Структуру заняття для магістрів за темою "Пошук інформації в інформаційно-аналітичних системах" можна розробити наступним чином:

Частина 1: Вступ (15 хвилин)

1. **Привітання та організаційні моменти** (5 хвилин)
   - Привітання студентів.
   - Огляд плану заняття.
   - Визначення цілей заняття.
2. **Вступ до інформаційно-аналітичних систем** (10 хвилин)
   - Короткий огляд інформаційно-аналітичних систем.
   - Важливість пошуку інформації.
   - Значення структури даних.

Частина 2: Теоретичні аспекти (30 хвилин)

1. **Структура даних в інформаційно-аналітичних системах** (15 хвилин)
   - Опис різних типів структур даних: бази даних, XML, JSON, індекси тощо.
   - Значення структуризації даних для ефективного пошуку.
2. **Алгоритми пошуку** (15 хвилин)
   - Перелік та класифікація алгоритмів пошуку.
   - Порівняння алгоритмів з точки зору ефективності та придатності для різних типів даних.

Частина 3: Практичні демонстрації (30 хвилин)

1. **Демонстрація структури даних** (15 хвилин)
   - Реальний приклад структури даних у системі.
   - Маніпуляції з даними (вставка, видалення, оновлення).
2. **Демонстрація алгоритмів пошуку** (15 хвилин)
   - Програмування базових алгоритмів пошуку (лінійний пошук, двійковий пошук).
   - Використання готових бібліотек для пошуку в реальних даних.

Частина 4: Практичні завдання та обговорення (15 хвилин)

1. **Виконання практичного завдання студентами** (10 хвилин)
   - Студенти роблять невелике завдання на пошук, використовуючи навчене.
   - Завдання може бути зосереджене на використанні конкретного алгоритму пошуку в заданій структурі даних.
2. **Обговорення та відповіді на питання** (5 хвилин)
   - Відповіді на питання студентів.
   - Обговорення розповсюджених помилок та кращих практик.

Частина 5: Висновки та домашнє завдання (10 хвилин)

1. **Підведення п

ідсумків заняття** (5 хвилин)

- Коротке нагадування основних пунктів заняття.
- Важливість знань для майбутньої професійної діяльності.

1. **Розподіл домашнього завдання** (5 хвилин)
   - Деталізація домашнього завдання, яке може бути пов'язане з розробкою або аналізом алгоритмів пошуку.
   - Підкреслення термінів виконання та критеріїв оцінювання.

За такою структурою, заняття має баланс між теоретичними основами і практичними вправами, що дозволить студентам краще засвоїти матеріал.

Давайте розробимо контент для розділу "Структура даних в інформаційно-аналітичних системах". Нижче наведено опис різних типів структур даних із прикладами коду на Python для кожного типу.

1. Бази даних

Опис: База даних — це організована колекція даних, яка може бути легко доступна, керована та оновлювана. Бази даних часто використовують таблиці для представлення даних та їх відносин. Системи управління базами даних (СУБД), такі як MySQL, PostgreSQL, SQLite, забезпечують інструменти для створення, оновлення та управління даними.

Python приклад (SQLite):

```
import sqlite3

# Підключення до бази даних (або створення, якщо вона не існує)
conn = sqlite3.connect('example.db')

# Створення курсора для маніпуляції з базою даних
cursor = conn.cursor()

# Створення таблиці
cursor.execute('''CREATE TABLE IF NOT EXISTS students
                  (id INTEGER PRIMARY KEY, name TEXT, major TEXT)''')

# Вставка даних
cursor.execute("INSERT INTO students (name, major) VALUES ('John Doe',
'Computer Science')")

# Збереження змін
conn.commit()

# Запит для отримання даних
cursor.execute("SELECT * FROM students")

# Отримання результатів
print(cursor.fetchall())

# Закриття з'єднання
conn.close()
```

2. XML

Опис: XML (Extensible Markup Language) — це мова розмітки, яка використовується для представлення даних в ієрархічному форматі. XML широко використовується для обміну даними між різними системами і платформами.

Python приклад (ElementTree):

```
import xml.etree.ElementTree as ET

# Створення структури XML
data = '''<students>
    <student id="1">
        <name>John Doe</name>
        <major>Computer Science</major>
    </student>
    <student id="2">
        <name>Jane Smith</name>
        <major>Data Science</major>
    </student>
</students>'''

# Парсинг XML
root = ET.fromstring(data)

# Прохід по елементах
for student in root.findall('student'):
    name = student.find('name').text
    major = student.find('major').text
    print(f'Student name: {name}, Major: {major}')
```

3. JSON

Опис: JSON (JavaScript Object Notation) — це легкий формат обміну даними. Він легкий для людини для читання і написання, і легкий для машин для генерації та розбору. JSON використовується в багатьох програмних інтерфейсах API та конфігураціях.

Python приклад:

```
import json

# Створення JSON даних
students_json = '''
{
    "students": [
        {"id": 1, "name": "John Doe", "major": "Computer Science"},
        {"id": 2, "name": "Jane Smith", "major": "Data Science"}
    ]
}
'''
```

```
# Парсинг JSON
students_data = json.loads(students_json)

# Робота з JSON даними
for student in students_data['students']:
    print(f"

Student ID: {student['id']}, Name: {student['name']}, Major:
{student['major']}")
```

4. Індекси

Опис: Індекси в базах даних використовуються для прискорення пошуку даних. Вони працюють подібно до індексу в книзі і дозволяють швидко знаходити записи без необхідності перегляду всієї таблиці.

Python приклад (SQLite):

```
# ... Використовуємо попередньо створений з'єднання з SQLite ...

# Створення індексу для колонки 'major'
cursor.execute("CREATE INDEX IF NOT EXISTS idx_major ON students
(major)")

# Проведення запиту з використанням індексу
cursor.execute("SELECT * FROM students WHERE major = 'Computer
Science'")

# Отримання та вивід результатів
print(cursor.fetchall())

# ... Закриття з'єднання ...
```

Кожен з цих прикладів можна використовувати в якості демонстрації структур даних і методів їх обробки. Значення структуризації даних полягає в тому, що вона дозволяє впорядковувати та швидко доступати до великих об'ємів інформації, що є критично важливим для аналітичних систем.

Значення структуризації даних для ефективного пошуку

Контент для обговорення:

Структуризація даних відіграє ключову роль у ефективності пошукових операцій. Ось кілька пунктів, що підкреслюють її значення:

1. **Індексація:** Структурування даних дозволяє індексувати інформацію, що суттєво зменшує час пошуку, подібно до використання індексу в книзі.

2. **Швидкий доступ:** Дані, впорядковані у вигляді таблиць, дерев, хеш-таблиць тощо, забезпечують швидкий доступ до записів.
3. **Керування великими обсягами даних:** Структуризація допомагає управляти великими наборами даних, розподіляючи їх в структуровані блоки.
4. **Сортування та фільтрація:** Впорядковані дані можна ефективно сортувати та фільтрувати, що полегшує пошук.
5. **Оптимізація запитів:** Оптимізовані структури даних, такі як індексовані колонки в базах даних, сприяють швидшому виконанню запитів.

Python код для демонстрації:

Ми можемо продемонструвати ефективність пошуку за допомогою індексованих структур даних на прикладі Python, використовуючи список і словник для порівняння часу доступу.

```python
import time

# Створення великого списку
large_list = list(range(1000000))

# Створення словника з тими ж значеннями, де ключі - це елементи
large_dict = {i: True for i in range(1000000)}

# Пошук елемента у списку
start_time = time.time()
'999999' in large_list
end_time = time.time()
list_search_time = end_time - start_time
print(f"Час пошуку в списку: {list_search_time:.6f} секунд")

# Пошук ключа у словнику (який є хеш-таблицею)
start_time = time.time()
'999999' in large_dict
end_time = time.time()
dict_search_time = end_time - start_time
print(f"Час пошуку в словнику: {dict_search_time:.6f} секунд")

# Порівняння часу
if list_search_time > dict_search_time:
    print("Пошук в структурованому словнику значно швидший.")
else:
    print("Пошук в списку виявився швидшим, що незвично.")
```

Цей код демонструє, як структуризація даних у формі хеш-таблиці (як у Python словник) може істотно прискорити пошук у порівнянні зі списком, де пошук відбувається послідовно.

У випадку словника, час доступу до елемента в середньому є сталим (O(1)), тоді як у списку він

Хеш-таблиця — це структура даних, яка використовує хеш-функцію для відображення ідентифікуючих значень, званих ключами, на місця в масиві, званому хеш-таблицею. Основна ідея полягає в тому, що хеш-функція приймає вхідний ключ і перетворює його у величину, яка використовується як індекс у хеш-таблиці, де зберігається значення.

Ключові концепції хеш-таблиці:

1. **Хеш-функція:** Використовується для обчислення індексу в хеш-таблиці на основі ключа. Ідеальна хеш-функція є швидкою, рівномірно розподіляє ключі та мінімізує кількість колізій.
2. **Колізії:** Виникають, коли різні ключі дають однаковий хеш. Існує кілька стратегій вирішення колізій, найпопулярніші з яких — ланцюжкове зчеплення та відкрите адресування.
3. **Ланцюжкове зчеплення:** При цьому методі кожна клітина хеш-таблиці містить посилання на список (або ланцюжок) всіх елементів, що мають той самий хеш. При вставці нового елемента, якщо виникає колізія, він додається до списку.
4. **Відкрите адресування:** Коли виникає колізія, цей метод шукає наступну вільну клітину в хеш-таблиці для зберігання елемента, використовуючи стратегію, таку як лінійне зондування, квадратичне зондування чи подвійне хешування.
5. **Час доступу:** В ідеальному випадку (без колізій) час доступу до елемента в хеш-таблиці становить O(1) як для пошуку, так і для вставки. Наявність колізій може збільшити цей час, але хороша хеш-функція та достатній розмір таблиці можуть мінімізувати їх вплив.
6. **Масштабованість:** Хеш-таблиці добре масштабуються з великими обсягами даних, оскільки вони дозволяють швидко доступатися до елементів незалежно від розміру даних.

Python приклад із словником (хеш-таблицею):

```
# Створення хеш-таблиці за допомогою словника в Python
hash_table = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

# Вставка нового елемента (O(1) час доступу)
hash_table['key4'] = 'value4'

# Пошук елемента (O(1) час доступу)
value

 = hash_table.get('key2', 'Не знайдено')
print(f"Значення за ключем 'key2': {value}")
```

```
# Обробка колізії відбувається автоматично в словнику Python
```

Цей код демонструє базову операцію з хеш-таблицею в Python. Хеш-таблиці (символізовані тут словниками) є дуже ефективними для швидких пошукових операцій і часто використовуються в програмуванні для оптимізації пошуку та доступу до даних.

Перелік та класифікація алгоритмів пошуку

Контент для обговорення:

Алгоритми пошуку можуть бути класифіковані на основі різних критеріїв, включно з типами даних, над якими вони оперують, та їх ефективністю. Ось деякі з основних алгоритмів пошуку:

1. **Лінійний пошук:** Перебирає кожен елемент послідовно до знаходження потрібного.
2. **Бінарний пошук:** Працює на впорядкованих масивах, діливши область пошуку навпіл з кожним кроком.
3. **Інтерполяційний пошук:** Схожий на бінарний пошук, але вибирає точки для порівняння на основі розподілення значень у масиві.
4. **Пошук за допомогою хеш-таблиці:** Використовує хеш-функцію для прямого доступу до елемента.
5. **Пошук у глибину (DFS):** Алгоритм обходу графа, який просувається вперед до найглибшого вузла перед тим, як змінити напрямок.
6. **Пошук у ширину (BFS):** Алгоритм обходу графа, який проходить через всі сусідні вузли перед тим, як перейти на наступний рівень.
7. **Евристичний пошук (наприклад, A*):** Використовує оцінки вартості для ефективного знаходження найкоротшого шляху у графі.

Порівняння алгоритмів з точки зору ефективності та придатності

Лінійний пошук:

- Ефективність: O(n)
- Придатність: Корисний для невеликих або невпорядкованих наборів даних.

Бінарний пошук:

- Ефективність: O(log n)
- Придатність: Найкращий для великих впорядкованих наборів даних.

Інтерполяційний пошук:

- Ефективність: O(log log n) в оптимальних умовах
- Придатність: Ефективний для великих впорядкованих наборів даних з рівномірним розподілом.

Пошук за допомогою хеш-таблиці:

- Ефективність: O(1) в середньому
- Придатність: Швидкий для пошуку за ключем; вимагає достатнього простору пам'яті.

Пошук у глибину (DFS):

- Ефективність: O(V + E) для графів
- Придатність: Для розв'язання задач, що потребують перевірки всіх можливостей, наприклад, лабіринти.

**Пошук у ш

ирину (BFS):**

- Ефективність: O(V + E) для графів
- Придатність: Знаходження найкоротшого шляху в неважких графах.

Евристичний пошук (A*):

- Ефективність: Залежить від евристики; часто набагато швидше ніж BFS або DFS
- Придатність: Для пошуку найкоротшого шляху у графах, особливо з великою кількістю вершин.

Демонстрація коду на прикладі бінарного пошуку:

```
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2

        # Якщо елемент присутній посередині саме тут
        if arr[mid] == x:
            return mid

        # Якщо елемент менший за середній, він може бути лише у лівій
частині
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        # Інакше елемент може бути лише у правій частині
        else:
            return binary_search(arr, mid + 1, high, x)

    else:
        # Елемент відсутній
        return -1
```

```
# Тестування алгоритму
arr = [2, 3, 4, 10, 40]
x = 10

# Функція поверне індекс 'x', якщо він присутній в 'arr'
result = binary_search(arr, 0, len(arr)-1, x)

if result != -1:
    print(f"Елемент знайдений на індексі {result}")
else:
    print("Елемент у масиві не знайдений")
```

Цей код демонструє алгоритм бінарного пошуку в Python, який є ефективним способом пошуку елемента в сортованому списку.

Інтерполяційний пошук - це алгоритм пошуку, який працює подібно до бінарного пошуку, але в якому вибір середнього індексу відбувається на основі оцінки, де шуканий ключ може знаходитися з урахуванням розподілу значень у масиві. Цей метод найефективніший для впорядкованих масивів з рівномірним розподілом значень.

Ось приклад реалізації інтерполяційного пошуку на Python:

```
def interpolation_search(sorted_list, to_find):
    low = 0
    high = len(sorted_list) - 1

    while low <= high and to_find >= sorted_list[low] and to_find <=
sorted_list[high]:
        if low == high:
            if sorted_list[low] == to_find:
                return low
            return -1

        # Визначення позиції для порівняння
        pos = low + ((to_find - sorted_list[low]) * (high - low)) //
(sorted_list[high] - sorted_list[low])

        # Порівняння елемента за знайденою позицією
        if sorted_list[pos] == to_find:
            return pos

        # Якщо шуканий елемент більший, значить він знаходиться вправо
        if sorted_list[pos] < to_find:
            low = pos + 1
        # Якщо шуканий елемент менший, значить він знаходиться вліво
        else:
            high = pos - 1

    return -1
```

```
# Приклад використання інтерполяційного пошуку
sorted_list = [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42,
47]
to_find = 18

index = interpolation_search(sorted_list, to_find)

if index != -1:
    print(f"Елемент знайдений на позиції {index}")
else:
    print("Елемент у масиві не знайдений")
```

Інтерполяційний пошук покладається на те, що він може здогадатися, де у впорядкованому масиві може бути шуканий елемент, на основі його значення та значень на краях пошукової області. Це робить інтерполяційний пошук надзвичайно швидким для великих наборів даних з рівномірним розподілом, але його ефективність суттєво падає, якщо значення розподілені нерівномірно.

Для демонстрації реального прикладу структури даних у системі і маніпуляцій з даними можемо використати простий клас у Python, який імітує просту базу даних з використанням словника для зберігання об'єктів. У нашому випадку, це може бути проста система управління інформацією про студентів.

Клас `StudentDatabase`

```
class StudentDatabase:
    def __init__(self):
        self.students = {}

    def add_student(self, student_id, student_data):
        if student_id in self.students:
            print("Студент з таким ID вже існує.")
        else:
            self.students[student_id] = student_data
            print("Студента успішно додано.")

    def remove_student(self, student_id):
        if student_id in self.students:
            del self.students[student_id]
            print("Студента успішно видалено.")
        else:
            print("Студента з таким ID не знайдено.")

    def update_student(self, student_id, updated_data):
        if student_id in self.students:
            self.students[student_id].update(updated_data)
            print("Інформацію про студента оновлено.")
        else:
            print("Студента з таким ID не знайдено.")
```

```
    def display_students(self):
        for student_id, student_data in self.students.items():
            print(f"ID студента: {student_id}, Інформація:
{student_data}")
```

Демонстрація використання

```
# Створення бази даних студентів
db = StudentDatabase()

# Додавання студентів
db.add_student("A001", {"ім'я": "Анна", "вік": 20, "факультет":
"Фізика"})
db.add_student("A002", {"ім'я": "Богдан", "вік": 21, "факультет":
"Математика"})

# Виведення списку студентів
db.display_students()

# Оновлення інформації про студента
db.update_student("A001", {"вік": 21})

# Видалення студента
db.remove_student("A002")

# Виведення оновленого списку студентів
db.display_students()
```

Цей код створює систему, де ми можемо додавати, видаляти, оновлювати і переглядати записи про студентів. В якості основи для зберігання даних використовується словник, що дозволяє швидко знаходити студента за його ID і ефективно виконувати всі необхідні маніпуляції.

Програмування базових алгоритмів пошуку

Лінійний пошук:

Лінійний або послідовний пошук - це метод для знаходження конкретного значення в списку, який перевіряє кожен елемент списку до тих пір, поки не знайде потрібний елемент або не перевірить усі елементи.

```
def linear_search(data, target):
    for index, value in enumerate(data):
        if value == target:
            return index
    return -1

# Демонстрація лінійного пошуку
data = [5, 3, 7, 1, 9, 8]
```

```
target = 7
result = linear_search(data, target)
if result != -1:
    print(f'Елемент знайдений на позиції {result}')
else:
    print('Елемент не знайдений')
```

Двійковий пошук:

Двійковий пошук - це значно швидший алгоритм пошуку, який працює на вже впорядкованих масивах, розділяючи масив на дві половини та визначаючи, в якій з них може знаходитися елемент.

```
def binary_search(data, target):
    low = 0
    high = len(data) - 1
    while low <= high:
        mid = (low + high) // 2
        if data[mid] == target:
            return mid
        elif data[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

# Демонстрація двійкового пошуку
sorted_data = sorted(data) # Двійковий пошук вимагає сортованих даних
result = binary_search(sorted_data, target)
if result != -1:
    print(f'Елемент знайдений на позиції {result}')
else:
    print('Елемент не знайдений')
```

Використання готових бібліотек для пошуку в реальних даних

Python має кілька потужних бібліотек для роботи з реальними даними, наприклад `pandas`, який є відмінним інструментом для маніпулювання та аналізу даних.

Пошук в DataFrame з використанням `pandas`:

```
import pandas as pd

# Створення DataFrame
data = {
    'id': [1, 2, 3, 4, 5],
    'name': ['Anna', 'Bob', 'Charlie', 'Diana', 'Edward'],
    'age': [23, 35, 45, 36, 50]
}
```

```
df = pd.DataFrame(data)

# Пошук за іменем
search_for = 'Charlie'
result = df[df['name'] == search_for]

# Виведення результатів
if not result.empty:
    print(result)
else:
    print('Ім’я не знайдено')
```

Цей приклад показує, як можна використовувати `pandas` для швидкого і зручного пошуку в структурованих даних. `DataFrame` від `pandas` дозволяє легко фільтрувати рядки за певними критеріями.

Регулярні вирази (regex) використовуються для пошуку та маніпуляції текстом за допомогою заздалегідь визначених шаблонів. У Python для роботи з регулярними виразами є вбудований модуль `re`.

Контент для вивчення регулярних виразів:

- **Основи регулярних виразів:** Поняття метасимволів, ескейпінгу, квантифікаторів, груп і діапазонів.
- **Методи модуля `re`:**
  - `re.search`: Пошук першого збігу для шаблону.
  - `re.match`: Перевіряє, чи починається рядок з заданого шаблону.
  - `re.findall`: Знаходить усі збіги з шаблоном у рядку.
  - `re.sub`: Замінює збіги на заданий рядок.
- **Компіляція регулярних виразів:** Для багаторазового використання регулярного виразу його можна скомпілювати і використовувати багаторазово.
- **Флаги:** Наприклад, `re.IGNORECASE` для пошуку без врахування регістру, `re.MULTILINE` для роботи з багаторядковими рядками тощо.

Приклад коду:

Давайте розглянемо приклади регулярних виразів на Python.

```
import re

# Текст для пошуку
text = "Цей текст містить числа 123 та спеціальні символи $%^."

# Шукаємо всі слова
words = re.findall(r'\b\w+\b', text)
print("Слова:", words)

# Шукаємо всі числа
numbers = re.findall(r'\b\d+\b', text)
```

```
print("Числа:", numbers)

# Знаходимо перше входження цифр
first_number = re.search(r'\d+', text)
if first_number:
    print("Перше знайдене число:", first_number.group())

# Заміна спеціальних символів на пробіли
sanitized_text = re.sub(r'[^\w\s]', ' ', text)
print("Текст після видалення спецсимволів:", sanitized_text)

# Пошук слова з великої літери
capitalized_words = re.findall(r'\b[A-ZA-ЯЇІЄҐ]\w+', text)
print("Слова, що починаються з великої літери:", capitalized_words)

# Перевірка наявності дати у форматі дд.мм.рррр
date = "Текст з датою 12.11.2022 в середині."
date_match = re.search(r'\b\d{2}\.\d{2}\.\d{4}\b', date)
if date_match:
    print("Дата знайдена:", date_match.group())
```

Ці приклади ілюструють базове використання регулярних виразів у Python для різних пошукових і заміняючих завдань. З їх допомогою можна виявляти патерни у тексті, екстрактувати інформацію і трансформувати текстові дані.

Регулярні вирази (regular expressions, regex) є потужним інструментом для роботи з текстом. Ось декілька прикладів регулярних виразів та їхні описи, що можуть бути корисними для аналітичних систем:

1. **Валідація Email Адреси:**

```
\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b
```

Цей вираз перевіряє, чи рядок виглядає як email адреса.

1. **Пошук IP-Адреси:**

```
\b(?:\d{1,3}\.){3}\d{1,3}\b
```

Цей вираз знаходить рядки, які відповідають звичайному вигляду IPv4 адреси.

1. **Знаходження Дати у Форматі ДД.ММ.РРРР:**

```
\b\d{1,2}\.\d{1,2}\.\d{4}\b
```

Цей вираз знаходить дати у конкретному форматі.

1. **Пошук Годин у Форматі ГГ:ХХ (24-годинний формат):**

```
\b([01]?[0-9]|2[0-3]):[0-5][0-9]\b
```

Цей вираз виявляє час, що вказаний у 24-годинному форматі.

1. **Виявлення Слів, Що Починаються З Великої Літери:**

```
\b[A-ZA-ЯЇІЄҐ][a-za-яїієґ]*\b
```

Цей вираз відшукує слова, що починаються з великої літери.

1. **Пошук Лише Цифр:**

```
\b\d+\b
```

Цей вираз відшукує послідовності, які складаються тільки з цифр.

1. **Валідація Номера Телефону (простий варіант):**

```
\b\d{3}[-.]?\d{3}[-.]?\d{4}\b
```

Цей вираз перевіряє, чи рядок виглядає як номер телефону у форматі 123-456-7890 або 123.456.7890 або 1234567890.

1. **Пошук HEX Кольорів:**

```
#([a-fA-F0-9]{6}|[a-fA-F0-9]{3})\b
```

Цей вираз знаходить HEX-коди кольорів, які використовуються у CSS і HTML.

Для застосування цих виразів у Python, використовуйте модуль `re`, як показано нижче:

```python
import re

text = "Мій email це example@mail.com і мій IP це 192.168.1.1."

# Знайти всі email адреси
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]
{2,}\b', text)
print(emails)

# Знайти всі IP адреси
ips = re.findall(r'\b(?:\d{1,3}\.){3}\d{1,3}\b', text)
print(ips)
```

Цей код відшукає всі входження, які відповідають регуля

Регулярні вирази (regex) дозволяють описати складні шаблони для пошуку і маніпуляції рядками. Ось кілька прикладів регулярних виразів та їх застосування:

1. **Пошук електронної адреси:**

```
\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b
```

1. **Перевірка сильного пароля (мінімум 8 символів, з великою та малою літерами, числами та спеціальними символами):**

```
^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$
```

1. **Виділення дати у форматі ДД/ММ/РРРР:**

```
\b(0?[1-9]|[12][0-9]|3[01])/(0?[1-9]|1[012])/\d{4}\b
```

1. **Пошук тегів HTML:**

```
<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)
```

1. **Валідація номеру телефону (формати включають (123) 456-7890, 123-456-7890, 123.456.7890, +31636363634, 075-63546725):**

```
(?:\+?\d{1,3}[\s.-]?)?(?:\(\d{3}\)|\d{3})[\s.-]?\d{3}[\s.-]?\d{4}
```

1. **Заміна кінця рядка/речення на період:**

```
[.!?]\s
```

1. **Виділення слова з урахуванням великої літери:**

```
\b[A-ZA-ЯЇІЄҐ][a-za-яїієґ]+
```

1. **Пошук IP-адрес (простий варіант):**

```
\b(?:\d{1,3}\.){3}\d{1,3}\b
```

1. **Пошук HEX-кольорів (наприклад, #a3c113):**

```
#([a-fA-F0-9]{6}|[a-fA-F0-9]{3})
```

1. **Вилучення всіх слів, що містять великі літери:**

```
\b[A-ZA-ЯЇІЄҐ]+\b
```

Ось як ці регулярні вирази можуть бути використані у Python з використанням модуля `re`:

```python
import re

text = "Це є email@example.com серед тексту. Ще один:
another.email@domain.com. Номер телефону: +380(99)123-4567."

# Пошук всіх електронних адрес
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]
{2,}\b', text)
print("Електронні адреси:", emails)

# Пошук номерів телефонів
phones = re.findall(r'(?:\+?\d{1,3}[\s.-]?)?(?:\(\d{3}\)|\d{3})[\
s.-]?\d{3}[\s.-]?\d{4}', text)
print("Номери телефонів:", phones)
```

Використання регулярних виразів дозволяє ефективно обробляти текстові

дані, вилучати інформацію та здійснювати складні заміни.

Цей регулярний вираз використовується для визначення номерів телефонів, що можуть мати різні формати. Ось детальний опис кожної його частини:

1. `(?:\+?\d{1,3}[\s.-]?)?`
   - `?:` - це некаптурна група, що означає, що вона не захоплює дані для пізнішого використання.
   - `\+?` - відповідає знаку плюсу (+), який може бути в начале міжнародного номера телефону. Знак запитання означає, що знак плюсу є необов'язковим.
   - `\d{1,3}` - відповідає від однієї до трьох цифр, які можуть представляти міжнародний код країни.
   - `[\s.-]?` - відповідає одному пропуску, крапці, або тире, які можуть використовуватися як роздільники в міжнародних номерах. Це також необов'язковий символ.
2. `(?:\(\d{3}\)|\d{3})`
   - Ця частина виразу шукає або три цифри в дужках (наприклад, `(123)`), або просто три цифри без дужок (наприклад, `123`). Це може представляти код області або першу частину місцевого номера.
3. `[\s.-]?`
   - Це той самий необов'язковий роздільник, що був описаний вище.
4. `\d{3}`
   - Відповідає наступним трьом цифрам номера телефону.
5. `[\s.-]?`
   - Ще один необов'язковий роздільник.
6. `\d{4}`
   - І, нарешті, останні чотири цифри номера телефону.

Весь вираз може визначати номери телефонів різних форматів, наприклад:

- `+123 456 7890`
- `123-456-7890`
- `(123) 456-7890`
- `123.456.7890`
- `1234567890`

Регулярний вираз не враховує всі можливі формати та варіації номерів телефонів, які існують у світі, але є гнучким для визначення багатьох з них. Однак, якщо потрібно точніше визначення номерів для конкретної країни або формату, вираз може бути складнішим і специфічнішим.

Сформулюємо 30 завдань для практичної роботи студентів, використовуючи Jupyter Notebook (.ipynb). Ці завдання будуть включати різні аспекти роботи зі структурами даних і алгоритмами пошуку:

1. Створіть список з 10 випадково генерованих чисел і виведіть його.
2. Виконайте лінійний пошук заданого елемента в списку з п.1.
3. Реалізуйте функцію для двійкового пошуку та перевірте її на списку з п.1.

4. Сортуйте список з п.1 за допомогою алгоритму сортування бульбашкою.
5. За допомогою модуля `random` створіть список з 20 цілих чисел та виконайте пошук мінімального і максимального елементів.
6. Реалізуйте інтерполяційний пошук і перевірте його ефективність на великих масивах.
7. Створіть двовимірний масив (матрицю) випадкових чисел розміром 5x5 та знайдіть рядок із найбільшою сумою елементів.
8. Напишіть функцію для обходу матриці з п.7 спірально, починаючи з верхнього лівого кута.
9. Використовуйте регулярні вирази для пошуку всіх дат у заданому тексті.
10. Створіть функцію, яка перевіряє, чи є заданий рядок електронною адресою.
11. Використайте регулярний вираз для розділення тексту на речення.
12. Створіть хеш-таблицю для зберігання пар 'ключ-значення' і продемонструйте додавання і видалення елементів.
13. Реалізуйте простий алгоритм хешування для розподілу слів зі списку по хеш-таблиці.
14. Порівняйте швидкість виконання лінійного пошуку та двійкового пошуку на великих масивах.
15. Використовуйте бібліотеку `pandas` для завантаження датасету у форматі CSV та здійсніть пошук по колонках.
16. Застосуйте алгоритм пошуку до даних, отриманих з веб-сервісу (наприклад, REST API).
17. Використайте бібліотеку `BeautifulSoup` для парсингу HTML сторінки та виконайте пошук всіх посилань.
18. Застосуйте `regex` для валідації номерів кредитних карток, що вводяться користувачем.
19. Створіть клас, що представляє просту структуру даних типу черга і реалізуйте базові операції.
20. Використайте `numpy` для створення великого масиву чисел і знайдіть всі прості числа за допомогою фільтрації.
21. Розробіть скрипт для зчитування JSON-файлу та виконайте пошук за конкретним ключем.
22. Порівняйте ефективність вставки, видалення та оновлення елементів у різних структурах даних (список, множина, словник).
23. Реалізуйте алгоритм A* для пошуку шляху на двовимірній карті.
24. Використовуйте `matplotlib` для візуалізації результатів пошуку або аналізу даних.
25. Продемонструйте використання стеку (стопки) для перевірки правильності послідовності дужок у математичному виразі.
26. Розробіть програму для шифрування та дешифрування тексту методом Цезаря та здійсніть пошук ключа шифрування.
27. Використовуйте регулярний вираз для пошуку та заміни імен власних у тексті.
28. Створіть скрипт для автоматизації пошуку файлів за патерном у файловій системі.
29. Виконайте пошук за допомогою регулярного виразу в лог-файлі для визначення помилок та попереджень.

30. Реалізуйте алгоритм пошуку шляху в лабіринті за допомогою алгоритму пошуку в глибину (DFS).

Кожне завдання може бути реалізовано як окрема Jupyter Notebook комірка з інструкціями та необхідними вхідними даними для студентів.

Оцінюючи складність завдань із зазначеного списку, слід враховувати кілька факторів, включаючи алгоритмічну складність, потребу в розумінні специфічних даних, вимоги до аналізу та оптимізації. Тут є мій вибір десяти завдань, які можуть бути найбільш складними для студентів:

Визначення "найскладніших" завдань може залежати від критеріїв оцінки, таких як обсяг роботи, рівень алгоритмічної складності, потреба в глибокому розумінні специфічних доменів або технологій тощо. Однак, з огляду на те, що більшість завдань зі списку стосуються алгоритмів та обробки даних у Python, можна виділити наступні завдання як потенційно найскладніші:

1. **Реалізуйте алгоритм A* для пошуку шляху на двовимірній карті.**
   - Вимагає глибокого розуміння алгоритму пошуку шляху та роботи з графами.
2. **Напишіть функцію, яка реалізує інтерполяційний пошук, та продемонструйте його ефективність порівняно з бінарним пошуком.**
   - Вимагає знання складних алгоритмів пошуку та аналізу їхньої ефективності.
3. **Створіть систему індексації текстових файлів та реалізуйте пошук за ключовими словами.**
   - Включає в себе розуміння принципів роботи пошукових систем.
4. **Реалізуйте алгоритм пошуку в глибину (DFS) та пошуку в ширину (BFS) для проходження лабіринту.**
   - Потрібно розуміння двох фундаментальних алгоритмів обходу графа.
5. **Розробіть програму для шифрування та дешифрування тексту методом Цезаря та здійсніть пошук ключа шифрування.**
   - Потребує знань у сфері криптографії та алгоритмів розшифровки.
6. **Розробіть скрипт для зчитування JSON-файлу та виконайте пошук за конкретним ключем.**
   - Включає роботу з JSON-структурами та розуміння рекурсивного пошуку.
7. **Продемонструйте використання стеку (стопки) для перевірки правильності послідовності дужок у математичному виразі.**
   - Вимагає розуміння структур даних та алгоритмів для обробки рядків.
8. **Використовуйте `matplotlib` для візуалізації результатів пошуку або аналізу даних.**
   - Вимагає знань з візуалізації даних та бібліотеки matplotlib.
9. **Реалізуйте алгоритм A* для пошуку шляху на двовимірній карті з використання

м вагових коефіцієнтів.**

- Це розширення першого завдання з додаванням складності через вагові коефіцієнти.

1. **Створіть систему рекомендацій на основі фільтрації змісту, використовуючи набір даних фільмів або книг.**

– Потрібне розуміння систем рекомендацій та машинного навчання.

Завдання можуть бути різною складністю в залежності від контексту курсу та попереднього досвіду студентів. Наприклад, для студентів, які не мають досвіду з певними бібліотеками або алгоритмами, навіть менш складні задачі можуть здатися важкими.