якщо біжучий елемент менший, то значення функції набуває значення цього елемента, інакше знову йде звернення до функції, яка знаходить мінімальний елемент серед наступної частини таблиці, номер першого елемента якої на 1 більше вже розглянутого.

Якщо цей самий алгоритм описати за допомогою рекурсивної процедури знаходження мінімального елемента в згаданій глобальній таблиці А, то він буде мати вигляд:

```

алг Min (ціл n, дійсн X)
    арг n
    рез X
    поч
        якщо n=1
            то x:=A[1]
            інакше Min (n-1 ,x)
                якщо A[n]<x
                    то x:=A[n]
        все
    все
кін

```

**Приклад 8. Скласти рекурсивний алгоритм для знаходження найбільшого спільного дільника двох чисел.**

Для побудови рекурсивного алгоритму знаходження найбільшого спільного дільника двох чисел розглянемо приклад. Нехай задано два числа  $m=15$  і  $n=48$ . Якщо друге число  $n$  більше першого  $m$ , то поміняємо їх місцями:

$m=48$ ,  $n=15$ . Знайдемо остачу від цілочисельного ділення числа  $m$  на  $n$  ( $48 \bmod 15=3$ ). Тепер будемо розглядати числа  $m=15$  (менше з двох попередніх, що розглядалися) і  $n=3$  (остачу від ділення). Для таких чисел також знайдемо остачу від ділення ( $15 \bmod 3=0$ ). Тепер знову розглянемо числа  $m=3$  (менше з двох попередніх) і  $n=0$  (остачу від ділення). І коли друге число стало дорівнювати нулю, то це означає, що процес треба закінчити, тобто знайдено найбільший спільний дільник, і він дорівнює числу  $m=3$ .

Як видно з наведеного прикладу, для розв'язування задачі багатократно використовувались одні й ті самі дії: порівнювались два числа і більше з них ставилось на перше місце, потім перевірялось друге число, і якщо воно дорівнювало нулю, то процес закінчувався і за найбільший спільний дільник вибиралось перше число. Якщо друге число відмінне від нуля, то процес продовжувався, і тепер аналогічні дії виконували з другим числом і остачею від ділення першого числа на друге.

Опис алгоритму навчальною алгоритмічною мовою

```
алг ціл НСД (ціл m, n)
поч ціл h
    якщо n > m
        то h:=НСД( n, m )
        інакше
            якщо n=0
                то h:=m
                інакше h:=НСД (n, ціл(m,n))
    все
все
знач = h
кін
```

### Висновки до розділу

В даному розділі проаналізовані мови програмування які можливо використовувати в закладах загальної середньої освіти. Проведено аналіз програм та підручників з теми «Алгоритми та програми» в закладах загальної середньої освіти. Розглянуто методику викладання теми «Рекурсія».

## РОЗДІЛ II

### РЕКУРСІЯ ТА РЕКУРСИВНІ АЛГОРИТМИ

#### 2.1. Рекурсивні функції та алгоритми

##### 2.1.1. Рекурсивні функції

Концепція рекурсії, що передбачає визначення об'єкта через посилення на самого себе, є однією з фундаментальних ідей в програмуванні. Її коріння сягають середини XX століття, коли в контексті процедурного програмування з'явилася практика взаємних викликів функцій. Незважаючи на появу нових парадигм програмування, рекурсія залишається актуальним інструментом розробки алгоритмів, хоча і розглядається як альтернатива ітеративним методам.

По суті один і той же метод, стосовно до різних областей носить різні назви – це індукція, рекурсія і рекурентні співвідношення – відмінності стосуються особливостей використання.

**Під індукцією** розуміється метод доведення тверджень, який будується на базі індукції при  $n = 0, 1$ , потім твердження покладається правильним при  $n = n_b$  і проводиться доказ для  $n + 1$ .

**Рекурсія** – це визначення об'єкта через звернення до самого себе.

**Рекурсивний алгоритм** – це алгоритм, в описі якого прямо або побічно міститься звернення до самого себе. У техніці процедурного програмування дане поняття поширюється на функцію, яка реалізує рішення окремого блоку завдання за допомогою виклику зі свого тіла інших функцій, в тому числі і себе самої. Якщо при цьому на черговому етапі роботи функція організовує звернення до самої себе, то така функція є рекурсивною.

Рекурсія - це потужний інструмент програмування, який дозволяє визначити функцію через її власний виклик. Існує два основних типи рекурсії: пряма і непряма.

Пряма рекурсія передбачає, що функція безпосередньо викликає саму себе з іншими аргументами. Класичним прикладом прямої рекурсії є обчислення факторіалу числа.

Непряма рекурсія має місце тоді, коли функція викликає іншу функцію, яка, в свою чергу, може викликати вихідну функцію. Таким чином, виклики функцій

утворюють цикл. Прикладом непрямої рекурсії може бути взаємний виклик двох функцій для обходу бінарного дерева.

Розробка рекурсивних алгоритмів передбачає чітке розуміння та виконання трьох основних етапів: параметризації, виділення бази та декомпозиції.

Параметризація полягає у визначенні сукупності параметрів, які повністю описують вхідні дані задачі. Цей етап є критичним, оскільки від правильного вибору параметрів залежить ефективність та коректність подальших розрахунків.

Виділення бази рекурсії передбачає визначення тривіальних випадків задачі, для яких рішення відоме безпосередньо. База рекурсії слугує точкою зупинки рекурсивних викликів і гарантує завершення алгоритму.

Декомпозиція полягає у розбитті задачі на підзадачі, які є меншими копіями вихідної задачі. При цьому, кожна підзадача повинна бути вирішена за допомогою того ж самого алгоритму, але з іншими значеннями параметрів.

Існує декілька категорій завдань, що допускають рекурсивні визначення. Одна з категорій – математичні формули, визначення яких рекурсивне за своєю суттю.

Основним завданням дослідження рекурсивних функцій є отримання  $F(n)$  в явній або як ще кажуть «замкнутій» формі, тобто у вигляді аналітично заданої функції.

Класичний приклад функції, визначення якої може задаватися в рекурсивній формі,  $F(n) = n!$

$$F(n) = n * (n-1) * \dots * 1, 0! = 1$$

Рекурсивне визначення цієї функції має вид:

$$F(n) = \begin{cases} 1, & n = 0 \\ n * F(n-1), & n > 0 \end{cases}$$

де  $F(n-1) = (n-1)!$

Другий приклад – це опис синтаксису конструкцій мов програмування за

допомогою Бэкуса Наура форми (БНФ).

### **Приклад.**

Визначення ідентифікатора в мові Паскаль: зараз C++

$\langle \text{літера} \rangle ::= a|b|\dots|z,$

$\langle \text{цифра} \rangle ::= 0|1|\dots|9;$

$\langle \text{ідентифікатор} \rangle ::= \langle \text{літера} \rangle | \langle \text{ідентифікатор} \rangle \langle \text{літера} \rangle | \langle \text{ідентифікатор} \rangle \langle \text{цифра} \rangle.$

### **2.1.2. Рекурсивні процедури і функції**

Рекурсія є потужним інструментом програмування, який знаходить широке застосування в різних областях. Вона особливо корисна для задач, де:

- Математична модель має рекурсивну форму.
- Структури даних мають рекурсивну структуру.
- Процес розв'язання задачі можна розбити на підзадачі того ж типу.

Рекурсивні алгоритми дозволяють отримати компактні та елегантні рішення для багатьох складних задач.

Підпрограма називається рекурсивною, якщо в її визначенні присутній прямо або побічно виклик самої обумовленої підпрограми.

При прямій рекурсії підпрограма створює циклічний потік виконання, де кожен наступний виклик повторює логіку попереднього, але з іншими вхідними даними.

Непряма рекурсія створює взаємозалежність між кількома підпрограмами, де кожна з них може викликати іншу, утворюючи цикл викликів.

### **Реалізація рекурсивних підпрограм**

**Стек** – це фундаментальна структура даних, яка працює за принципом LIFO (Last In, First Out). Його основна функція полягає в збереженні інформації про поточний стан виконання програми, особливо в контексті рекурсивних викликів. Кожен новий виклик рекурсивної функції призводить до створення нового запису у стеку, який містить локальні змінні, параметри

та адресу повернення. Таким чином, стек забезпечує механізм для відстеження послідовності викликів та повернення до попереднього стану після завершення кожного виклику.

У загальному випадку при виклику процедурою А процедури В відбувається наступне:

1. В вершину стека поміщається фрагмент потрібного розміру. До нього входять такі, дані:

- (а) показники фактичних параметрів виклику процедури В;
- (б) порожні комірки для локальних змінних, визначених у процедурі В;
- (в) адреса повернення (АВ), тобто адреса команди в процедурі А, яку слід виконати після того, як процедура В закінчить свою роботу.

Якщо В – функція, то у фрагмент стеку для В поміщається показчик осередку у фрагменті стека для А, в якій належить помістити значення цієї функції (адреса значення).

2. Управління передається першому оператору процедури В.

3. При завершенні роботи процедури В управління передається процедурі А за допомогою наступної послідовності кроків:

- (а) адреса повернення витягується з вершини стеку;
- (б) якщо В – функція, то її значення запам'ятовується в комірці, на яку вказує показчик адреси значення;
- (в) фрагмент стека процедури В витягується з стека, в вершину ставиться фрагмент процедури А;
- (г) виконання процедури А поновлюється з команди, зазначеної в адресі повернення.

Рекурсивний алгоритм складається з двох основних частин: базового випадку та рекурсивного випадку. Базовий випадок визначає тривіальне рішення для найпростіших входів і слугує точкою зупинки для рекурсивних викликів. Рекурсивний випадок виражає проблему в термінах самої себе, але з меншими за розміром даними, і повторюється доти, доки не буде досягнуто базового випадку.

## Рекурсія і ітерація

Рекурсивний підхід не завжди є оптимальним вибором для розв'язання задачі. Ітеративні алгоритми, в багатьох випадках, можуть забезпечити більш просте та ефективне рішення.

**Приклад.** Необхідно скласти визначення функції для обчислення значень деяких поліномів, визначення яких має наступний вигляд:

$$S_n(x) = \begin{cases} 0, & \text{если } n = 0, \\ 2x, & \text{если } n = 1, \\ \frac{2n}{n-1} S_{n-1}(x) + \frac{n-1}{2n} S_{n-2}(x), & \text{если } n > 1 \end{cases}$$

Розглянемо ряд прикладів:

б) Приклади рекурсивних функцій

1.

$$\begin{cases} f(0) = 1 \\ f(n) = n * f(n-1) \end{cases}$$

Зрозуміло, що  $f(n) = n!$

2.

$$\begin{cases} f(0) = 1 \\ f(n) = n * f(n-1) \end{cases}$$

Послідовна підстановка дає –  $f(n) = 1 * 2 * 3 * \dots * n = n!$

Для повноти відомостей наведемо формулу Стірлінга для наближеного обчислення факторіала для великих  $n$ :  $n! \approx (2\pi n)^{1/2} * (n^n) / (e^n)$

3.

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n) = f(n-1) + f(n-2), & n \geq 2 \end{cases}$$



Ця рекурсивна функція визначає числа Фібоначчі: 1 1 2 3 5 8 13, які досить часто виникають при аналізі різних завдань, у тому числі і при аналізі алгоритмів. Відзначимо, що асимптотично  $f(n) \gg [1,618 n]$  [8]

4.

$$\begin{cases} f(0)=1 \\ f(n)=f(n-1)+f(n-2)+\dots+1=\sum f(i)+1 \end{cases}$$

Для отримання функції в явному вигляді розглянемо її послідовні значення:  $f(0)=1$ ,  $f(1)=2$ ,  $f(2)=4$ ,  $f(3)=8$ , що дозволяє припустити, що  $f(n)=2^n$ , точний доказ виконується по індукції.

5.

$$\begin{cases} f(0)=1 \\ f(n)=2*f(n-1) \end{cases}$$

Ми маємо справу з прикладом того, що одна і та ж функція може мати різні рекурсивні визначення –  $f(n) = 2^n$ , як і в прикладі 4, що перевіряється елементарною підстановкою.

6.

$$\begin{cases} f(0)=1 \\ f(1)=2 \\ f(n)=f(n-1)*f(n-2) \end{cases}$$

У цьому випадку ми можемо отримати рішення в замкнутій формі, зіставивши що значенням функції відповідають ступені двійки:

Позначаючи через  $F_n$  –  $n$ -е число Фібоначчі, маємо:  $f(n)=2^{F_n}$

### **2.1.3. Аналіз трудомісткості рекурсивних алгоритмів методом підрахунку вершин дерева рекурсії**

Рекурсивні алгоритми, незважаючи на свою елегантність та інтуїтивну зрозумілість, часто характеризуються підвищеною обчислювальною складністю. Це пов'язано з кількома факторами. По-перше, глибока рекурсія може призвести до значного збільшення розміру стеку викликів, що може

обмежити застосування таких алгоритмів для великих вхідних даних. По-друге, додаткові операції, пов'язані з управлінням стеком (створення, знищення, пошук елементів), також впливають на загальну часову складність алгоритму. Крім того, кількість переданих параметрів у рекурсивних викликах може суттєво впливати на ефективність алгоритму.

Одним із методів оцінки ефективності рекурсивних алгоритмів є аналіз рекурсивного дерева. Цей метод дозволяє візуалізувати структуру рекурсивних викликів та оцінити їх кількість.

Для оцінки трудомісткості рекурсивних алгоритмів будується **повне дерево рекурсії**. Воно являє собою граф, вершинами якого є набори фактичних параметрів при всіх викликах функції, починаючи з першого звернення до неї, а ребрами – пари таких наборів, відповідних взаємним викликам. При цьому вершини дерева рекурсії відповідають фактичним викликам рекурсивних функцій. Слід зауважити, що одні й ті ж набори параметрів можуть відповідати різним вершинам дерева. Корінь повного дерева рекурсивних викликів – це вершина повного дерева рекурсії, відповідна початковим зверненням до функції.

Важливою характеристикою рекурсивного алгоритму є **глибина рекурсивних викликів** – найбільше одночасне кількість рекурсивних звернень функції, що визначає максимальну кількість шарів рекурсивного стека, в якому здійснюється зберігання відкладених обчислень. Кількість елементів повних рекурсивних звернень завжди не менше глибини рекурсивних викликів. При розробці рекурсивних програм необхідно враховувати, що глибина рекурсивних викликів не повинна перевершувати максимального розміру стека використовуваного обчислювального середовища.

При цьому **обсяг рекурсії** – це одна з характеристик складності рекурсивних обчислень для конкретного набору параметрів, що представляє собою кількість вершин повного рекурсивного дерева без одиниці.

#### 2.1.4. Рекурсивна реалізація алгоритмів

Більшість сучасних мов високого рівня підтримують механізм рекурсивного виклику, коли функція, як елемент структури мови програмування, що повертає обчислене значення по своєму імені, може викликати сама себе з іншим аргументом. Ця можливість дозволяє безпосередньо реалізовувати рекурсивне обчислення певних функцій.

Рекурсивний алгоритм може бути реалізований ітераційною послідовністю дій.

Розглянемо приклад рекурсивної функції, що обчислює факторіал:  $F(n)$ .

*If  $n=0$  or  $n=1$*  (перевірка можливості прямого обчислення)

*Then  $F \leftarrow 1$*

*Else  $F \leftarrow n * F(n-1)$* ; ( рекурсивний виклик функції)

*Return ( $F$ );*

*End*

Аналіз трудомісткості рекурсивних реалізацій алгоритмів, очевидно, пов'язаний як з кількістю операцій, виконуваних при одному виклику функції, так і з кількістю таких викликів. Графічне представлення породжуваної даними алгоритму ланцюжка рекурсивних викликів називається деревом рекурсивних викликів. Більш детальний розгляд призводить до необхідності врахування витрат як на організацію виклику функції та передачі параметрів, так і на повернення обчислених значень і передачу управління в точку виклику. Можна помітити, що деяка гілка дерева рекурсивних викликів обривається при досягненні такого значення переданого параметра, при якому функція може бути обчислена безпосередньо. Таким чином, рекурсія еквівалентна конструкції циклу, в якому кожен прохід є виконанням рекурсивної функції з заданим параметром.

Розглянемо приклад для функції обчислення факторіалу (рис 2.1.)

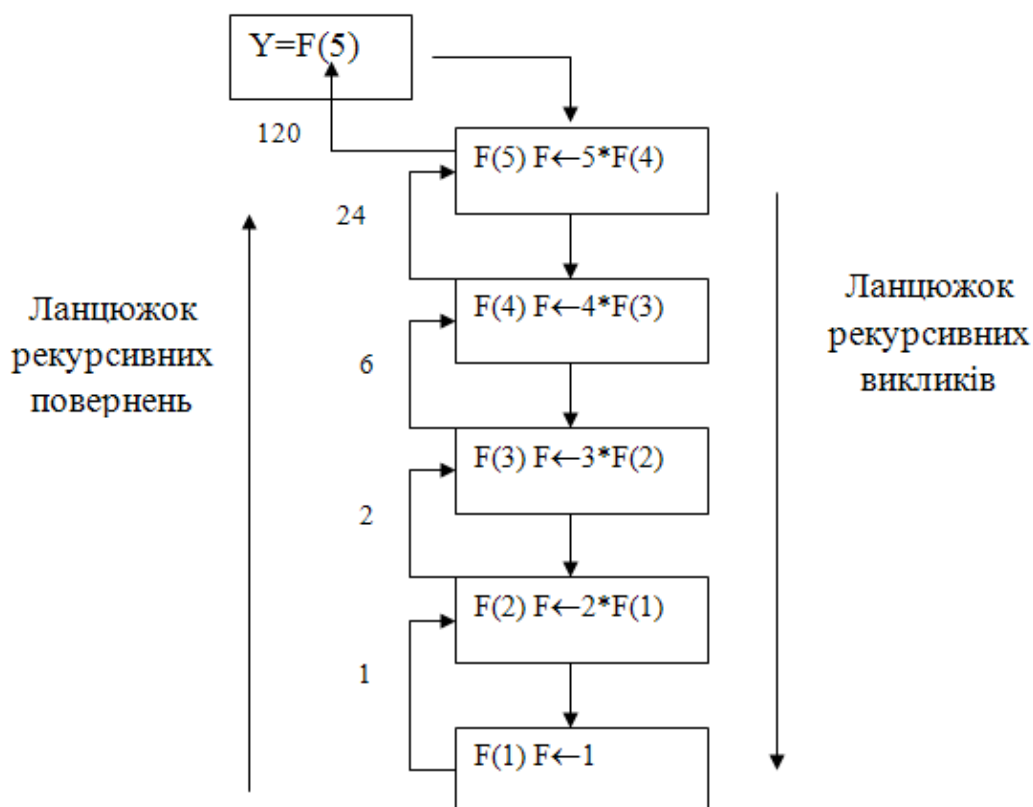


Рис. 2.1. Дерево рекурсії при обчисленні факторіалу –  $F(5)$

Дерево рекурсивних викликів може мати і більш складну структуру, якщо на кожному виклику породжується кілька звернень – фрагмент дерева рекурсії для чисел Фібоначчі представлений на рис. 2.2.

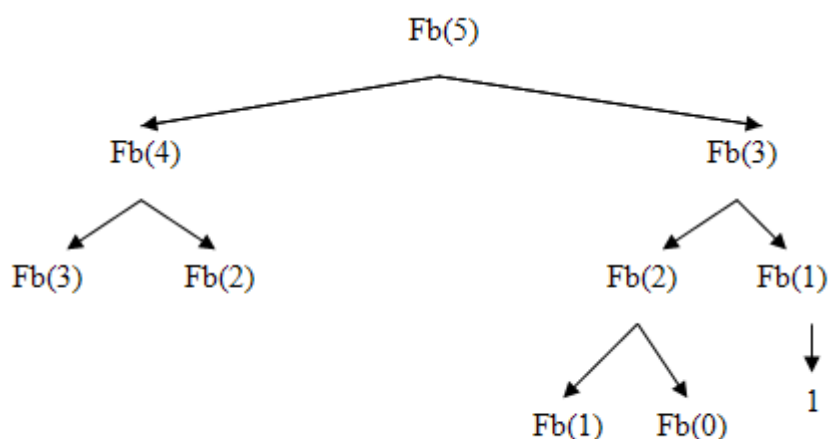


Рис. 2.2. Фрагмент дерева рекурсії при обчисленні чисел Фібоначчі –  $F(5)$

Механізм виклику функції або процедури в мові високого рівня суттєво залежить від архітектури комп'ютера і операційної системи. У рамках IBM PC сумісних комп'ютерів цей механізм реалізований через програмний стек.

Які передаються в процедуру або функцію фактичні параметри, так і які повертаються з них значення, поміщаються в програмний стек спеціальними командами процесора. Додатково зберігаються значення необхідних реєстрів і адреса повернення в процедуру.

Схематично цей механізм ілюстрований на рис 2.3.

Для підрахунку трудомісткості виклику будемо вважати операції поміщення слова в стек і вибірку його з стека елементарними операціями у формальній системі. Тоді при виклику процедури або функції в стек поміщається адреса повернення, стан необхідних реєстрів процесора, адреси повернених значень і передані параметри. Після цього виконується перехід за адресою на викликувану процедуру, яка витягує передані фактичні параметри, виконує обчислення, поміщає їх за вказаними в стеці адресами, і при завершенні роботи відновлює реєстри, виштовхує з стека адресу повернення і здійснює перехід за цією адресою.

Позначимо через:

$m$  – кількість фактичних параметрів, що передаються,

$k$  – кількість значень, що повертаються процедурою,

$r$  – кількість реєстрів в стеку, що оберігаються,

маємо:  $f_{\text{виклик}} = m+k+r+1+m+k+r+1 = 2*(m+k+r+1)$  елементарних операцій на один виклик і повернення.

Аналіз трудомісткості рекурсивних алгоритмів в частині трудомісткості самого рекурсивного виклику можна виконувати різними способами в залежності від того, як формується підсумкова сума елементарних операцій – розглядом окремо ланцюжка рекурсивних викликів і повернень, або сукупно по вершинах дерева рекурсивних викликів.

#### **2.1.5. Аналіз трудомісткості алгоритму обчислення факторіала**

Для розглянутого вище рекурсивного алгоритму обчислення факторіала кількість вершин рекурсивного дерева одно, очевидно,  $n$ , при цьому передається і повертається по одному значенню ( $m = 1, k = 1$ ), в припущенні

про збереження чотирьох регістрів –  $r = 4$ , а на останньому рекурсивному виклику значення функції обчислюється безпосередньо – в результаті:

$$fA(n) = n * 2 * (1 + 1 + 4 + 1) + (n-1) * (1 + 3) + 1 * 2 = 18 * n - 2$$

Відзначимо, що  $n$  – параметр алгоритму, а не кількість слів на вході.

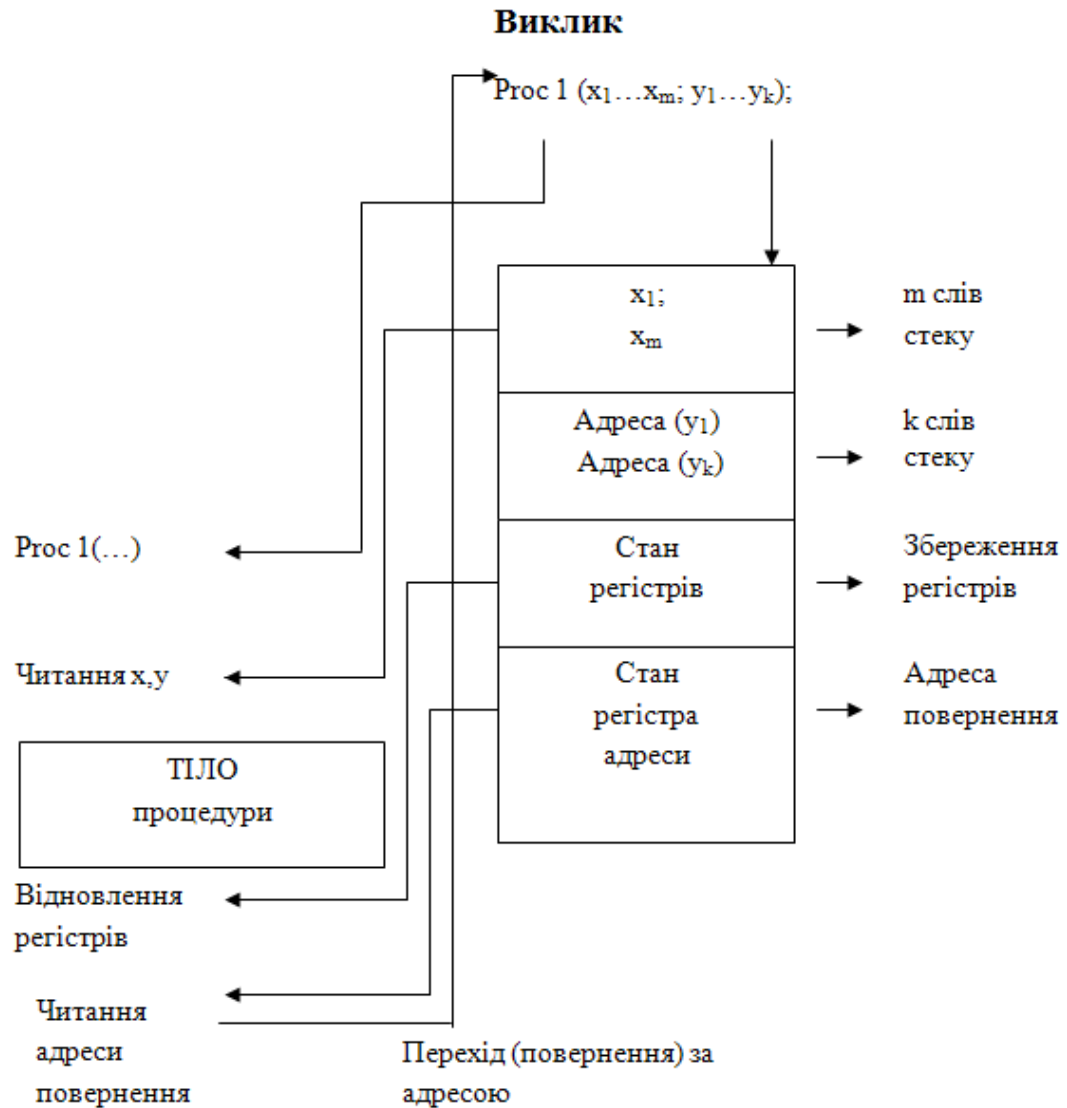


Рис. 2.3. Механізм виклику процедури з використанням програмного стека

## 2.2. Рекурсивні алгоритми і методи їх аналізу

### 2.2.1. Логарифмічні тотожності

При аналізі рекурсивних алгоритмів досить часто використовуються логарифмічні тотожності, далі передбачається, що,  $a > 0$ ,  $b > 0$ ,  $c > 0$ , основа логарифма не дорівнює одиниці:

$$b^{\log_b a} = a; e^{\ln x} = x; \log_b a^c = c * \log_b a; \log_b a = 1 / \log_a b$$

$$\log_b a = \log_c a / \log_c b \Rightarrow \text{записі } Q(\ln(x))$$

Основа логарифма не суттєва, якщо віна більше одиниці, оскільки константа ховається в позначенні  $Q$ .

$$a^{\log_b c} = c^{\log_b a}$$

Можна показати, що для будь-якого  $\xi > 0$ ,  $\ln(n) = o(n^\xi)$ , при  $n \rightarrow \infty$ .

### 2.2.2. Методи рішення рекурсивних співвідношень

В математиці розроблено ряд методів, за допомогою яких можна отримати явний вигляд функції, які задані у рекурсивному виді [1, 5] – метод індукції, формальні степеневі ряди, метод ітерацій і т.д.

Розглянемо деякі з них:

#### а) Метод індукції

Метод полягає в тому, що б спочатку вгадати рішення, а потім довести його правильність за допомогою індукції.

Приклад:

$$\begin{cases} f(0)=1 \\ f(n+1)=2*f(n) \end{cases}$$

Припущення:  $f(n)=2^n$

Базис: якщо  $f(n)=2^n$ , тоді  $f(0)=1$ , що виконано за визначенням.

Індукція: Нехай  $f(n)=2^n$ , тоді для  $n+1 \Rightarrow f(n+1)=2*2^n=2^{n+1}$

Зауважимо, що базис суттєво впливає на рішення. Так, наприклад:

- якщо  $f(0)=0$ , то  $f(n)=0$ ;
- якщо  $f(0)=1/7$ , то  $f(n)=(1/7)*2^n$ ;
- якщо  $f(0)=1/64$ , то  $f(n)=(2)^{n-6}$

#### б) Метод ітерації (підстановки)

Суть методу полягає в послідовній підстановці – ітерації рекурсивного визначення, з наступним виявленням загальних закономірностей:

Нехай  $f(n)=3*f(n/4)+n$ , тоді:

$$f(n)=n+3*f(n/4)=n+3*[n/4+3*f(n/16)]=n+3*[n/4+3\{n/16+3*f(n/64)\}], \text{ і}$$

розкриваючи дужки, отримуємо:

$$f(n)=n+3*n/4+9*n/16+27*n/64+\dots+3^i*n/4^i$$

Зупинка рекурсії відбудеться при  $(n/4^i) \leq 1 \Rightarrow i \geq \log_4 n$ , в цьому випадку останній доданок не більше, ніж  $3^{\log_4 n} \Theta(1) = n^{\log_4 3} \Theta(1)$ .

$$f(n) = n \sum (3/4)^k + n^{\log_4 3} \Theta(1),$$

тобто

$$\sum (3/4)^k = 4/n,$$

то остаточно:

$$f(n) = 4 * n + n^{\log_4 3} \Theta(1) = \Theta(n)$$

### 2.2.3. Рекурсивні алгоритми

Основний метод побудови рекурсивних алгоритмів – це метод декомпозиції. Ідея методу полягає в розкладі задачі на частини меншої розмірності, отримані рішення для кожної частини і об'єднання рішень.

В загальному вигляді, якщо відбувається розділення задачі на  $b$  підзадач, яке призводить до необхідності вирішення  $a$  підзадач розмірністю  $n/b$ , то загальний вигляд функції трудомісткості має вигляд [5]:

$$f_A(n) = a * f_A(n/b) + d(n) + U(n) \quad (2.1),$$

де:

$d(n)$  – трудомісткість алгоритму розподілу задачі на підзадачі,

$U(n)$  – трудомісткість алгоритму об'єднання отриманих рішень.

Розглянемо, наприклад, відомий алгоритм сортування злиттям, що належить Дж. Фон Нейману [5]:

На кожному рекурсивному виклику переданий масив ділиться навпіл, що дає оцінку для  $d(n) = Q(1)$ . Далі рекурсивно викликається сортування отриманих масивів половинній довжини (до тих пір, поки довжина масиву не стане рівною одиниці), і зливаються повернені відсортовані масиви за  $Q(n)$ .

Тоді очікувана трудомісткість на сортування складе:

$$f_A(n) = 2 * f_A(n/2) + \Theta(1) + \Theta(n)$$

Виникає задача про отримання оцінки складності функції трудомісткості, заданої у вигляді (2.1), для довільних значень  $a$  і  $b$ .



#### 2.2.4. Основна теорема про рекурентні співвідношення

Наступна теорема належить Дж. Бентлі, Д. Хакену та Дж. Саксу (1980 г.), достатньо повний доказ цієї теореми наведено в [5].

Теорема.

Нехай  $a \geq 1$ ,  $b > 1$  – константи,  $g(n)$  – функція.

Нехай далі:

$$f(n) = a * f(n/b) + g(n),$$

де

$$n/b = \lfloor (n/b) \rfloor \text{ або } \lceil (n/b) \rceil$$

Тоді:

1) Якщо  $g(n) = O(n^{\log_b a - \xi})$ ,  $\xi > 0$ , то  $f(n) = \Theta(n^{\log_b a})$ .

Приклад:  $f(n) = 8f(n/2) + n^2$ , тоді  $f(n) = \Theta(n^3)$

2) Якщо  $g(n) = \Theta(n^{\log_b a})$ , то  $f(n) = \Theta(n^{\log_b a} * \log n)$ .

Приклад:  $f_A(n) = 2 * f_A(n/2) + \Theta(n)$ , тоді  $f(n) = \Theta(n * \log n)$

3) Якщо  $g(n) = \Omega(n^{\log_b a + \epsilon})$ ,  $\epsilon > 0$ , то  $f(n) = \Theta(g(n))$ .

Приклад:  $f(n) = 2 * f(n/2) + n^2$ , маємо:  $n^{\log_b a} = n^1$ , і відповідно:  $f(n) = \Theta(n^2)$

Дана теорема є потужним засобом аналізу асимптотичної складності рекурсивних алгоритмів, на жаль, вона не дає можливості отримати повний вид функції трудомісткості.

#### Висновки до розділу

В даному розділі рекурсія та рекурсивні алгоритми розглянуто значення рекурсії. Розглянуто категорії задач, що дозволяють рекурсивні визначення. Проведено аналіз реалізації рекурсивних підпрограм. Проведено аналіз трудомісткості рекурсивних алгоритмів методом підрахунку вершин дерева рекурсії.

## РОЗДІЛ III

### РОЗРОБКА СИСТЕМИ ТВОРЧИХ ЗАВДАНЬ ДЛЯ ФОРМУВАННЯ ЗДАТНОСТІ СКЛАДАТИ РЕКУРСИВНІ АЛГОРИТМИ

#### 3.1. Методичні рекомендації щодо опису рекурсивних функцій

Рекурсивно описати функцію – значить не використовувати в ній операторів циклу та переходу, а також не використовувати глобальних (описаних поза функцією) змінних, якщо, звичайно, цього не потрібно за умови завдання.

При описі рекурсивних функцій є типовою наступна ситуація. Якщо вже є рекурсивна формула для обчислення функції (як, наприклад, для факторіалу), то особливих проблем з описом цієї функції мовою Паскаль немає навіть у програмістів-початківців. Проблеми з'являються, коли для вирішення деякого завдання запропоновано рекурсивно описати функцію, але при цьому не сказано, звідки тут береться рекурсія. Спробуємо дати деякі рекомендації щодо того, як треба «виловлювати» рекурсію. Відразу зазначимо, що це не точні, не суворі правила побудови рекурсивних функцій (таких правил взагалі немає), а лише підказки, що вказують напрямок, у якому треба рухатися за такої побудови. Спочатку наведемо ці рекомендації у загальному вигляді, а потім покажемо їх застосування на конкретних прикладах.

У цих рекомендаціях задля стислості викладу будемо використовувати наступні терміни:

вихідне завдання – та задача, для вирішення якої ми описуємо функцію; наприклад, для функції  $F(N)$  вихідним завданням є обчислення факторіалу  $N!$

підзавдання – вихідне завдання, але у простішому варіанті; наприклад-заходів для обчислення  $N!$  підзадачами є обчислення  $(N-1)!$ ,  $(N-2)!$  і т.д.

Отже, нехай потрібно рекурсивно описати функцію для вирішення певного завдання. Тоді слід дотримуватися наступних рекомендацій.

**Рекомендація 1.** Насамперед слід звести вихідне завдання до одного або кількох підзавдань, з вирішення яких можна побудувати рішення вихідного завдання.

Така інформація нам потрібна для того, щоб з'явилася рекурсія. Справді, що означає рекурсивне звернення до цієї функції? Цим зверненням вирішуємо підзавдання, тобто. те саме завдання, що й вихідне, але в більш простому випадку. Наприклад, визначаючи в функції  $F$  факторіал числа  $N$ , ми рекурсивно звертаємося до цієї ж функції для обчислення більш простого факторіалу  $(N-1)!$  Тому, щоб з'явилася рекурсія, потрібно у вихідному завданні хоча б раз розв'язати підзавдання, яке маніпулює простішими даними.

При цьому важливо, щоб вихідне завдання зводилося не до будь-яких підзавдань, а лише до таких, з відповідей яких можна побудувати відповідь для вихідного завдання. Наприклад, при обчисленні  $N!$  можна, звичайно, розглянути підзавдання обчислення  $0!$ , проте з відповіді  $1$  цього підзавдання навряд можна обґрунтовано отримати величину  $N!$  при довільному  $N$ . А ось вирішивши підзавдання обчислення  $(N-1)!$ , Ми зможемо легко отримати величину  $N!$

Зазначимо, що зведення вихідного завдання до підзавдань є найскладнішим і найважливішим моментом у «виловлюванні» рекурсії. Але, на жаль, якихось загальних правил, як це робити, тут немає; все залежить від конкретного завдання.

Але нехай ми так чи інакше зуміли звести вихідне завдання до підзадач. Що робити далі? Потрібно вирішити ці підзадачі. Але як? Про це свідчить наступна рекомендація.

**Рекомендація 2.** Вважаючи, що функція правильно вирішує будь-які підзадачі, для вирішення цих підзавдань слід рекурсивно звернутися до нашої ж функції та з відповідей підзавдань побудувати відповідь на вихідне завдання.

Згідно з цією рекомендацією, описувати дії функції при вирішенні вихідного завдання треба припускаючи, що функція вже вміє вирішувати будь-які підзадачі (доводи на користь цього наведені нижче). Тому для вирішення відібраних підзавдань треба сміливо звернутися до нашої функції. При цьому «лізти» всередину цих рекурсивних звернень не потрібно, а слід

лише зрозуміти, які відповіді будуть видані цими зверненнями і як ці відповіді скомбінувати, щоб отримати відповідь. Наприклад, при описі  $F(N)$  ми використовуємо результат обчислення  $F(N-1)$ , не думаючи, як влаштовано це обчислення.

Зазвичай, таке комбінування не викликає особливих труднощів, так як ця проблема вирішується ще на етапі зведення вихідного завдання до підзавдань.

До цього часу ми розглядали дії функції у рекурсивних гілках. Але в функції мають бути й нерекурсивні гілки, в яких відповіді даються явно, без повторного звернення до визначеної функції. Такі гілки відповідають вирішенню вихідного завдання у найпростіших випадках.

А як дізнатися, в яких випадках треба давати явні відповіді? Наприклад, в функції  $F(N)$  явну відповідь можна дати і для  $N=0$  ( $0!=1$ ), і для  $N=1$  ( $1!=1$ ), і  $N=2$  ( $2!=2$ ), і  $N=3$  ( $3!=6$ ) тощо. Коли зупинитися? Відповідь на це питання дає наступна рекомендація.

**Рекомендація 3.** Явну відповідь треба давати, коли вихідне завдання не зводиться до підзавдань.

Наприклад, факторіал  $0!$  не можна звести до більш простого факторіалу, тому за  $N=0$  у функції  $F$  треба явно вказувати її значення. У той же час факторіал  $1!$  можна звести до більш простого випадку, так як  $1!=1*0!$ , тому виписувати нерекурсивну гілку випадку  $N=1$  не треба, він підпадає під загальний, рекурсивний випадок.

Такими є загальні рекомендації, яких слід дотримуватись при побудові рекурсивної функції. Далі ми розглянемо конкретні випадки їхнього застосування. Але насамперед зробимо пару зауважень.

*Зауваження 1.* Відповідно до рекомендації 1, вихідне завдання треба звести до підзавдань, тобто. до аналогічних більш простих завдань. Але що таке «простіше завдання»?

Загального визначення цього поняття немає, у різних випадках воно розуміється по-різному. Але яке б не було визначення простішого завдання,

воно має бути таким, щоб процес спрощення для будь-якого завдання був кінцевим, тобто. через кінцеве число кроків ми маємо обов'язково прийти до завдання, яке спростити вже не можна. Інакше при виконанні рекурсивної функції, коли завдання зводяться до більш простих завдань, ми ніколи не вийдемо з рекурсії.

Вибір відповідного визначення найпростішої задачі – одна з основних труднощів при зведенні вихідної задачі до підзавдання, цьому аспекту ми будемо приділяти особливу увагу у конкретних прикладах.

*Зауваження 2.* Згідно з рекомендацією 2, описувати поведінку рекурсивної функції в загальному випадку слід у припущенні, що вона діє в більш простих випадках. Тому для того, щоб вирішити підзавдання, потрібно сміливо звернутися до цієї функції.

Наскільки законне це припущення? Тут слід згадати метод математичної індукції. Нехай треба довести деяке твердження  $P(N)$  для всіх цілих  $N \geq 0$ . Тоді, згідно з методом математичної індукції, пропонується діяти так: спочатку треба довести істинність затвердження для найпростішого  $N=0$ , а потім, припускаючи істинним  $P(N-1)$ , треба довести істинність  $P(N)$ . Звідси випливає істинність вихідного твердження:

$$P(0), \forall N: P(N-1) \rightarrow P(N) \Rightarrow \forall N \geq 0: P(N)$$

Зверніть увагу: доводячи істинність  $P(N)$ , ми припускаємо тільки істинність  $P(N-1)$ , але не розуміємо, як доводиться істинність цього припущення.

Аналогічна ситуація і з рекурсивною функцією: якщо функція описана так, що вона правильно працює у найпростіших випадках і вирішує вихідне завдання на основі припущення коректного вирішення нею підзавдань, тоді ця функція в цілому є правильною.

### **3.2. Використання творчих завдань для формування здатності складати рекурсії у числових задачах**

У цьому розділі ми розглянемо побудову рекурсивних функцій, що залежать від цілих аргументів.

### 3.2.1. Рекурсія за величиною числа

У області невід’ємних чисел простішими, зазвичай, вважаються менші за величиною числа. Наприклад, для  $N$  більш простими є числа  $N-1$ ,  $N-2$  і т.д. Очевидно, таке зменшення числа можна зробити лише кінцеву кількість разів – до 0.

**Завдання 1.** Потрібно рекурсивно описати функцію  $f(x, n)$ , яка обчислює величину  $x^n/n!$  за будь-якого дійсного  $x$  та будь-якого невід’ємного цілого  $n$ .

Рішення. Насамперед, згідно з рекомендаціями, обчислення  $f(x, n)$  треба звести до підзадачі – до обчислення  $x^k/k!$  при деякому  $k < n$ . Яке саме  $k$  взяти? Можна, наприклад, вибрати  $k=n-1$  так як за  $x^{n-1}/(n-1)!$  легко отримати  $x^n/n!$  Оскільки ми описуємо  $f(x, n)$  у припущенні, що функція правильно обчислює  $x^k/k!$  при будь-якому  $k < n$ , то в описі функції ми повинні обчислювати  $x^{n-1}/(n-1)!$  рекурсивно звернутися до  $f(x, n-1)$ , а потім отриману величину помножити на  $x/n$ , щоб отримати значення  $f(x, n)$ . Нерекурсивний випадок – обчислення  $f(x, 0)$ , так як обчислення  $x^0/0!$  не можна звести до обчислення  $x^k/k!$  при  $k < 0$ .

Отже, отримуємо опис нашої функції мовою Паскаль:

```
function f(x:real; n:integer): real;      {n ≥ 0}
begin if n=0 then f:=1 else f:=x/n*f(x,n-1)end;
```

Може виникнути питання: якщо вихідне завдання можна звести до різних підзавдань, за відповіддю кожного з яких можна побудувати відповідь вихідного завдання, то яке з цих підзавдань вибрати? У цьому прикладі ми звели обчислення  $f(x, n)$  до обчислення  $f(x, n-1)$ , тобто. зменшили другий параметр на 1. А чи можна було вибрати інше підзавдання, наприклад, обчислення  $f(x, n-2)$ ? Так, можна, оскільки за відповіддю цієї підзадачі легко виходить відповідь вихідного завдання:

$$f(x, n) = \frac{x^2}{n \cdot (n-1)} \cdot f(x, n-2)$$

Але за такого вибору нам доведеться вказувати дві нерекурсивні гілки – при  $n=0$  та  $n=1$ , так як при цих значеннях  $n$  вираз  $x^{n-2}/(n-2)!$  не має сенсу. У результаті отримуємо:

```

function f(x:real; n:integer): real; {n ≥ 0}
begin if n=0 then f:=1 else
      if n=1 then f:=x else f:=x*x/n/(n-1)*f(x,n-2)
end;

```

Цей приклад показує, що при виборі підзавдання, до якого ми зводимо вихідне завдання, небажано занадто «далеко» йти від вихідного завдання, краще брати підзавдання, найбільш «близьке» до нього.

Однак буває і так, що за відповідями «близьких» підзадач не вдається побудувати відповідь вихідного завдання, і тому доводиться використовувати досить «далеке» підзавдання. Розглянемо відповідні приклади.

**Завдання 2.** Не використовуючи операції множення та поділу, рекурсивно описати функцію  $M(a,b)$  від цілих чисел  $a$  і  $b$  ( $a \geq 0$ ,  $b > 0$ ), яка обчислює залишок від ділення  $a$  на  $b$ , тобто.  $M(a,b) = a \bmod b$ .

Рішення. Тут насамперед треба визначитися з тим, за яким параметром функції  $M$  будемо вести рекурсію, тобто. який параметр спрощуватимемо. Вести рекурсію за другим параметром  $b$  не можна, оскільки ніякої хорошої залежності між  $a \bmod b$  і  $a \bmod c$ , де  $c < b$ , не має. Тому рекурсію вестимемо за першим параметром  $a$ .

При  $a \geq b$  правильна рівність  $a \bmod b = (a-b) \bmod b$ . Тому якщо ми зможемо обчислити  $M(a-b,b)$ , то зможемо обчислити і  $M(a,b)$  – ці величини збігаються; отже, вихідну задачу треба зводити до підзадачі, де величина  $a$  зменшена не на 1, а на  $b$ .

Що ж до нерекурсивного випадку, він виникає при  $a < b$ , оскільки різниця  $a-b$  виходить із області допустимих значень першого параметра функції. І тут відповіддю є саме число  $a$ , так як  $a \bmod b = a$ . Отже, отримуємо такий опис нашої функції:

```

function M(a, b:integer): integer; {a ≥ 0, b > 0}
begin if a < b then M:=a else M:=M(a-b,b) end;

```

**Завдання 3.** Описати рекурсивну функцію  $degree5(N)$ , яка обчислює, яким ступенем числа 5 є натуральне число  $N$ . Якщо  $N$  не ступінь п'яти, функція

повинна повернути число -1. Наприклад,  $\text{degree5}(50) = -1$ ,  $\text{degree5}(125) = 3$ ,  $\text{degree5}(5) = 1$ ,  $\text{degree5}(1) = 0$ .

Рішення. Ідея циклічного розв'язання задачі зрозуміла: отже ділити наше число на 5, поки це можливо. В результаті серії таких поділів ми або дійдемо до одиниці (це позитивний результат, при якому шуканий показник відповідає числу виконаних поділів), або в якийсь момент з'ясується, що ділити націло на 5 далі неможливо (негативний результат з відповіддю -1).

Схожі міркування покладемо в основу рекурсивного розв'язання завдання. У цьому зауважимо, що якщо величина  $N \div 5$  – ступінь п'ятірки з показником  $k$ , то очевидно, що і число  $N$  – теж ступінь п'ятірки, але з показником  $k+1$ . Також зауважимо, що одиниця - це найменша ступінь п'ятірки з показником 0.

Звернемо увагу, що вихідне завдання для числа  $N$  слід звести до підзадачі для числа  $N \div 5$  лише тоді, коли наше число  $N$  ділиться націло на 5 (так як в цьому випадку залишається надія на позитивний результат розв'язання задачі), інакше відповідь вже зрозуміла (-1). Спрощення завдання тут йде у напрямку переходу до ступеня з меншим показником і спроби дійти до ступеня з мінімальним показником.

Нерекурсивних випадків тут два: перший виникає при  $N = 1$  (позитивний результат з відповіддю 0), другий - якщо  $N$  не вдалося поділити націло на 5 (негативний результат з відповіддю -1).

```
function degree5(N: integer): integer; {N>=1 }  
    var k: integer;  
    begin if N=1 then degree5:=0 else  
        if N mod 5 <> 0 then degree5:= -1 else  
            begin k:=degree5(N div 5);  
                if k=-1 then degree5:=-1  
                    else degree5:=k + 1  
            end  
        end;  
    end;
```



Ми розглянули випадки, де значенням параметрів були негативні цілі числа, а тепер розглянемо випадок будь-яких цілих чисел.

**Завдання 4.** Рекурсивно описати функцію  $\text{pow}(x,n)$ , яка обчислює  $x^n$  для будь-якого дійсного  $x$  ( $\neq 0$ ) та будь-якого цілого  $n$ .

Рішення. Оскільки  $x^n = x \cdot x^{n-1}$ , то, здавалося б, обчислення  $x^n$  треба зводити до обчислення  $x^{n-1}$ . Однак, це не так.

Особливість цього завдання в тому, що другий параметр ( $n$ ) функції  $\text{pow}$ , за яким будемо вести рекурсію, може бути будь-яким цілим числом, а в області цілих чисел процес спрощення числа  $n$  шляхом віднімання з нього 1 нескінченний:  $n, n-1, \dots, 1, 0, -1, -2, \dots$ . Тому ми ніколи не дійдемо до такого значення  $n$ , при якому рекурсія зупиниться. З урахуванням цього, в області цілих чисел треба якось інакше визначити поняття більш простого числа. Зробити це можна по-різному.

Можливий варіант: розглянути в області цілих чисел дві підобласті - позитивні і негативні числа, і в кожній з них використовувати своє розуміння більш простого числа, вважаючи простішим числом те, яке ближче до 0:

$n > 0$ :  $n, n-1, n-2, \dots, 2, 1$  (тут  $n-1$  простіше  $n$ )

$n < 0$ :  $n, n+1, n+2, \dots, -2, -1$  (тут  $n+1$  простіше  $n$ )

А для  $n=0$  можна дати відповідь:  $\text{pow}(x,0) = 1$ .

Якщо так і зробити, то отримаємо наступну рекурсивну формулу:

$$x^n = \begin{cases} 1, & n=0 \\ x \cdot x^{n-1}, & n>0 \\ x^{n+1}/x, & n<0 \end{cases}$$

У цьому випадку опис функції  $\text{pow}(x,n)$  виглядає так:

```
function pow(x:real; n:integer): real; {x<>0}
```

```
begin if n=0 then pow:=1 else
```

```
    if n>0 then pow:=x*pow(x,n-1) else pow:=pow(x,n+1)/x
```

```
end;
```

Інший варіант: оскільки правильна формула  $x^{-n} = 1/x^n$  ( $n > 0$ ), то можна за негативного показника перейти до позитивного згідно з цією формулою, а потім традиційно спростувати позитивне число, віднімаючи від нього 1. Тут

для негативного числа більш простим вважається його модуль, а для позитивного – число, на 1 менше його.

Це дає таку рекурсивну формулу:

$$x^n = \begin{cases} 1, & n=0 \\ 1/x^{|n|}, & n<0 \\ x \cdot x^{n-1}, & n>0 \end{cases}$$

І тоді опис функції pow виглядає так:

```
function pow(x:real; n:integer): real; {x≠0}
begin if n=0 then pow:=1 else
      if n<0 then pow:=1/pow(x,abs(n))
      else pow:=x*pow(x,n-1)
end;
```

Можна придумати й якісь інші визначення більш простого числа в області цілих чисел, але в будь-якому разі треба уважно слідкувати за тим, щоб процес спрощення був кінцевим.

### 3.2.2. Рекурсія за записом числа у позиційній системі числення

Тепер розглянемо клас завдань обробки цілих чисел, де важлива не величина чисел, а набір цифр з яких складені записи цих чисел. Таке, наприклад, завдання знаходження суми цифр у десятковому записі заданого негативного цілого числа. Зрозуміло, що величина числа не відіграє тут головну роль, оскільки сума цифр, скажімо, у 7-ному записі того самого числа буде іншою.

У таких завданнях простішим слід вважати не число, менше вихідного числа за величиною, а число, запис якого отримуємо відкиданням однієї цифри (наприклад, останньої, самої правої) із запису вихідного числа: 123 простіше 1234. Зрозуміло, що отримане таким способом число простіше (у ньому менше цифр) і що подібне спрощення чисел можливе лише кінцеву кількість разів.

Саме з цим більш простим числом має працювати підзавдання, до якого зводимо вихідне завдання. Нерекурсивний випадок виникає для числа однієї цифри, так як відкидання цифри з однозначного числа призводить до порожнього запису, але не запису числа.

Розглянемо кілька прикладів для обробки цифр у запису чисел.

**Завдання 1.** Рекурсивно описати функцію  $\text{maxdig}(N)$ , яка знаходить найбільшу цифру в десятковому записі негативного цілого числа  $N$ . Наприклад,  $\text{maxdig}(27306) = 7$ .

**Рішення.** Згідно з запропонованою рекурсивною схемою, як підзавдання беремо знаходження найбільшої цифри в числі, отриманому з вихідного числа (скажімо, з 27306) відкиданням останньої цифри (у числі 2730). Далі, згідно з загальною рекомендацією, припускаємо, що наша функція правильно вирішує будь-яке підзавдання, тому для вирішення нашої підзадачі рекурсивно застосовуємо функцію до 2730, в результаті чого вона видасть у відповіді цифру 7. Тепер треба побудувати відповідь для вихідного завдання: порівнюємо отриману цифру 7 з останньою цифрою 6 вихідного числа та найбільшу з них видаємо як відповідь вихідного завдання. Для однозначного числа видаємо саме це число як відповідь. Отже, маємо такий опис:

```
function maxdig(N: integer): integer; {N≥0}
var m, last: integer;
begin
    if N<10 then maxdig:=N else
        begin last:=N mod 10; {остання цифра}
            m:=maxdig(N div 10); {найбільша цифра серед інших}
            if last>m then maxdig:=last else maxdig:=m
        end
    end;
end;
```

**Завдання 2.** Рекурсивно описати функцію  $\text{Head3}(N)$ , яка обчислює число, що отримується приписуванням зліва цифри 3 до десяткового запису цілого негативного числа  $N$ . Наприклад:  $\text{Head3}(1592) = 31592$ .

**Рішення.** І тут вихідне завдання, скажімо, із числом 1592, зводимо до підзадачі, де розглядається число 159, отримане з вихідного числа відкиданням останньої цифри. Припускаючи, що функція правильно вирішує підзавдання, ми рекурсивно звертаємося до функції і отримуємо для нашої

підзадачі відповідь 3159. Як з цього числа побудувати відповідь для вихідного числа? Треба просто приписати праворуч до цього числа останню цифру вихідного числа, тобто. обчислити  $3159 * 10 + 2$ . У нерекурсивному разі, тобто. при однозначному числі, приписування ліворуч цифри 3 реалізується додатком 30 до цього числа, наприклад, для числа 7 маємо:  $30+7=37$ . Отримуємо наступний опис функції:

```
function Head3(N: integer): integer; {N ≥ 0}
begin if N < 10 then Head3 := 30 + N
      else Head3 := Head3(N div 10) * 10 + N mod 10
end;
```

Якщо у всіх попередніх прикладах ми описували рекурсивні функції, тепер наведемо приклад рекурсивної процедури.

**Завдання 3.** Рекурсивно описати процедуру RevPrint(N), яка друкує у зворотному порядку цифри десяткового запису цілого невід'ємного числа N. Наприклад, RevPrint(12345) має вивести текст 54321.

Рішення. Знову розглядаємо підзавдання для числа, отриманого з вихідного числа, скажімо, 12345 видаленням останньої цифри, тобто. для числа 1234. Рішення цієї підзадачі означає висновок 4321. Як отримати правильний висновок для вихідного числа? Для цього треба спочатку вивести останню цифру 5 вихідного числа, а потім рекурсивно звернутися до функції для вирішення даної підзадачі, щоб вивести в зворотному порядку інші цифри. Для однозначного числа перевертання означає просто виведення цього ж числа. Отримуємо такий опис:

```
procedure RevPrint(N: integer); {N ≥ 0}
begin if N < 10 then write(N)
      else begin write(N mod 10); RevPrint(N div 10) end
end;
```

Звернімо увагу, яким коротким виявився рекурсивний опис цієї процедури; циклічне розв'язання задачі було б значно довшим.

### 3.2.3. Більш складні завдання

У наведених вище прикладах рекурсія велася лише за одним параметром. Розглянемо тепер завдання, де рекурсія необхідна відразу за декількома параметрами.

**Завдання 1.** Описати рекурсивну функцію `equal(N, S)` (де  $N$  і  $S$  – невід'ємні цілі числа), яка перевіряє, чи співпадає сума цифр у десятичному записі числа  $N$  зі значенням  $S$ . Наприклад: `equal(12345,15) = true`, `equal(24,7) = false`, `equal(100,1) = true`.

Рішення. Тут при переході від вихідного завдання до підзадачі спрощуємо одразу два параметри: у числа  $N$  відкидаємо його молодшу цифру, а значення  $S$  зменшуємо при цьому на величину втраченої цифри. Тим самим, вирішення вихідного завдання повністю залежить від розв'язання отриманої більш простої підзадачі. Найпростіший випадок має місце для числа з однієї цифри: за результатом порівняння чисел  $N$  і  $S$  визначається шукана відповідь.

Маємо наступний опис функції `equal`:

```
function equal(N,S: integer): boolean; {S,N>=0}
begin if N<10 then equal:=N=S
      else equal:=equal(N div 10,S-N mod 10)
end;
```

Однак при уважному розгляді отриманого рішення можна помітити його істотний недолік. Нехай, наприклад, відбулося звернення до нашої функції з аргументами  $N=12345$  і  $S=4$ . допустимих значень, тому рекурсивно викликати `equal(1234, -1)` не можна. Є два способи виправлення рішення. Перший полягає у використанні охорони перед рекурсивним викликом:

```
function equal(N,S: integer): boolean; {S,N>=0}
var d:0..9;
begin if N<10 then equal:=N=S
      else
        begin d:= N mod 10;
          if S<d then equal:= false
```

```

else equal:=equal(N div 10,S-d)
end

```

```
end;
```

Як бачимо, текст функції ускладнився, до того ж з'явилася локальна змінна, що автоматично означає втрати пам'яті при рекурсивних викликах. Інший спосіб дозволяє уникнути цих недоліків. Він полягає у розширенні області допустимих значень аргументів функції: довизначимо  $\text{equal}(N,S)$  на область негативних значень  $S$ , вважаючи, що  $\text{equal}(N,S) = \text{false}$  для будь-якого  $S < 0$ . Отримуємо:

```

function equal(N,S: integer): boolean;
begin if S<0 then equal:=false else
      if N<10 then equal:=N=S
      else equal:=equal(N div 10,S-N mod 10)
end;

```

Цей прийом – розширення області значень і довизначення функції – досить плідний прийом у програмуванні рекурсії, часто дозволяє зробити текст функції найбільш коротким.

Розглянемо одну нестандартну задачу, що показує, як вводити додаткові параметри, якщо вони потрібні для проведення рекурсії, але на жаль відсутні у необхідної функції.

**Завдання 2.** Рекурсивно описати функцію  $\text{divs}(N)$  для підрахунку кількості всіх дільників цілого числа  $N$  ( $N > 1$ ), без урахування дільників 1 і  $N$ . Наприклад:  $\text{divs}(5)=0$ ,  $\text{divs}(18)=4$ .

Рішення. Зрозуміло, що вирішувати завдання (з обмежень її умови) необхідно рекурсивно. Але як тут можна «виловити» рекурсію? Спроба як-небудь спростити параметр  $N$  навряд чи зможе привести нас до якоїсь осмисленої підзадачі, з вирішення якої можна побудувати рішення вихідного завдання. Тут слід шукати інший підхід. У зв'язку з цим звернемо увагу на те, що шукати дільники  $N$  потрібно лише серед чисел з діапазону  $[2..N \text{ div } 2]$ . Замість вихідної розглянемо допоміжне завдання: підрахувати кількість

дільників числа  $N$  (фіксованого), що лежать в діапазоні  $[k..N \text{ div } 2]$ . Вирішення вихідної задачі виходить як розв'язання допоміжної задачі для  $k = 2$ .

Допоміжне завдання можна вирішувати рекурсивно. Завдання по підрахунку дільників з діапазону  $[k .. N \text{ div } 2]$  зводиться до підзадачі підрахунку дільників з більш вузького діапазону  $[k+1 .. N \text{ div } 2]$  і одночасної перевірки, чи є при цьому викинуте з діапазону число  $k$  дільником нашого  $N$ . Нерекурсивним випадком у задачі буде пустий діапазон, в якому немає дільників числа  $N$ .

Але все-таки ще не ясно, за рахунок якого параметра проводити рекурсію, звужуючи діапазон (рухаючи праворуч його нижню межу)? Адже потрібна функція  $\text{divs}(N)$  повинна мати лише один параметр  $N$ , якого, як ми зрозуміли, нам явно мало. Підемо тоді на наступну «хитрість». У тілі функції  $\text{divs}(N)$  опишемо допоміжну рекурсивну функцію  $\text{divs1}(k)$ , яка займається підрахунком дільників числа  $N$  серед «кандидатів» з діапазону  $[k.. N \text{ div } 2]$ . Тоді відповідь для вихідної функції  $\text{divs}(N)$  співпадатиме з результатом обчислення функції  $\text{divs1}(2)$  (тут 2 – нижня межа діапазону, що перевіряється).

```
function divs(N: integer): integer; {N>1}
    function divs1(k: integer): integer;
        {k-ть дільників числа N серед кандидатів
        з діапазону [k..N div 2]}
        begin if k>N div 2 then divs1:= 0 {діапазон порожній}
            else divs1:= ord(N mod k = 0) + divs1(k+1)
        end;
    begin divs = divs1(2) end;
```

Звернемо увагу, що результат перевірки числа  $k$ , що відкидається, оформлений у вигляді виразу  $\text{ord}(N \text{ mod } k = 0)$ , що в даному випадку зручно, так як у такий спосіб в потрібну суму відразу ж поставляється потрібна нам відповідь 0 або 1 (без залучення умовних операторів).

### **3.3. Використання творчих завдань для формування здатності складати рекурсію та послідовно організовані дані**

Розглянемо структуровані дані – одновимірні масиви, файли та списки. Зазначені структури даних мають послідовну природу: масив – кінцева послідовність перенумерованих елементів, файл та списки також є послідовностями елементів. При роботі з такими даними природно організовувати рекурсію відповідно до структури даних. Вихідне завдання можна представити як комбінацію підзадач: 1) обробка першого елемента; 2) обробка решти послідовності. Найпростішим (нерекурсивним) випадком при цьому буде обробка порожньої послідовності.

Звичайно, застосування рекурсії до кожної структури – масиву, файлу, списку – має свої особливості, які будуть розглянуті у відповідних пунктах.

#### **3.3.1. Рекурсивна робота з одновимірними масивами**

Оскільки масив складається з наперед заданої фіксованої кількості елементів, рекурсію безпосередньо організувати неможливо: не можна звести завдання обробки масиву до завдання обробки цього ж масиву, спрощення даних не відбувається. У цьому випадку використовують наступний прийом. Замість вихідної задачі обробки масиву, скажімо,  $x[1..n]$ , розглядають завдання обробки вибраного підмасиву, наприклад, що складається з елементів  $x_k, x_{k+1}, \dots, x_n$ . Нове завдання вирішується рекурсивно, оскільки зводиться до обробки елемента  $x_k$  та обробки більш короткого підмасиву  $x_{k+1}, \dots, x_n$ . Вихідна задача при цьому полягає в обробці підмасиву  $x_1, \dots, x_n$ . Розглянемо, як працює цей прийом на такому прикладі.

**Завдання 1.** Нехай є опис `const n = 100; type vector = array[1..n] of real;`

Рекурсивно описати процедуру `PrNeg (x)`, яка друкує від'ємні елементи вектора  $x$ .

Рішення. Застосуємо розібраний прийом: розглянемо, як надрукувати від'ємні елементи підмасиву  $x_k, x_{k+1}, \dots, x_n$ . Для вирішення цього завдання спочатку потрібно надрукувати  $x_k$ , якщо він негативний, а потім надрукувати всі від'ємні елементи підмасиву  $x_{k+1}, \dots, x_n$ . Нерекурсивним випадком буде



ситуація, коли оброблено весь масив, тобто.  $k > n$ , у разі нічого робити не потрібно.

Рішення допоміжної підзадачі обробки масиву оформимо у вигляді вкладеної рекурсивної процедури від параметра  $k$ .

```
procedure PrNeg(var x: vector);  
    procedure PrNeg1(k: integer);  
        begin if  $k \leq n$  then  
            begin if  $x[k] < 0$  then write (x[k]);  
                PrNeg1(k+1)  
            end  
        end;  
    begin PrNeg1(1) end;
```

Як наступний приклад розглянемо задачу перевірки, чи є деяка пропозиція паліндромом. Нагадаємо, що паліндром - це пропозиція, яка однаково читається від початку до кінця і від кінця до початку. Приклади паліндромів: «а роза упала на лапу Азора»; «дорога за город»; «молебен о коне белом» та інші.

Подаємо пропозицію у вигляді масиву символів. Для простоти припустимо, що в цій пропозиції немає розділових знаків - використовуються тільки пробіли і малі літери. Прогалини вважаються незначними символами. Нехай довжина речення не перевищує ста знаків. Опишемо відповідну константу та тип масиву:

```
const n = 100; type txt = array [1..n] of char;
```

**Завдання 2.** Рекурсивно описати логічну функцію  $\text{Pal}(t)$ , яка перевіряє, чи є пропозиція  $t$  типу  $\text{txt}$  паліндромом.

Рішення. По суті, в задачі потрібно перевірити, чи є симетричною послідовність букв, що міститься в  $t$ . Для аналізу симетричності потрібно переглядати  $t$  з двох сторін і порівнювати відповідні елементи  $t$  (див. рис. 3.1).

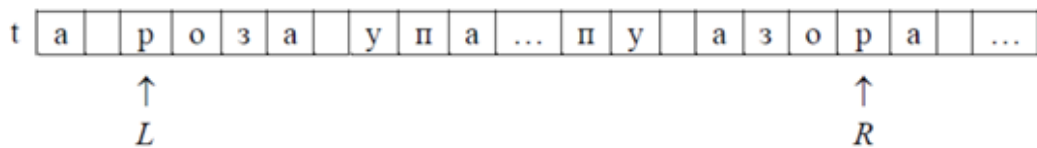


Рис. 3.1. Перевірка букв на симетричність

Використовуємо позначення:  $L$  – індекс найлівішого, а  $R$  – індекс найправішого непроглянутого символу. Нехай  $t[L]$  і  $t[R]$  – не пробіли. Якщо  $t[L] \neq t[R]$ , пропозиція не є паліндромом. Якщо ж  $t[L] = t[R]$ , пропозиція буде паліндромом, якщо послідовність символів між  $t[L]$  та  $t[R]$  – паліндром. У процесі аналізу індекси  $L$  та  $R$  рухаються назустріч один одному. Аналіз завершується, коли значення  $L$  стане більше або дорівнює значення  $R$ .

Якщо  $t[L]$  (або  $t[R]$ ) – пробіл, потрібно перейти до наступного символу  $t[L+1]$  (відповідно,  $t[R-1]$ ).

Як і в прикладі 1, всередині функції  $Pal(t)$  опишемо допоміжну функцію для організації рекурсії. Як початок рекурсії у разі  $L \geq R$  повертатимемо значення `true`, оскільки цей випадок реалізується, коли переглянуті всі пари відповідних символів і жодного разу не зустрілося розбіжність літер.

```
function Pal(var t: txt): boolean;
    function P (L, R: integer): boolean;
    begin if L>=R then P:= true
          else if t[L]=' ' then P:= P(L+1,k)
               else if t[R]=' ' then P:= P(L,R-1)
               else {t[L],t[R] - букви}
                   if t[L]<>t[R] then P:= false
                   else P:= P(L+1,R-1)
    end;
begin Pal: = P (1, n) end;
```

Як останній приклад рекурсивної роботи з масивами розглянемо реалізацію алгоритму бінарного пошуку елемента в упорядкованому масиві. Нехай є масив  $X$  цілих чисел, упорядкований за зростанням елементів, і ціле число  $k$ . Завдання полягає в тому, щоб перевірити, чи зустрічається число  $k$  в

масиві  $X$ . Метод бінарного пошуку полягає в наступному. Порівнюємо  $k$  із середнім елементом масиву  $X$ . Якщо числа збіглися, елемент знайдено, пошук закінчено. В іншому випадку можливі два випадки:  $k$  менше середнього елемента, тоді воно може бути тільки в першій половині масиву; якщо число  $k$  більше середнього елемента, воно може бути лише у правій половині масиву (так як масив упорядкований за зростанням елементів). У будь-якому разі, кількість елементів, серед яких є сенс продовжувати вести пошук, скорочується вдвічі.

**Завдання 3.** Нехай є опис `const n=100; type vector=array[1..n] of integer;`

Рекурсивно описати функцію `Search(X, k)`, що визначає, чи входить ціле число  $k$  упорядкований за зростанням масив  $X$  типу `vector`.

Рішення. Використовуємо змінні  $L$  та  $R$  для позначення меж підмасиву, в якому ведеться пошук. Спочатку  $L = 1$ ,  $R = n$ . У процесі роботи межі  $L$  і  $R$  зближуються, переходячи через середину підмасиву, поки не буде знайдений шуканий елемент (відповідь `true`) або поки підмасив не закінчиться ( $L > R$ , відповідь `false`).

```
function Search(var X: vector; k: integer): boolean;
    function Search1(L,R: integer): boolean;
        var m: integer;
        begin
            if L>R then Search1:=false
            else begin m:=(L+R) div 2; {індекс середнього елемента}
                if k=X[m] then Search1:=true
                else if k<X[m] then Search1:=Search1(L,m-1)
                    else Search1:=Search1(m+1,R)
                end
            end
        end;
    begin Search:=Search1(1,N) end;
```

Слід підкреслити, що зазвичай для роботи з масивами застосовують циклічні алгоритми. Для розглянутих вище завдань не важко написати ітеративні рішення. У реальній програмістській практиці так і слід чинити.

### **3.3.2. Рекурсивна робота з файлами**

При рекурсивній обробці файлів необхідно звернути увагу на те, що процедури, що встановлюють режим роботи з файлом (reset, rewrite), потрібно застосовувати лише один раз. Якщо ми помістимо reset у рекурсивну процедуру:

```
procedure P(var f: text);  
begin reset(f); {***} P(f){***} end;
```

при кожному рекурсивному виклику буде заново виконуватися reset (f), поточна позиція буде зрушуватися на початок файлу f, файл буде простежуватися кожен раз з самого початку, таким чином, рекурсія зациклиться. Цю проблему легко вирішити. Досить рекурсивно оформити обробку не всього файлу, а тільки його частини, починаючи з поточної позиції і до кінця. Підготовку ж файлу до роботи слід зробити в основній процедурі (функції).

**Завдання 1.** Нехай є опис: type numbers=file of real;

Рекурсивно описати процедуру PrNeg (f), яка друкує негативні елементи файлу чисел f.

Рішення. Спочатку будемо вважати, що f вже відкрито для читання. Якщо файл f є порожнім, з ним нічого робити не потрібно. Якщо ж f не порожній, треба прочитати один елемент, надрукувати його, якщо елемент негативний, і обробити аналогічну частину файлу. Ці дії опишемо як допоміжні процедури без параметрів PrNeg1. В основній процедурі залишається відкрити f на читання та викликати процедуру PrNeg1.

```
procedure PrNeg(var f: numbers);  
var a: real;  
    procedure PrNeg1;  
begin
```

```

        if not eof(f) then
            begin read(f,a); if a<0 then write(a); PrNeg1 end
        end;
    begin reset(f); PrNeg1 end;

```

Для економії пам'яті рекурсивна процедура PrNeg1 працює із глобальною змінною a. Кожен рекурсивний виклик змінює значення a, так що в процесі обробки файлу змінна a приймає відтак значення всіх елементів файлу і після завершення рекурсії містить останній елемент файлу f. Однак це не заважає роботі процедури PrNeg1, оскільки прочитане з f значення відразу ж обробляється, його не потрібно зберігати до завершення обробки частини файлу, що залишилася.

**Завдання 2.** Нехай є опис `type numbers = file of real`;

Написати процедуру `Inverse (f1, f2)`, яка у зворотному порядку записує у файл f2 елементи файлу дійсних чисел f1.

Рішення. Як і в попередньому прикладі, рекурсивну частину описуємо у вкладеній допоміжній процедурі `Inv`. Основна процедура `Inverse` буде встановлювати режими роботи для файлів f1 і f2 і викликати процедуру `Inv`.

Найпростіший, нерекурсивний випадок маємо, якщо файл f1 порожній; тоді потрібно повернути порожній файл f2, процедура `Inv` ніяких дій не повинна виконати. Нехай f1 не порожній. Як переставити його елементи у зворотному порядку, якщо вважати, що ми вміємо робити таку перестановку для будь-якого більш короткого файлу? Треба прочитати і запам'ятати перший елемент (при цьому кількість необроблених елементів f1 скоротиться на 1), застосувати `Inv` для запису в f2 у зворотному порядку, що залишилася, більш короткої частини f1, потім записати в f2 запам'ятований елемент.

```

procedure Inverse (var f1, f2: numbers);
    procedure Inv;
        var a: real;
        begin if not eof(f) then
            begin read(f1,a); Inv; write(f2,a) end
        end;
    end;

```

end;

begin reset(f1); rewrite(f2); Inv end;

Зауважимо, що використання локальної змінної - принциповий момент у вирішенні даної задачі. У міру обробки файлу f1 його елементи записуються в пам'ять як значення локальної змінної а послідовних рекурсивних викликів. Зрештою, всі елементи f1 опиняються в оперативній пам'яті. Коли досягнуто кінець файлу f1, процес рекурсивних викликів закінчується і починається закриття у зворотному порядку викликаних процедур із записом значень локальних змінних послідовних викликів у файл f2. Таким чином у f2 виявляються елементи файлу f1, переставлені в зворотному порядку.

Необхідно зробити таке зауваження. Немає жодної гарантії, що наша процедура зможе успішно обробити файл будь-якої довжини, пам'яті може просто не вистачити, і тоді програма аварійно завершить свою роботу. Файл з погляду мови Паскаль - структура даних, що зображує зовнішнє запам'ятовуючий пристрій на магнітній стрічці. Вся робота з файлами будується на тому, що файл цілком не можна помістити в пам'ять і на тому, що працювати з файлом необхідно поелементно. Розглянута задача є суто навчальним прикладом, ілюстрацією використання локальних змінних у рекурсії.

**Завдання 3.** Рекурсивно описати функцію Empty(t), яка визначає з скількох порожніх рядків починається текстовий файл t.

Рішення. Рішення полягає в перегляді та підрахунку порожніх рядків на початку файлу, поки файл не скінчиться або поки не зустрінеться непустий рядок. Нерекурсивними випадками є ситуації, коли файл порожній або файл починається з непустого рядка. У цих випадках слід повернути 0. Рекурсивний випадок: прочитати перший порожній рядок файлу і додати 1 до порожніх рядків в частині файлу, що залишилася.

```
function Empty(var: text): integer;  
    function Empty1: integer;  
    begin if eof(t) then Empty1:= 0
```

```

else if not eoln(t) then Empty1:= 0
      else begin readln(t); Empty1:= 1+Empty1 end
end;
begin reset(t); Empty:= Empty1 end;

```

Отже, якщо потрібно оформити рекурсивну обробку масиву або файлу, потрібно описувати вкладені рекурсивні функції та процедури. Справа йде інакше при роботі зі списками, що ми побачимо у наступному пункті.

### 3.3.3. Рекурсивна робота зі списками

Розглянемо лінійні однонаправлені списки без заголовної ланки (див. рис. 3.2). [1, 2] Зображений на рисунку 3.2 список будемо записувати у вигляді  $(e_1, e_2, \dots, e_n)$ .

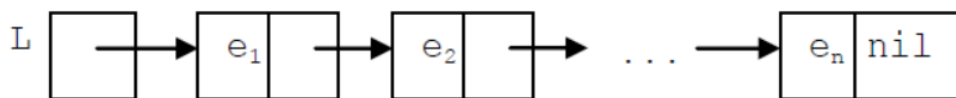


Рис. 3.2. Список

Рекурсивне оформлення обробки списку, як правило, не представляє труднощів. Наведемо необхідні описи:

```

type TE = ...; {тип елементів списку}
list = ↑chain;
chain = record elem: TE; next: list end;

```

**Завдання 1.** Описати рекурсивну процедуру PrNeg(L), яка друкує негативні елементи списку дійсних чисел L (TE=real).

**Рішення.** Якщо список L порожній (L = nil), процедура нічого не повинна робити. Якщо ж L не порожній, треба надрукувати перший елемент, якщо він негативний, а потім обробити частину списку, що залишилася.

```

procedure PrNeg(L: list);
begin if L <> nil then
    begin if L↑.elem < 0 then write(L↑.elem);
          PrNeg(L↑.next)
    end
end

```

end;

Як бачимо, ніякі додаткові вкладені процедури, подібні до тих, що використовувалися при роботі з масивами і з файлами, при обробці списків не потрібні.

**Завдання 2.** Описати рекурсивну функцію `Increase(L)`, яка перевіряє, чи впорядковано елементи списку чисел `L` по неспаданню ( $TE = \text{integer}$ ).

Рішення. Сформулюємо ідею рекурсивного рішення. Порожній список і список з одного елемента, як завжди, вважаються упорядкованими. Якщо ж у списку два і більше елементів, він буде упорядкованим, якщо перший елемент менше другого і список, що починається з другого елемента, впорядкований за зростанням.

```
function Increase(L: list): boolean;  
begin if L=nil then Increase:= true  
      else if L↑.next=nil then Increase:=true  
            else if L↑.elem>L↑.next↑.elem  
                  then Increase:=false  
                  else Increase:=Increase (L↑.next)  
end;
```

Звернімо увагу на важливу особливість рішення. У рекурсивній гілці можна було використовувати оператор присвоєння `Increase:=(L↑.elem<=L↑.next↑.elem) and Increase(L↑.next)`, явно відповідний сформульованій ідеї рішення. Однак з міркувань ефективності була використана конструкція з повним умовним оператором, що забезпечує для неупорядкованих списків менше операцій. Справді, якщо при перевірці першої пари елементів з'ясувалося, що перший елемент більший за другий, відповідь відомий – список не впорядкований – і немає сенсу робити рекурсивні виклики для решти списку.

Розглянемо тепер завдання не на перегляд, а на зміну списку. Розв'яжемо задачу включення елемента в упорядкований список. Важливою особливістю подібних завдань є необхідність передачі в процедуру параметра-змінної, а не



параметра-значення. Посилання на першу ланку може змінитися, наприклад, якщо елемент включається до порожнього списку або перед першою ланкою списку.

**Завдання 3.** Описати рекурсивну процедуру Insert (L, k), яка включає число k в упорядкований по неспаданню список L так, щоб збереглася вихідна впорядкованість (TE = integer).

Рішення. Якщо розв'язувати задачу без рекурсії, за допомогою циклів, рішення вийде таким: знайти місце у списку, куди потрібно вставити елемент k, створити ланку, записати в неї число k та підчепити ланку у потрібне місце списку. Причому досить більшість рішення припадає на пошук першого елемента, більшого або рівного k.

У рекурсивному вирішенні цієї проблеми не виникає: спробуємо вставити число k перед першим елементом списку, а якщо перший елемент менше k, застосуємо ті ж дії до списку, що починається з другого елемента і т.д. Таким чином, розкручування рекурсії призводить до того, що елемент, перед яким потрібно вставити k визначається автоматично.

Окремого розгляду вимагають нерекурсивні випадки: ситуація включення до порожнього списку і включення перед першим елементом списку. Дії, які потрібно виконати в цих випадках, одні й самі, а саме, створити ланку, записати в неї число k і зробити нову ланку першою ланкою списку. Щоб не виписувати одну і ту ж послідовність операторів двічі, скористаємося логічною змінною before, що об'єднує ці випадки.

```
procedure Insert(var L: list; k: integer);  
var t: List; before: boolean;  
begin  
    if L=nil then before:= true else before:= L↑.elem>=k;  
    if before then  
        begin new(t); t↑.elem:= k; t↑.next:= L; L:= t end  
    else Insert(L↑.next, k)  
end;
```

Розглянемо докладніше, як відбувається включення елемента в середину списку. Нехай маємо список  $M = (100, 300)$ . Виконання процедури  $\text{Insert}(M, 200)$  відбувається в такий спосіб. Ім'я параметра  $L$  асоціюється з ім'ям  $M$ , в локальну змінну  $k$  записується число 200. В результаті виконання першого умовного оператора змінна  $\text{before}$  отримує значення  $\text{false}$  (так як  $100 < 200$ ). Другий умовний оператор викликає рекурсивну гілку  $\text{Insert}(L \uparrow .\text{next}, k)$  для  $L=M$  та  $k=200$ . На початку роботи процедури ім'я  $L$  асоціюється зі змінною  $M \uparrow .\text{next}$  і в створену локальну змінну  $k$  записується 200. Тепер ім'я  $L$  вказує на ланку, що містить число 300. При виконанні першого умовного оператора виконується порівняння  $300 \geq 200$  і вперед записується значення  $\text{true}$ . Виконання другого умовного оператора створює новий запис, у полі  $\text{elem}$  записує 200, на полі  $\text{next}$  – значення  $L$ , тобто. посилання на ланку з елементом 300. Далі  $L$  (яка зображує змінну  $M \uparrow .\text{next}$ , тобто поле  $\text{next}$  першої ланки списку  $M$ ) записується посилання на створений запис (200). Нова ланка виявляється прикріпленою до попередньої ланки.

### **3.4. Використання творчих завдань для формування здатності складати рекурсивне оброблення двійкових дерев**

При вирішенні наступних завдань зручно спиратися на наступне рекурсивне визначення двійкового дерева. Двійкове дерево - це кінцева множина безліч елементів, яка або порожня, або містить один елемент, званий коренем, а інші елементи діляться на дві непересічні підмножини, кожна з яких саме є двійковим деревом. Ці підмножини називаються лівим і правим піддеревами вихідного дерева. Кожен елемент двійкового дерева називається вершиною цього дерева. Рекурсія тут полягає в тому, що вихідне двійкове дерево визначається через двійкові дерева з меншою кількістю вершин (і в кінцевому рахунку зводяться до порожніх дерев).

Отже, кожне піддерево двійкового дерева – це також двійкове дерево. Тому вихідне завдання на обробку двійкового дерева будемо зводити до більш простих аналогічних завдань на обробку його дерев. У кінцевому рахунку таке зведення дасть нам найпростіші (нерекурсивні) випадки - дерево з однією

вершиною і/або порожнє дерево (залежно від конкретної задачі будуть використовуватися або обидва випадки, або якийсь один). Можливі й інші окремі випадки, що відповідають явним відповідям, що багато в чому залежить від розв'язуваного завдання (наприклад, з'ясувалося, що порушено властивість, а отже, немає сенсу в подальших перевірках і діях, тому що відповідь відома).

При розгляді прикладів завдань ми скористаємося уявленням дерева у вигляді сукупності однотипних динамічних об'єктів комбінованого типу (записів), пов'язаних між собою за допомогою посилань. Кожен запис представляє деяку вершину дерева і зберігає елемент, а також посилання на корені лівого та правого піддерев цієї вершини. Опишемо відповідні типи:

```
type TE = ...; {тип елемента дерева}
tree = ↑node;
node = record elem: TE; left, right: tree end;
```

Зазначимо деякі особливості. При такому поданні у всі вершини, крім кореня, входить рівно одна гілка. З кожної вершини можуть виходити 2 гілки: ліва та права. Причому, ліва гілка веде до коріння лівого піддерев, а права гілка – у корінь правого піддерев цієї вершини. Вершина, з якої не виходить жодної гілки, називається листом.

Тепер почнемо розгляд конкретних завдань.

**Завдання 1.** Написати рекурсивну функцію  $\text{Member}(T, E)$ , що визначає, чи входить елемент  $E$  в дерево  $T$  ( $TE = \text{integer}$ ).

Рішення. Найпростіший випадок тут відноситься до порожнього дерева, для якого даємо відповідь `false` (елемента  $E$  немає, тому що в порожньому дереві взагалі вершин немає). Якщо ж дерево непусте, то через покажчик  $T$  нам доступна коренева вершина, а значить ми зможемо перевірити, чи зберігається в ній шуканий елемент  $E$ . Якщо так, то відповідь готова (це нерекурсивна гілка, що відповідає відповіді `true`). Якщо ні, то вихідне завдання розпадається на дві підзадачі з перевірки вмісту лівого та правого піддерев. Якщо пошук елемента  $E$  у лівому піддереві увінчався успіхом, то на цьому

завершуємо роботу функції з відповіддю true (досліджувати праве піддерево у разі вже немає сенсу – лише втратимо час). Якщо ж у лівому піддереві елемент E не знайдено, то остаточна відповідь повністю залежатиме від результатів пошуку в правому піддереві. Отримуємо наступний опис нашої функції:

```
function Member(T: tree; E: TE): boolean;
begin
    if T=nil then Member:=false else {перевірка кореня:}
        if T↑.elem=E then Member:=true else {загальний випадок:}
            if Member(T↑.left,E) then Member:=true else
                Member:=Member(T↑.right,E)
end;
```

**Завдання 2.** Написати рекурсивну функцію Leaves(T) для підрахунку кількості листя у дереві T .

Рішення. У загальному випадку вихідне завдання по підрахунку листя в заданому дереві зводиться до підзавдання по підрахунку листя в лівому та правому піддеревих (рекурсія!) та підсумовування отриманих відповідей. Нерекурсивних гілок тут дві. Перша відповідає відповіді 0 – для порожнього дерева (у ньому листя немає, бо немає вершин). Друга відповідає відповіді 1 - для дерева з однієї вершини (вона є і коренем і листом одночасно). Приходимо до наступного рішення:

```
function Leaves(T: tree): integer;
begin
    if T=nil then Leaves:=0 else {перевірка на лист:}
        if (T↑.left=nil)and(T↑.right=nil) then Leaves:=1 else
            {рекурсія:} Leaves:=Leaves(T↑.left)+ Leaves(T↑.right)
end;
```

**Завдання 3.** Написати рекурсивну функцію Height(T) знаходження висоти дерева T (вважати, що висота непустиого дерева – це число

вершин на найдовшому зі шляхів від кореня дерева до листя, висота порожнього дерева – 0).

Рішення. Очевидно, що для непустиого дерева  $T$  висота  $h(T) = \max(h(T \uparrow .\text{left}), h(T \uparrow .\text{right})) + 1$ . Тому в загальному випадку первісну задачу можна звести до двох підзавдань зі знаходження висот лівого та правого піддерев. Тоді відповідь – це найбільша з двох отриманих висот, збільшена на 1. Найпростіший випадок тут відповідає порожньому дереву, висота якого, за визначенням, 0.

```
function Height(T: tree): integer;
var hL,hR: integer; {висоти лівого та правого піддерев}
begin if T=nil then Height:=0 else {загальний випадок:}
    begin
        hL:=Height(T↑.left); hR:= Height (T↑.right);
        if hL>hR then Height:=hL+1 else Height:=hR+1
    end
end;
```

**Завдання 4.** Написати рекурсивну функцію  $\text{Level}(T,n)$  для підрахунку числа вершин на  $n$ -му ( $n \geq 1$ ) рівні дерева  $T$  (вважати, що перший рівень відповідає кореню дерева, а рівень будь-якої іншої вершини – на одиницю більше рівня її батьківської вершини).

Рішення. При спрощенні завдання і переході від вихідного дерева до його піддерев ми повинні враховувати наявність у нашої функції другого параметра, що задає номер ( $n$ ) рівня. Тому підрахунок числа вершин на рівні  $n$  у дереві  $T$  (загальний випадок) зводитимемо до підрахунку числа вершин на рівні  $n-1$  у лівому та правому його піддеревах (рекурсія!), а потім – до підсумовування отриманих відповідей. Найпростіших випадків тут два. Перший відповідає порожньому дереву, де немає вершин (незалежно від заданого значення  $n$  даємо відповідь 0). Другий - відповідає найменшому (1) значенню  $n$  при непустому дереві: на 1-му рівні в непустому дереві є рівно одна вершина - корінь (даєм відповідь 1). Приходимо до такого рішення:

```
function Level(T: tree; n: integer): integer; {n>=1}
begin if T=nil then Level:=0 else {непусте дерево:}
```

```

    if n=1 then Level:=1 else {загальний випадок:}
        Level:= Level(T↑.left,n-1)+ Level(T↑.right,n-1)
    end;

```

**Завдання 5.** Описати рекурсивну функцію IsPos(T), яка перевіряє, чи є у дереві T хоча б один шлях (від кореня до листка включно), у всіх вершинах якого є лише позитивні елементи (TE=real).

Рішення. Якщо дерево T непусте (загальний випадок), то перевіривши елемент з кореневої вершини, ми відразу ж визначимо, чи є далі сенс шукати в цьому дереві заданий шлях. Справді, якщо елемент докорінно – позитивний (вдалий початок шляху), тоді все залежатиме від обставин в піддеревах. Якщо при цьому шуканий шлях знайдеться при проходженні через ліве піддерево, то праве розглядати немає сенсу. Інакше пошук слід продовжити у правому піддереві. По відношенню до піддерев застосовні аналогічні міркування, тим самим ми «зловили» рекурсію.

Виявимо тепер можливі окремі випадки. По-перше, це порожнє дерево - в ньому немає вершин, а, значить, немає і будь-якого шляху (даємо відповідь false). По-друге, якщо в корені зберігається елемент 0, то шуканого шляху точно немає (так як всі можливі шляхи виходять тільки з кореня). У цьому випадку даємо відразу відповідь false і не продовжуємо подальшого пошуку. По-третє, корінь (з позитивним елементом) може бути єдиною вершиною дерева, отже, шуканий шлях цим знайдено, тобто. дісталися листа - кінцевої вершини шляху (даємо відповідь true). Отримуємо наступний опис функції:

```

function IsPos(T: tree): boolean;
begin

```

```

    if T=nil then IsPos:=false

```

```

    else {перевірка вмісту кореневої вершини:}

```

```

        if T↑.elem<=0 then IsPos:=false {шляху немає}

```

```

        else {корінь позитивний:}

```

```

            if (T↑.left=nil)and(T↑.right=nil) then

```

```

                IsPos:=true {це лист, тобто. шлях знайдено}

```

```

else {пошук шляху в лівому піддереві:}
    if IsPos(T↑.left) then IsPos:=true
    else {пошук шляху в правому піддереві:}
        IsPos:=IsPos(T↑.right)
end;

```

Зазначимо, що в процесі роботи нашої рекурсивної функції звернення до чергового піддерева відповідає переходу на наступний (глибший) рівень рекурсії. Якщо піддерево не підійшло (його кореневий елемент  $\leq 0$  чи піддерево – порожнє), відбувається повернення до попереднього рівня рекурсії. Рекурсія тут розгортається складним каскадом: спуск по дереву (шляхом переходу до його піддерев) чергується з підйомом до попередніх (раніше пройдених) вершинам. Рано чи пізно відбудеться одне з двох: або на черговому рівні рекурсії ми опинимося в позитивному листі (потрібний шлях побудувати вдалося), або ми перевіримо всі можливі варіанти і жоден з них не підійде (шлях не знайдено).

**Завдання 6.** Написати рекурсивну процедуру Copy (T, T1), яка буде копіювати дерево T1 - копію дерева T.

Рішення. Якщо дерево T – порожнє, то, очевидно, що його копія T1 – порожнє дерево (це найпростіший випадок, який задає нерекурсивну гілку). Для копіювання непустилого дерева скористаємося рекурсією: спочатку виділимо пам'ять під корінь дерева T1, що формується, і запишемо туди елемент з кореня дерева T; а далі (тобто з використанням процедури Copy) побудуємо копії лівого та правого піддерев, записавши посилання на побудовані піддерева в поля left і right створеного кореня. Приходимо до такого опису нашої процедури:

```

procedure Copy(T: tree; var T1: tree);
begin
    if T=nil then {випадок порожнього дерева:} T1:=nil else
        begin {копіювання непустилого дерева:}
            new(T1); {Виділення пам'яті під корінь дерева T1}

```

```

    T1↑.elem:=T↑.elem; {копіювання елемента з кореня}
    Copy(T↑.left,T1↑.left);{копіювання лівого піддерева}
    Copy(T↑.right,T1↑.right){копіювання правого піддерева}
  end
end;

```

Звернемо увагу, оскільки дерево T1 є результатом роботи процедури, параметр T1 необхідно передати за посиланням. Зауважимо також, що за рахунок зазначеного способу передачі даного параметра посилання на побудовані копії лівого та правого піддерев будуть автоматично записані в поля left та right кореневої вершини дерева T1 (бо зазначені поля передаються як фактичні параметри при рекурсивному зверненні до процедури Copy). Нарешті, завдяки передачі цього параметра за посиланням у поля left і right кінцевих вершин (листя) будуть автоматично записані порожні посилання (див. гілку then).

**Завдання 7.** Написати рекурсивну процедуру Leaves (T), яка видаляє (зі звільненням пам'яті) все листя дерева T.

Рішення. Почнемо з розгляду окремих випадків, яких тут два. По-перше, це порожнє дерево, в якому немає вершин, а отже, немає і листя; над таким деревом не треба робити будь-яких дій. По-друге, це дерево з єдиної вершини (вона – одночасно лист і корінь); цю вершину слід видалити. У загальному випадку, коли в дереві більше однієї вершини, вихідне завдання зводимо до двох підзадач з видалення всіх листків у лівому і правому його піддерев. Отримуємо наступне рішення:

```

procedure Leaves(var T: tree);
begin {дії робити тільки з непустим деревом!}
  if T<>nil then {дерево непусте, у ньому одна вершина?}
  if (T↑.left=nil)and(T↑.right=nil)then {так, одна}
  begin {видалення єдиної вершини, тобто. листа:}
    dispose(T); {вивільнення пам'яті, зайнятої під аркуш}
    T:=nil {після видалення листа дерево стало порожнім}
  end
end

```



```

end else {у дереві є більше однієї вершини}
begin {рекурсивна обробка піддерев}
    Leaves(T↑.left); Leaves(T↑.right)
end

```

end;

Зазначимо, що параметри нашої процедури важливо передавати за посиланням (а не за значенням!). Це забезпечить збереження змін, виконаних над параметром  $T$  у тілі процедури. Зокрема, для дерева з більш ніж однієї вершини при видаленні його листя значення `nil` будуть автоматично записуватися в поля `left` (якщо лист був ліворуч – див. рис. 3.3) або `right` (якщо листок був праворуч – див. рис. 3.4) батьківських вершин (т.к. ці поля вказуються як фактичні параметри при рекурсивному зверненні до процедури `Leaves`).

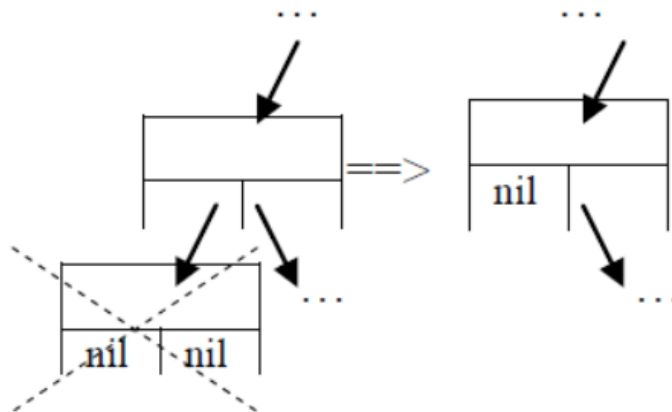


Рис. 3.3. Видалення листка зліва

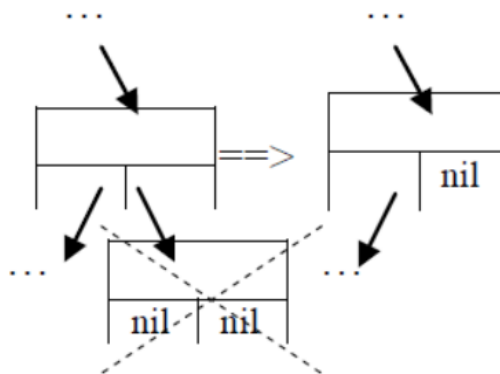


Рис. 3.4. Видалення листка справа

**Завдання 8.** Рекурсивно описати функцію Same(T), що визначає, чи є у дереві символів T хоча б два однакові символи (TE=char).

Рішення. Зазначимо, що, за умовою завдання, у вершинах дерева зберігаються елементи символьного типу. Згадаймо також, що у мові Паскаль з урахуванням значень типу char можна будувати множини. Скористаємося допоміжною множиною наступним чином.

Організуємо обхід дерева (відвідуючи кожную вершину тільки один раз), і в процесі обходу заноситимемо в нашу множину елементи з переглянутих вершин. При цьому будемо діяти за принципом: якщо елемент із розглянутої вершини раніше нам не зустрічався (тобто цього елемента зараз у множині немає), то додамо його в множину, а інакше (елемент вже присутній у множині) - завершимо огляд дерева, так як виявлено повтор. Решту елементів (з непроглянутих вершин) враховувати немає сенсу.

Ясно, що множина перед початком обходу дерева має бути порожньою. Причому, початкову ініціалізацію множини слід виконати лише один раз (у розділі операторів функції Same), а вже потім запускати рекурсивний обхід. Тому в тілі основної функції Same потрібно описати допоміжну рекурсивну функцію Same1, яка і виконуватиме такий обхід (використовуючи глобальну по відношенню до неї множину). Результат роботи Same1 буде прийнято за відповідь для основної функції Same.

Наведемо спочатку опис функції Same, а потім зробимо кілька додаткових зауважень.

```
function Same(T: tree): boolean;  
var S: set of char; {елементи з переглянутих вершин}  
    function Same1(T: tree): boolean;  
        begin {обхід дерева до першого збігу}  
        if T=nil the Same1:=false else {перевірка кореня:}  
            if T↑.elem in S then Same1:=true else  
                begin {елемент з кореня раніше не зустрічався}  
                    S:=S+[T↑.elem]; {додати новий елемент у S}
```

```

        if Same1(T↑.left) then Same1:=true
            else Same1:= Same1(T↑.right)
        end
    end;

begin S:=[]; Same:= Same1 (T) end;

```

У міру обходу заданого дерева будуть розглядатися (за рахунок рекурсивних звернень) всілякі його піддерева, в результаті чого в множину  $S$  будуть заноситись нові і нові елементи з пройдених вершин. Нехай відбулося чергове звернення до функції  $\text{Same1}$ , і нехай в корені чергового піддерева, що розглядається, зустрівся елемент, що не належить множині  $S$ . Тоді цей новий елемент додається в  $S$ , а далі - аналогічно обробляється спочатку ліве піддерево. Якщо в процесі обходу цього лівого піддерева знайшлася вершина з елементом, що вже знаходиться в множині  $S$ , то даємо відповідь `true` і завершуємо роботу функції. Інакше – відповідь залежатиме від результатів аналогічного перегляду правого піддерева. Зрештою, спрацює одне із двох. Або в корені якогось чергового піддерева зустрінеться елемент, раніше включений у множину  $S$  (відповідь `true`), або в процесі повного обходу дерева всі його елементи виявляться різними (відповідь `false`). Зазначимо, що окремий випадок  $T = \text{nil}$  дуже важливий, і не тільки для початкового порожнього дерева  $T$  (в якому повторюваних елементів, очевидно, немає), але і для порожніх піддерев, до яких можуть відбуватися звернення в процесі повного обходу  $T$ . також, що функція  $\text{Same}(T)$  отримує результат `false` після закінчення повного перегляду вершин дерева  $T$ , коли її рекурсивний виклик виконується для поля `right (= nil)` у самого правого листа.

Отже, розглянуті приклади показують, наскільки зручно працювати з деревами рекурсивно (з рекурсивної природи цих структур даних). Використані в прикладах рекурсивні функції та процедури при цьому відрізняються не тільки великою лаконічністю та наочністю, але й високою ефективністю. Звичайно, існують і циклічні алгоритми обробки дерев, але вони, як правило, моделюють рекурсію.

### **Висновки до розділу**

В даному розділі наведено методичні рекомендації щодо опису рекурсивних функцій. Розглянуто використання творчих завдань для формування здатності складати рекурсії у числових задачах, складати рекурсію та послідовно організовані дані, складати рекурсивне оброблення двійкових дерев.

## ВИСНОВКИ

В даній магістерській роботі було розглянуто розробку та дослідження системи творчих завдань для формування здатності складати рекурсивні алгоритми.

Не дивлячись на все різноманіття задач із програмування, їх можна розбити на обмежений набір базових алгоритмів. Мистецтво програмування полягає у представленні розв'язання загальної задачі через комбінацію відомих базових алгоритмів – проектуванні програми. Але одну й ту ж задачу можна представити різними комбінаціями базових алгоритмів. Тому чим більшим набором базових алгоритмів володіє програміст, чим ефективніше вміє їх комбінувати, тим вища його кваліфікація. У залежності від глибини розділення програм на функціональні частини можна отримати різну кількість базових алгоритмів.

Рекурсія – це фундаментальне поняття, яке означає покрокове отримання результату, причому на наступному кроці використовується результат, отриманий на попередньому кроці.

Рекурсія може бути отримана трьома способами:

- 1) Циклічний виклик оператора  $x = f(x)$ , де змінній  $x$  у лівій частині присвоюється нове значення, отримане за допомогою деякої функції  $f$  від старого значення.
- 2) Визначення рекурсивної функції, якщо така можливість передбачена мовою програмування, як у C++ і Паскалі. У вузькому програмістському розумінні рекурсія – це виклик у процедурі або функції її самої. Головний момент у побудові рекурсивної програми – визначення умови припинення рекурсії.
- 3) Враховуючи механізм виклику підпрограми через стек, можна побудувати програму з штучною рекурсією, створивши стек як динамічну структуру даних і записуючи туди у циклі значення

змінних програми, а потім відновлюючи їх ("штучний вихід з рекурсії").

Обрання того чи іншого способу отримання рекурсії визначається ефективністю отриманої програми, до характеристик якої входять:

- довжина коду (кількість операторів); - швидкодія (час виконання);
- необхідний обсяг пам'яті (як загальний, так і розбитий на окремі сегменти).

Рекурсивна програма найчастіше простіша за нерекурсивну, але кожен вхід у рекурсивну функцію та вихід з неї займає певний час, що уповільнює роботу програми.

Щодо обсягу пам'яті, то слід розрізняти, яку саме область пам'яті використовує програма і які на неї накладені обмеження.

Так, циклічна програма використовує основну пам'ять даних розміром до 64 К. Рекурсивна функція використовує системний стек, розмір якого обмежується директивою компіляції (максимум до 64К), але кожен вхід в рекурсію вимагає великих витрат пам'яті стека.

Програма зі штучним стеком використовує або динамічну пам'ять (розмір до 640К порціями по 16 байт), або пам'ять даних (в залежності від способу утворення стека).

Великим плюсом рекурсії є те, що вона дозволяє коротко і логічно ясно сформулювати алгоритм, що наочно демонструють розглянуті приклади. Ітеративні алгоритми вирішення тих самих завдань, як правило, довші.

Недоліком рекурсії є її затратність, що з особливостей виконання рекурсивних функцій і процедур. Згадаймо, що при кожному рекурсивному виклику створюються нові локальні змінні, до того ж необхідно запам'ятовувати точку повернення (місце, куди потрібно передати управління після завершення виконання процедури). Причому оцінити заздалегідь за текстом процедури чи функції, скільки буде зроблено рекурсивних викликів не можна – глибина рекурсії залежить від значення параметра. При великій

глибині рекурсії пам'яті комп'ютера може не вистачити, що призведе до аварійного завершення програми, що виконується.

Не можна забувати і тимчасові витрати: кожен виклик процедури або функції вимагає час на передачу параметрів (створити локальну змінну, присвоїти їй відповідне значення) і час на передачу управління процедурі, а також на повернення з неї.

Рекурсія – потужний та витончений механізм, однак, витратний за часом та пам'яттю. При вирішенні конкретних завдань програміст повинен зробити вибір на користь рекурсії або циклів, виходячи з особливостей завдання.

Якщо завдання легко вирішується ітеративно, рекурсію використовувати не варто. Однак якщо завдання полягає в роботі з рекурсивними структурами даних або сама природа завдання рекурсивна, немає сенсу відмовлятися від рекурсії.

Але завжди слід мати на увазі витрати на рекурсивні виклики і намагатися уникати їх, якщо можливо - використовуючи додаткові змінні для збереження вже отриманого результату або шляхом своєчасного виходу з процедури (функції), коли відповідь вже отримана.

При розв'язуванні творчих задач з програмування у школярів формуються уміння та навички розв'язування творчих задач, що сприяє їх розумовому розвитку, а також розвиваються такі мислительні операції, як класифікація та систематизація, при розв'язанні такого плану задач забезпечується творчість та дискусії у пошуку рішення, необхідність пояснювати власну відповідь та розмірковувати.

Підсумовуючи вищесказане, ми будемо вважати, що творча задача з програмування—це така задача, що передбачає пошук та побудову алгоритму її розв'язування з використанням існуючих методів, з подальшою реалізацією певною мовою програмування, у процесі чого учні активно засвоюють нові знання, опановують уміння та навички, розвивають абстрактне та логічне мислення, власні творчі здібності, пізнавальний інтерес.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Donald E. Knuth. The Art of Computer Programming. Volume 1: Fundamental Algorithms. — Addison-Wesley, 1997.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. — MIT Press, 2022.
3. Robert Sedgewick, Kevin Wayne. Algorithms. — Addison-Wesley, 2011.
4. Harold Abelson, Gerald Jay Sussman, Julie Sussman. Structure and Interpretation of Computer Programs. — MIT Press, 1996.
5. Jon Bentley. Programming Pearls. — Addison-Wesley, 2000.
6. David Harel, Yishai Feldman. Algorithmics: The Spirit of Computing. — Springer, 2004.
7. Steven S. Skiena. The Algorithm Design Manual. — Springer, 2020.
8. Erik D. Demaine, MohammadTaghi Hajiaghayi. The Bidimensionality Theory and Its Algorithmic Applications. // Journal of the ACM, 2005.
9. Tarjan, R. E. Depth-First Search and Linear Graph Algorithms. // SIAM Journal on Computing, 1972.
10. Mitchell Wand. Continuation-Based Program Transformation Strategies. // Journal of the ACM, 1980.
11. Jeffrey D. Ullman, Alfred V. Aho, John E. Hopcroft. Data Structures and Algorithms. — Addison-Wesley, 1983.
12. Brian W. Kernighan, Dennis M. Ritchie. The C Programming Language. — Prentice Hall, 1988.
13. Ахо, А. Структуры данных и алгоритмы / Ахо А., Хопкрофт Дж., Ульман Дж. — М.: Издательский дом «Вильямс», 2001. — 384 с.
14. Вакалюк Т. А. Структурне програмування мовою Pascal (лабораторний практикум). Навчальний посібник для студентів фізико-математичного факультету.— Вид. 2-ге / Тетяна Анатоліївна Вакалюк, Сергій Станіславович Жуковський.— Житомир : Вид-во ЖДУ, 2010.— 124 с.



15. Вирт Н. Алгоритмы и структуры данных / Н. Вирт – 2-ое изд., испр. – СПб.: Невский диалект, 2001. – 352 с.
16. Задачи по программированию / [С.М. Окулов, Т.В. Ашихмина, Н.А. Бушмелева и др.]. – [Под ред. С.М. Окулова]. – М. : БИНОМ Лаборатория знаний, 2006. – 820 с.
17. Златопольский Д. М. Сборник задач по программированию / Д. М. Златопольский. – [2-е изд., перераб. и доп.]. – СПб. : БХВ-Петербург, 2007. – 240 с.
18. Кноп К. Творческие задачи [Электронный ресурс] / К. Кноп. – Режим доступа : URL : <http://offline.computerra.ru/1998/237/1143>. – Название с экрана.
19. Кнут, Д. Искусство программирования. Тома 1, 2, 3. 3-е изд. / Д. Кнут. Уч. пос. – М.: Изд. дом "Вильямс", 2001. – 385 с.
20. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест – М.: МЦНМО, 2001. – 960 с.
21. Курс олімпійця [Електронний ресурс]. – Режим доступу: URL : <http://www.e-olimp.com/ua/articles/group14>. – Назва з екрана.
22. Макконнелл Дж. Анализ алгоритмов. Вводный курс / Макконнелл Дж. – М.: Техносфера, 2002. – 304 с.
23. Меньшиков Ф. В. Олимпиадные задачи по программированию / Ф. В. Меньшиков. – СПб. : Питер, 2006. – 315 с.
24. Программирование на языке Pascal: задачник / [Под. ред. Усковой О.Ф.] – СПб: Питер, 2003. – 336 с.
25. Юркин А. Г. Задачник по программированию. Учебное пособие / А. Г. Юркин. – СПб. : Питер, 2002. – 192 с.
26. Semenikhina O., Rudenko Yu. Problems of educating to programming of students and way of their overcoming. Information technologies and learning tools, 2018. 4(66). pp. 54-64.

- 27.Васенко О. В. Реалізація можливостей інтегрованого середовища розробки Lazarus у вивченні інформатики в школі. Комп'ютер у школі та сім'ї. 2016. № 7. С. 32-35.
- 28.Глинський Я.М., Палюшок Л.В. Яку мову програмування вивчати у школі (матеріали для дискусії). Комп'ютер у школі та сім'ї. 2013. № 8. С. 9-18.
- 29.Горошко Ю., Костюченко А., Шкардибарда М. Використання ВПЗ у процесі вивчення основ програмування. Інформатика та інформаційні технології. 2012. №1. С. 22–25.
- 30.Навчальна програма з інформатики (профільний рівень) для 10-11 класів загальноосвітніх шкіл, затверджена Наказом Міністерства освіти і науки № 1407 від 23 жовтня 2024 року. URL: <https://mon.gov.ua/ua/osvita/zagalna-serednya-osvita/navchalni-programi/navchalni-programi-dlya-10-11-klasiv> (дата звернення: 01.11.2024).
- 31.Навчальна програма з інформатики (рівень стандарту) для 10-11 класів загальноосвітніх шкіл, затверджена Наказом Міністерства освіти і науки № 1407 від 23 жовтня 2023 року. URL: <https://mon.gov.ua/ua/osvita/zagalna-serednya-osvita/navchalni-programi/navchalni-programi-dlya-10-11-klasiv> (дата звернення: 01.11.2024).
- 32.Нова українська школа – Веб-ресурс НУШ. URL: <https://nus.org.ua/>
- 33.Про затвердження Державного стандарту базової і повної загальної середньої освіти. URL: <https://zakon.rada.gov.ua/laws/show/1392-2011-%D0%BF> (дата звернення: 01.11.2024).