Ідеї та приклади вирішення задач

Об’єднання N відсортованих файлів

Задача: K веб-серверів породжують M файлів з логами кожен. Файли логів відсортовані за часом, вони дуже великого розміру.

Для аналізу роботи всієї системи з K серверів, ми хочемо переглянути дані одночасно з усіх K*M файлів, дані впорядковані за часом.

Для цього нам варто об’єднати всі лог файли в один, також відсортований за часом подій у логах.

Об’єднання N відсортованих файлів

Розв’язок №1

Завантажити логи з усіх файлів в оперативну пам’ять, склеїти в один масив даних, відсортувати.

```
k1 = [...]
k2 = [...]
...
kN = [...]

k = sorted(k1 + k2 + ... + kN)
```

Об’єднання N відсортованих файлів

Розв’язок №1

Переваги: простота

Недоліки:

- Так як лог файли зазвичай займають доволі багато місця на жорсткому диску, малоймовірно, що на нашому сервері буде достатньо оперативної пам’яті для завантаження всіх файлів
- Явно не перевикористовуємо властивість відсортованості усіх файлів

Об’єднання N відсортованих файлів

Розв’язок №2

Використаємо інваріанту, яка зустрічається в кожному файлі: кожен файл власне сам відсортований по часу.

Читаємо по рядку з кожного файлу, складаємо значення в масив. Знаходимо рядок з мінімальним часом серед зчитаних рядків, записуємо у результуючий файл. Запам’ятовуємо, з якого файлу було записано рядок.

З цього файлу в наш масив рядків зчитуємо наступний рядок, знову знаходимо мінімум, повторюємо.

Пошук мінімуму у масиві — O(N)

Об’єднання N відсортованих файлів

Розв’язок №2

```
ks = [
   [...], # array/file 1
   [...], # array/file 2
   ...,
   [...] # array/file N
]
k_iterators = [0, 0, ..., 0]
k = []

min_array = [ks[0][0], ks[1][0], ..., ks[N][0]]

while true("any elements in any of N arrays left"):
   # знаходимо індекс мінімального елементу в масиві
   mi = min_index(min_array)
   k.append(min_array)

   k_iterators[mi] += 1

   min_array[mi] = ks[k_iterators[mi]]
```

Об’єднання N відсортованих файлів

Розв’язок №3

K-way merge з купою

Заміняємо масив зчитаних рядків-логів на мінімальну купу, виконуємо видалення мінімального та вставку наступного за O(log(N))

Об’єднання N відсортованих файлів

Розв’язок №3

```
ks = [
   [...], # array/file 1
   [...], # array/file 2
   ...,
   [...] # array/file N
]
k_iterators = [0, 0, ..., 0]
k = []

min_heap = Heap([
   (ks[0][0], 0),
   (ks[1][0], 1),
   ...,
   (ks[N-1][0], N - 1)
])

while ["any elements in any of N arrays left"]:
   # знаходимо мінімальне значення в купі разом з індексом його масиву
   min_value, min_array_number = min_heap.delete_min()
   # рухаємось далі у масиві, з якого ми забрали мінімальне значення
   k_iterators[min_array_number] += 1

   # вставляємо в купу наступне значення з того ж масиву
   min_heap.insert(
       ks[min_array_number][k_iterators[min_array_number]],
       min_array_number
   )

   k.append(min_value)
```

Максимальний добуток бітонічного підряду розміру 3

Задано з натуральних цілих чисел розміром N, завдання полягає в тому, щоб знайти максимальний добуток бітонічного підряду розміру 3.

Бітонічний підряд: підряд, в якому елементи спочатку зростають, а потім спадають порядку зменшення. У елементах підряду можна задати наступний порядок: **arr[i] < arr[j]> arr[k]** для **i < j <k**, де **i, j, k** - індекс даного масиву.

Якщо таких трійок у масиві нема, потрібно вивести -1.

Максимальний добуток бітонічного підряду розміру 3

На вхід: arr = [1, 8, 3, 7, 5, 6, 7]

Вихід: 126

Пояснення:

Бітонічні підряди розміру 3:

{1, 8, 3}, {1, 8, 7}, {1, 8, 5}, {1, 8, 6}, {1, 7, 6}, {3, 7, 6}, {1, 7, 5}, {3, 7, 5}.

Максимальним добутком бітонічного підряду є 3*7*6 = 126

Максимальний добуток бітонічного підряду розміру 3

Ідея вирішення: **повний перебір.**

Потрібно перебрати можливі бітонічні трійки у трьох циклах по **i, j та k** та залишаті такі трійки, у яких **a[i] < a[j]** та **a[j] > a[k]**.

У цьому ж циклі перемножити їх, якщо добуток за попередній знайдений максимум — переписати максимум поточним значенням.

Максимальний добуток бітонічного підряду розміру 3

Псевдокод:

```
def maxProduct(arr, n):
  # проініціалізуємо відповідь -1, якщо бітонічних трійок у масиві нема
  ans = -1;

  # Вкладені цикли, щоб обирати трійки трійки з масиву
  for i in range(n - 2):
     for j in range(i + 1, n - 1):
        for k in range(j + 1, n):
           # перевіряємо умову бітонічності
           if arr[i] < arr[j] and arr[j] > arr[k]:
              ans = max(ans, arr[i] * arr[j] * arr[k])
  return ans
```

Максимальний добуток бітонічного підряду розміру 3

Асимптотична оцінка: $O(N^3)$.

Чи можемо краще?

Максимальний добуток бітонічного підряду розміру 3

Експлуатуємо ідею: для кожного елементу **arr[i]** нам потрібно швидко знаходити ***максимальний*** елемент зліва та справа, який ***менший*** за **arr[i]**.

Для швидкого пошуку максимального елементу, меншого за **arr[i]** використаємо AVL-множину, у якої реалізований метод lower_bound.

Максимальний добуток бітонічного підряду розміру 3

Псевдокод:

```
def maxProduct(arr, n):
    avlSet = AVLSet()

    # для кожного елементу arr[i] знайти найбільший елемент зліва від нього
    max_left_items = []
    for i in range(n):
        avlSet.add(arr[i])
        max_left_items[i] = avlSet.lower_bound(arr[i])

    avlSet.clear()

    # для кожного елементу arr[i] знайти найбільший елемент справа від нього
    max_right_items = []
    for i in reversed(range(n)):
        avlSet.add(arr[i])
        max_right_items[i] = avlSet.lower_bound(arr[i])

    # проініціалізуємо відповідь -1, якщо бітонічних трійок у масиві нема
    ans = -1;
    for i in range(n):
        if max_left_items[i] is not None and max_right_items[i] is not None:
            ans = max(ans, max_left_items[i] * max_right_items[i] * arr[i])

    return ans
```

Максимальний добуток бітонічного підряду розміру 3

Асимптотична оцінка

На пошук супремуму зліва витрачаємо **O(N*logN)**

На пошук супремуму справа витрачаємо **O(N*logN)**

На пошук максимального добутку після цього **O(N)**

В сумі, **O(N*logN) + O(N*logN) + O(N) = O(N*logN)**

Фондовий ринок

Вартість акцій на кожен день подається в масиві, потрібно знйти максимальний прибуток, який можна отримати, купуючи та продаючи акції в цей період.

Наприклад, якщо даний масив - **[100, 180, 260, 310, 40, 535, 695]**, максимальний прибуток можна отримати, купивши акції в день 0, продавши в день 3, знову купивши в день 4 і продавши в день 6.

(310 - 100) + (695 - 40) = 865

Якщо заданий масив цін на акції відсортований у порядку спадання, то прибуток взагалі не отримати неможливо.

Фондовий ринок

Роз’язок №1

Розглянемо всі можливі варіанти днів покупок та днів продажів, серед яких виконується умова “ціна в день покупки < ціна в день продажу”, і будемо запам’ятовувати максимальний прибуток.

Цей варіант розв’язку вкрай неефективний, адже вимагає перебір всіх пар днів, і кількість таких переборів росте експоненційно.

Фондовий ринок

Роз’язок №1

```
def max_profit(prices, start, finish):
    if finish <= start:
        return 0

    profit = 0

    for i in range(start, finish):
        for j in range(i + 1, finish + 1):
            if prices[i] < prices[j]:
                current_profit = prices[j] - prices[i] + \
                        max_profit(start, i - 1) +\
                        max_profit(j + 1, finish)

                profit = max(current_profit, profit)

    return profit
```

Фондовий ринок

Роз’язок №2

Ефективність попереднього розв’язку нас не влаштовує, тому потрібно розглянути інші деталі умови задачі.

Якщо розглянути приклад [100, 180, 260, 310, 40, 535, 695], стає зрозуміло, що нам не потрібно розглядати варіанти продажу, допоки акції не починають падати. Якщо купити акції по 100, потрібно пропустити всі варіанти продажі до того моменту, поки ціна на них не почне падати.

Продовжувати такий цикл, допоки ми не дістанемось кінця масиву цін.

Фондовий ринок

Роз’язок №2

```
def max_profit(prices):
    if len(prices) < 2:
        return 0

    profit = 0
    n = len(prices)
    i = 0
    while i < (n - 1):
        while i < (n - 1) and prices[i + 1] <= prices[i]:
            i += 1

        if (i == n - 1):
            break

        buy = i
        i += 1

        while ((i < n) and (prices[i] >= prices[i - 1])):
            i += 1

        sell = i - 1
        profit += prices[buy] - prices[sell]

    return profit
```

Фондовий ринок

Роз’язок №2

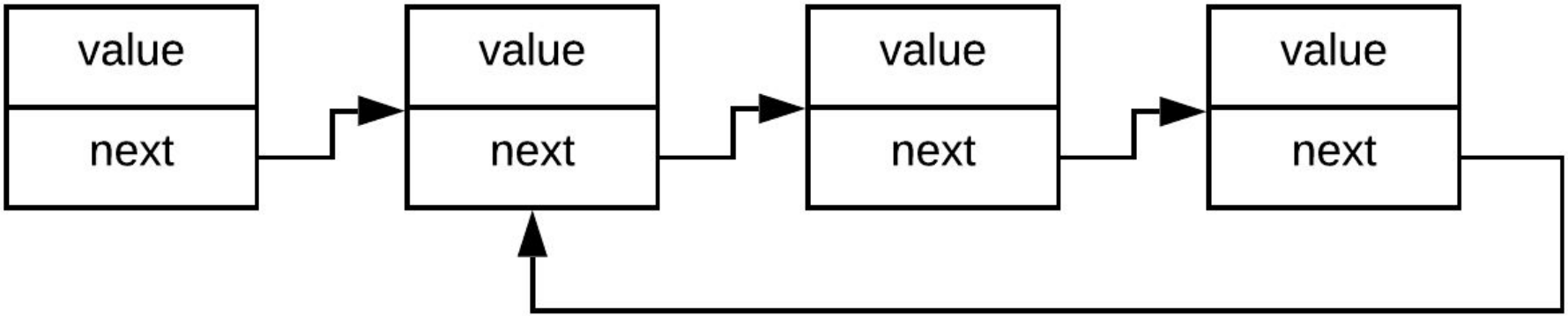
Складність по часу часу: зовнішній цикл while працює, поки і не дійде до позиції n-1. Два внутрішні цикли while лише збільшують значення і та ніколи його не зменшують — отже, загальна складність по часу становить O(N)

1. Пошук циклів

Задача: визначити, чи є цикли у заданому зв’язному списку.

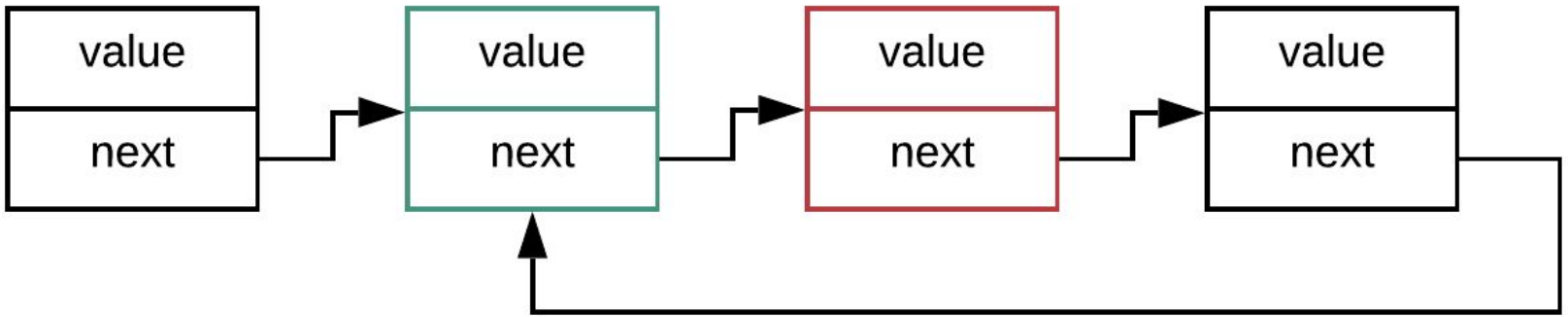
На вхід подається об’єкт класу `Node` – вузол зв’язного списку, який містить значення `value`, а такой вказівник (`next`) на наступний вузол.

Зв’язний список містить цикл, якщо при виконанні переходів між вузлами вони повторюються. Наприклад:



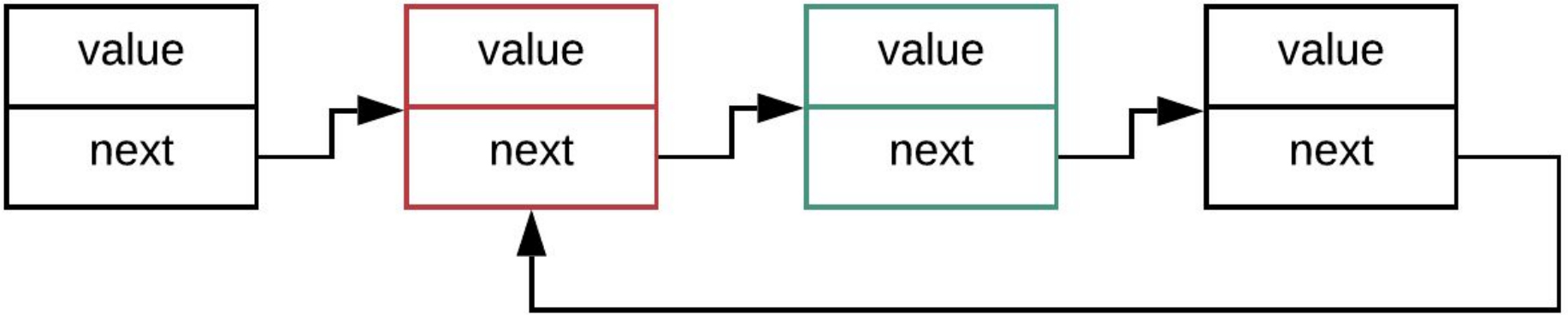
2. Пошук циклів. Алгоритм розв’язку

- Застосуємо **алгоритм Флойда**
- Ідея алгоритму полягає в тому, щоб використовувати два вказівники, які переміщуються по списку із різним кроком
- Наприклад, перший вказівник рухається з кроком 1, другий – із кроком 2
- Якщо обидва вказівники вказують на той самий вузол, то цикл знайдено
- Малюнок нижче ілюструє положення вказівників після першого кроку: зеленим – той, що має крок 1, червоним – вказівник із кроком 2:



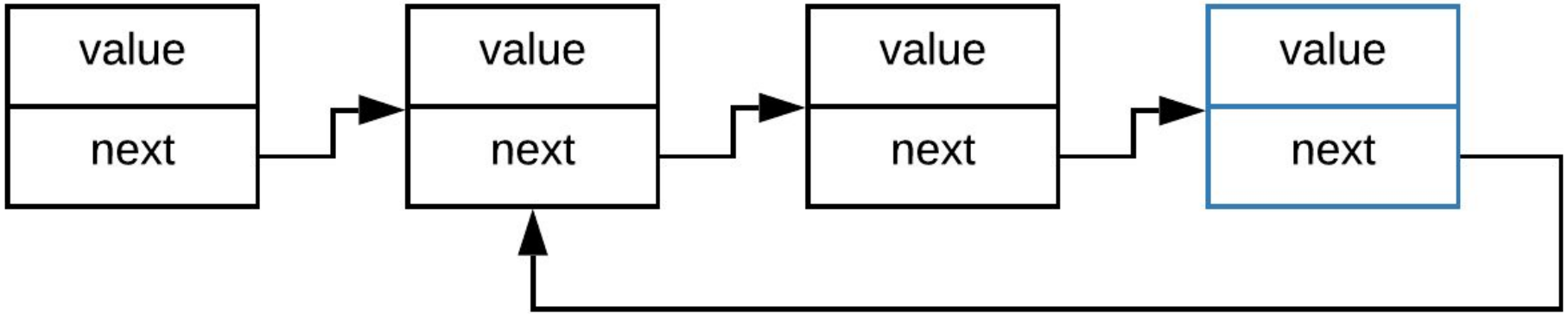
3. Пошук циклів. Алгоритм розв’язку

- Зробимо наступний крок переміщення вказівників:



4. Пошук циклів. Алгоритм розв’язку

- Після ще одного кроку обидва вказівники вказують на той самий вузол (виділено синім), а отже, цикл знайдено:



- Іншою умовою зупинки алгоритму може бути досягнення кінця списку (`next` вказує на `None`). У цьому випадку список не містить циклу

5. Пошук циклів. Реалізація алгоритму Флойда

```
def contains_cycle(start_node: Node) -> bool:
    def perform_steps(ptr1: Node, ptr2: Node) -> (Node, Node):
        ptr1 = ptr1.next
        ptr2 = ptr2.next
        if ptr2 is not None:
            ptr2 = ptr2.next
        return ptr1, ptr2

    pointer1, pointer2 = perform_steps(start_node, start_node)
    while True:
        if pointer1 is None or pointer2 is None:
            return False
        elif pointer1 is pointer2:
            return True
        else:
            pointer1, pointer2 = perform_steps(pointer1, pointer2)
```

6. Пошук циклів

- Цей підхід може застосовуватися також для виявлення циклів у графах, і загалом для виявлення функцій, які відображають певну скінченну множину в себе (оскільки у цьому випадку значення при ітеративному застосуванні функції періодично повторюватимуться)
- **Складність алгоритму:** $O(\lambda + \mu)$ по часу, де λ – довжина циклу, а μ – індекс першого елементу у циклі; $O(1)$ по пам’яті.
- Також наведений алгоритм можна модифікувати, щоб знаходити параметри λ і μ