ПОШУКОВІ АЛГОРИТМИ

Лекція 4

Дисципліна «Алгоритми та структури даних»

Поняття пошуку даних

Пошук даних (Data Search) - це процес знаходження одного або декількох елементів даних, які задовольняють певним критеріям, усередині більшого набору (масиву, списку, бази даних або іншої структури даних)

Задача пошуку відповідає на питання: **“Чи існує елемент X у множині S, і якщо так, то де він знаходиться?"**

Класифікація алгоритмів пошуку
1. За вимогою до впорядкованості даних

1.1. Пошук у невпорядкованих даних (Unsorted Search)

Ці алгоритми можуть працювати з будь-яким набором даних, незалежно від того, чи відсортовані вони

Лінійний (послідовний) пошук (Linear/Sequential Search) - найпростіший метод. Елементи перевіряються по черзі, починаючи з першого, доки не знайдеться потрібний або не буде досягнуто кінця

Складність: O(n), де n - кількість елементів

1.2. Пошук у впорядкованих даних (Sorted Search)

Ці алгоритми вимагають, щоб дані були відсортовані, але працюють значно швидше

Бінарний (двійковий) пошук (Binary Search) - порівнює цільове значення із середнім елементом масиву. Якщо значення не збігається, пошук продовжується в одній із половин (лівій чи правій)

Складність: O(logn)

Інтерполяційний Пошук (Interpolation Search) - варіант бінарного пошуку, який, ґрунтуючись на припущенні про рівномірний розподіл даних, обчислює більш імовірне місце розташування цільового елемента (аналогічно тому, як людина шукає слово у словнику)

Складність: у найгіршому випадку **O(n)**

Класифікація алгоритмів пошуку

2. За методикою виконання пошуку (для складних структур)

Ця класифікація стосується пошуку в графах, деревах та інших структурах даних

2.1. Пошук на основі хешування (Hashing)

Використовує хеш-функцію для обчислення індексу (адреси), де, ймовірно, знаходиться елемент; забезпечує надзвичайно швидкий доступ.

Пошук у хеш-таблиці (Hash Table Search) - в ідеальному випадку дозволяє знаходити елемент за константний час

Складність: У середньому O(1)

2.2. Пошук у деревах (Tree Search)

Використовується для ієрархічно структурованих даних.

Пошук у двійковому дереві пошуку (Binary Search Tree - BST) - кожен вузол має не більше двох нащадків; усі вузли лівого піддерева менші, а правого - більші

Складність: У середньому O(logn), у найгіршому випадку O(n)

2.3. Пошук у графах (Graph Search)

Застосовується для знаходження шляхів або вузлів у складних мережевих структурах

Пошук у ширину (Breadth-First Search - BFS) - обходить граф рівень за рівнем, знаходячи найкоротший шлях до цілі у кількості ребер

Пошук у глибину (Depth-First Search - DFS) - обходить граф, заходячи якомога глибше по одній гілці, перш ніж повертається назад

Повний перебір і лінійний пошук

Повний перебір (Exhaustive Search), відомий також як **груба сила (Brute-Force Search)**, є одним із фундаментальних алгоритмів пошуку даних

Повний перебір - це алгоритмічна техніка, яка перевіряє **кожен можливий елемент** у заданому просторі пошуку (наприклад, у масиві, списку або множині) для того, щоб знайти той, що відповідає певним критеріям, або для того, щоб знайти оптимальне рішення задачі

У контексті пошуку елементів у неструктурованому списку чи масиві, повний перебір **еквівалентний лінійному (послідовному) пошуку (Linear Search)**

Лінійний (послідовний) пошук

Лінійним або послідовним пошуком називається алгоритм відшукання елемента серед заданого набору шляхом послідовного перебору всіх елементів

Починаючи з першого елемента в списку, послідовно рухаємося по елементах послідовності (**у порядку зростання індексів**), до тих пір, поки або не знайдемо те, що шукаємо, або не досягнемо останнього елемента (це означає, що послідовність не містить шуканого елемента)

Приклад реалізації на Python

```
def linear_search(data_list, target):
# 1. Ітеруємо через кожен елемент списку за його індексом
    for index in range(len(data_list)):
# 2. Перевіряємо, чи поточний елемент дорівнює цільовому значенню
        if data_list[index] == target:
# 3. Якщо знайдено, повертаємо його індекс
            return index


# 4. Якщо цикл завершився, і елемент не був знайдений, повертаємо -1
    return -1
```

Розглянемо, як функція шукає число **90** у списку
[45, 88, 72, 90, 65, 50]

Лінійний пошук у найгіршому випадку виконується за лінійний час, тобто **O(n)**
У випадку, якщо шуканий елемент знаходиться на першій позиції, то лінійний пошук здійснюється за сталий час **O(1)**

Однією з класичних задач, що розв’язуються повним перебором, є задача про рюкзак

Турист зібрався у похід і вирішив відповідально підійти до вибору того, що він візьме з собою. У туриста є n речей, які він міг би взяти з собою у рюкзаку. Кожна річ важить 1 кілограм. Речі мають різну "користність" для туриста. Похід очікується досить тривалий, і турист хотів би носити рюкзак вагою не більше w кілограм. Допоможіть йому визначить максимальну сумарну "корисність" предметів у нього в рюкзаку при вазі рюкзака не більше w кілограм

Задачу можна розв’язати, повністю перебравши всі можливі розв’язки. Отже, у нас є n речей, які можна укладати в рюкзак. Для кожного предмета існує 2 варіанти: предмет або кладеться в рюкзак, або ні

Тоді, алгоритм, що використовує повний перебір всіх можливих варіантів має складність $\mathbf{O(2^n)}$

Це дозволяє його використовувати лише для невеликої кількості предметів. З ростом кількості предметів задача стає нерозв'язною даним методом за прийнятний час

Бінарний пошук

Бінарний пошук - двійковий пошук, пошук діленням навпіл; може бути застосований для **впорядкованих** послідовностей за незменшенням або незбільшенням

Незменшення: 2 2 2 2 3 3 3 5 5 5
Незбільшення: 5 5 5 3 3 3 2 2 2 2

Асимптотична складність – O(logn)
Алгоритм
1) Знаходимо індекс середнього елементу;
2) Порівнюємо значення, яке шукаємо з середнім елементом послідовності. Розглядаємо 3 можливі випадки:

- шукане значення = середньому елементу, тоді пошук завершено;
- шукане значення < середнього елементу, тоді здійснюємо пошук у лівій половині послідовності;
- шукане значення > середнього елементу, тоді здійснюємо пошук у правій половині послідовності.

3) Продовжуємо поки інтервал пошуку не перетвориться в одне число або поки не буде знайдений елемент.

Алгоритм, що знаходить випадковий елемент із співпадаючих з шуканим

Розглянемо бінарний пошук для послідовності з n елементів, впорядкованих за незменшенням: для кожної пари виконується умова A[i]<=A[i+1]

	L=0											R=n-1
	0	1	2	3	4	5	6	7	8	9	10	11
A[n]	5	12	18	24	31	38	45	50	62	77	84	91
n=12												
x=31					i = ціла частина((L+R)/2)=5							

L - Індекс початку поточного діапазону пошуку

R - Індекс кінця поточного діапазону пошуку

Середній індекс, обчислюється як ціла частина **(L+ R)/2**

X<A[i], 31<38. Цільовий елемент **менший**

Це означає, що потрібний елемент знаходиться у лівій частині (від індексу 0 до 4), тому **R=i-1=5-1=4**

L=0											
0	1	2	3	4	5	6	7	8	9	10	11
5	12	18	24	31	38	45	50	62	77	84	91

L=0				R=4
0	1	2	3	4
5	12	18	24	31

i = ціла частина((L+R)/2)=(4+0)/2=2

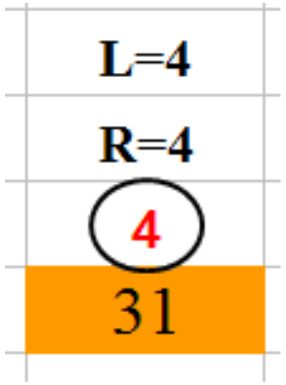
X>A[i], 31>18. Цільовий елемент **більший**. Це означає, що потрібний елемент знаходиться у правій частині (від індексу 3 до 4), тому **L=i+1=2+1=3**

L=0				
0	1	2	3	4
5	12	18	24	31

L=3	R=4
3	4
24	31

i = ціла частина((L+R)/2)= ціла частина((4+3)/2)=3

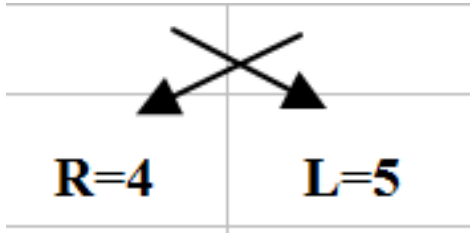
X>A[i], 31>24. Цільовий елемент **більший**. Це означає, що потрібний елемент знаходиться у правій частині (від індексу 4 до 4), тому **L=3+1=4**, **R=4**



i = ціла частина((L+R)/2)=(4+4)/2=4

X=A[i], 31=31, i=4

А якщо шукати не 31, а елемент 32? X>A[i], 32>31. Переставляємо ліву границю **L=4+1=5**, а права границя залишається **R=4.** Границі перетнулись, діапазону пошуку немає:



Елемент не знайдено

Обчислення середнього індексу

Краще робити за формулою: $i = L + \left[\frac{R-L}{2}\right]$ замість $i = \left[\frac{L+R}{2}\right]$

Формула $i = L + \left[\frac{R-L}{2}\right]$ позбавлена потенційної помилки виходу значення індексу за межі діапазону значень свого типу у випадку великих обсягів даних і використовується для запобігання **цілочисельному переповненню (integer overflow)**

Якщо **L** (нижня границя) і **R** (верхня границя) є дуже великими індексами, їхня сума може **перевищити максимальне значення**, яке може зберігати стандартний 32-розрядний або навіть 64-розрядний цілочисельний тип даних, який використовується в конкретній мові програмування

Якщо відбувається переповнення, результат L+R стає **неправильним** (зазвичай, від'ємним числом або дуже малим позитивним), що призводить до некоректного індексу і і, як наслідок, до **збою (crash)** програми або **нескінченного циклу**

Приклад реалізації на Python

```
def binary_search(data_list, target):
low = 0
    high = len(data_list) - 1
    step_count = 0  # Лічильник кроків
        while low <= high:
        step_count += 1
# Безпечне обчислення середнього індексу для уникнення переповнення
        mid = low + (high - low) // 2
    current_value = data_list[mid]
# Крок 1: Елемент знайдено
        if current_value == target:
            print(f"Елемент {target} знайдено! Він має індекс {mid}.")
            return mid
# Крок 2: Цільовий елемент менший, відкидаємо праву половину
        elif current_value > target:
        high = mid - 1
# Крок 3: Цільовий елемент більший, відкидаємо ліву половину
        else:
        low = mid + 1
# Якщо цикл завершився і елемент не знайдено
    print(f"Елемент {target} не знайдено у списку.")
    return -1
```

Алгоритм, що знаходить «найлівіший» та «найправіший» елемент із співпадаючих із шуканим

Стандартний бінарний пошук повертає **один** з індексів цільового елемента, якщо він трапляється кілька разів, але не гарантує, що це буде саме **найлівіший** (перший) або найправіший (останній) індекс

Щоб знайти **найлівіший** елемент, модифікують алгоритм бінарного пошуку. **Ключова зміна** полягає в тому, що коли **знаходиться** збіг, пошук не зупиняється, а зберігається знайдений індекс і продовжується пошук в **лівій половині** послідовності

Щоб знайти **найправіший** елемент із співпадаючих з цільовим значенням у відсортованій послідовності, використовується модифікація бінарного пошуку, аналогічну тій, що застосовується для пошуку найлівішого елемента. **Ключова ідея**: коли **знаходиться** збіг, зберігається цей індекс і продовжується пошук у **правій половині** масиву, щоб перевірити, чи немає там ще подальших входжень