

ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ Python

Урок 5

Сортування, пошук

ЗМІСТ

1. Сортування	3
1.1. Сортування бульбашкою.....	6
1.2. Сортування Шелла (Shell Sort)	11
1.3. Сортування злиттям.....	16
1.4. Швидке сортування (Quick Sort)	21
1.5. Порівняння алгоритмів сортування.....	27
1.5.1. Сортування Шелла.....	36
1.5.2. Сортування злиттям.....	40
1.5.3. Швидка сортування.....	45
2. Пошук.....	49
2.1. Лінійний пошук	49
2.2. Бінарний пошук	53

1. Сортування

Сортування — це процес впорядкування (розміщення у певному порядку) даних у певному наборі.

Найбільш поширені способи такого впорядкування для *числових даних*:

- **сортування за зростанням** (менше число йде у наборі першим, після нього — більше, правіше меншого числа);
- **сортування за спаданням** (від найбільшого числа в наборі до найменшого, при цьому числа з більшими значеннями розташовуються лівіше від своїх менших «сусідів»).

Наприклад:

Початковий набір чисел (у довільному порядку):

4	-2	1	-5	8
---	----	---	----	---

Відсортований набір за зростанням:

-5	-2	1	4	8
----	----	---	---	---

Відсортований набір за зменшенням:

8	4	1	-2	-5
---	---	---	----	----

Для текстових даних використовуються в алфавітному порядку (лексикографічному) та у порядку, зворотному від алфавітного:

Початковий набір текстових даних (наприклад, список імен клієнтів):

Bob	Anna	Joe	Nick
-----	------	-----	------

Відсортований набір *в алфавітному порядку*:

Anna	Bob	Joe	Nick
------	-----	-----	------

Відсортований набір *у порядку, зворотному, від алфавітного*:

Nick	Joe	Bob	Anna
------	-----	-----	------

Важливість сортування полягає в тому, що пошук даних оптимізується (спрощується, прискорюється), а дані представляються у більш прийнятному для читання (зрозумілому) форматі.

У прикладі вище ми відсортували список клієнтів. Також найчастішим завданням є відсортувати, наприклад, список товарів, щоб користувачеві зручніше було переглядати, що є в асортименті.

А список цін товарів традиційно сортується від найдешевших до найдорожчих, або навпаки, тобто потрібна реалізація сортування за зростанням та за спаданням ціни.

Раніше ми вже використовували вбудовані механізми мови Python для сортування: метод `sort()` для сортування списків та функцію `sorted()` для інших типів колекцій.

Однак, якщо в наборі дуже велика кількість елементів, то сортування вбудованими механізмами може зайняти дуже багато часу.

Сьогодні існує велика кількість різних алгоритмів сортування: одні є дуже простими по реалізації, інші — орієнтовані на прискорення процесу сортування, треті — оптимізовані за обсягами пам'яті, що використовуються і т. д.

Також постійно розробляються нові методи сортуння, які найчастіше є варіаціями (удосконаленням) існуючих алгоритмів і виникають з метою покращити їх показники.

Найпоширенішими на практиці є:

- Сортуння бульбашкою (обміном);
- сортуння злиттям;
- сортуння Шелла;
- Швидке сортуння.

Коли ми говоримо про поняття **оптимальності сортуння**, то маємо на увазі його ефективність за обсягом пам'яті, що використовується (під час роботи алгоритму), і за часом роботи.

Коли алгоритм оцінюється за місткістю пам'яті, що витрачається на виконання алгоритму, то враховується як допоміжна пам'ять, яка використовується обчислювальними процесами, так і пам'ять для зберігання даних.

Часова складність (також відома як обчислювальна складність) — це час, за який алгоритм виконає завдання сортуння з урахуванням вхідних даних. Можна сказати, що це деяка функція, яка описує час роботи алгоритму залежно від розміру вхідних даних.

Найчастіше обчислювальну складність описують за допомогою великої літери «O». «O» — це кількість елементарних операцій (просте присвоєння, індексація елемента масиву, арифметичні операції, операції порівняння, логічні операції), які має виконати алгоритм сортуння (адже кожна така операція займає певний час). Такий підхід дає нам можливість оцінити, як буде

змінюватися час виконання алгоритму залежно від кількості вхідних даних.

Коли ми говоримо про обчислювальну складність, то маємо на увазі, що може бути три ситуації:

- найгірший випадок (коли набір елементів відсортовано у зворотному порядку) — тоді кожен елемент не на своєму місці і його потрібно перемістити, тобто виконати максимальну кількість елементарних операцій;
- середній випадок (коли початковий порядок сортованих вхідних даних довільний);
- найкращий випадок (набір вхідних даних забезпечує мінімально можливу кількість елементарних операцій, наприклад, вже відсортований список).

Вибираючи алгоритм, ми часто фактично робимо вибір між об'ємом пам'яті та швидкістю сортування. Завдання з сортування можна вирішити швидко, використовуючи великий обсяг пам'яті (наприклад, задіявши додаткові списки для проміжного зберігання), або повільніше, займаючи менший обсяг пам'яті, але при цьому виконавши більшу кількість елементарних операцій. Тому, порівнюючи різні алгоритми, важливо знати, як залежить обсяг використовуваних обчислювальних ресурсів (обсяг пам'яті та час виконання) від обсягу вхідних даних.

1.1. Сортування бульбашкою

Bubble Sort (*Сортування бульбашкою* або *сортування обміном*) — один із найпростіших для розуміння та реалізації алгоритмів сортування. Його назва походить від особливостей роботи алгоритму: при кожному новому

проході елемент з найбільшим (або маленьким, залежно від порядку сортуння: за зростанням або за спаданням) значенням у списку «піднімається вгору» — «спливає» у кінець списку.

Пари сусідніх елементів порівнюються між собою послідовно і, якщо порядок у парі порушений, то міняються місцями.

Розглянемо на прикладі сортуння списку з п'яти елементів за спаданням. Індксація елементів на рисунку йде знизу вгору, тобто найвищий елемент рисунку — останній у списку. Жовтим кольором покровоко виділяється порівнювана пара елементів, а червоним шрифтом — найменший елемент у парі (т. к. сортуння за спаданням, то «випливати» мають менші «бульбашки») (рис. 1).

6		6		6		6	обмін	2
7		7		7	обмін	2		6
4		4	обмін	2		7		7
2	обміну немає	2		4		4		4
5		5		5		5		5

Рисунок 1

За один (перший прохід) у нас «піднялася» тільки одна «бульбашка» — найменше число — 2 після перевірки чотирьох пар, тобто кількість пар на одиницю менша за кількість невідсортованих елементів. Ми пройшлися від 0-го елемента до N-1-го елемента.

Наступним кроком нам потрібно працювати вже не з усім масивом (число 2, виділене на рисунку 2 зеленим кольором, вже знаходиться у правильній позиції).

2		2		2		2
6		6		6	обмін	4
7		7	обмін	4		6
4	обміну немає	4		7		7
5		5		5		5

Рисунок 2

На другому кроці ми проаналізували три пари, починаючи від 0-го елемента до $N-2$ -го елемента. Вже на цьому кроці можна помітити, що діапазон індексів невідсортованих елементів змінюється від 0 до $N-i$, де i — номер кроку.

Третій крок:

2		2		2
4		4		4
6		6	обмін	5
7	обмін	5		6
5		7		7

Рисунок 3

Четвертий крок:

2	
4	
5	
6	обміну немає
7	

Рисунок 4

Таким чином, ми відсортували наш список за 4 кроки ($N-1$), на кожному з яких відбувалося поступове зміщення елементів із меншим значенням у кінець списку (вгору).

Сортвання бульбашкою складається за списком з кількох кроків, на кожному з яких запускається процес аналізу пар невідсортованої частини списку. Порівнюється кожна пара сусідніх елементів, і елементи змінюються місцями, якщо вони розташовані не в потрібному порядку.

І тут нам знадобляться два цикли. Зовнішній цикл виконуватиме кроки у кількості $N-1$, як у наведеному вище прикладі, який ми проаналізували.

Внутрішній цикл здійснюватиме порівняння пар у невідсортованій частині списку (число ітерацій $N-i$). Якщо i -ий елемент менший за сусідній справа ($i+1$ -го елемента), то вони міняються місцями. Таким чином, найменший елемент буде в правій крайній частині списку.

Реалізуємо алгоритм бульбашкового сортвання як окремої функції. Також створимо функцію для виведення елементів списку, і викличемо її для виведення початкового і відсортованого списків (після виклику функції сортвання).

```
numbers=[5,2,4,7,6]
```

```
def myBubbleSort(myList):  
    for i in range(len(myList)-1):  
        for j in range(len(myList)-i-1):  
            if myList[j]<myList[j+1]:  
                temp=myList[j]  
                myList[j]=myList[j+1]  
                myList[j+1]=temp
```

```
def printList(myList):  
    for index, elem in enumerate(myList):  
        print("element {}: {}".format(index+1, elem))
```

```
print("Original list:")
printList(numbers)
myBubbleSort(numbers)
print("Sorted list:")
printList(numbers)
```

Результат:

```
Original list:
element 1: 5
element 2: 2
element 3: 4
element 4: 7
element 5: 6
Sorted list:
element 1: 7
element 2: 6
element 3: 5
element 4: 4
element 5: 2
```

Рисунок 5

Однак, якщо список вже відсортований (наприклад, спочатку ми маємо 7, 6, 5, 4, 2), то наш алгоритм все одно виконає порівняння усіх пар на кожному кроці.

Удосконалимо його роботу такою модифікацією: якщо під час поточного кроку обміну не було (а це означає, що усі елементи вже відсортовані), то перервемо процес сортування (зовнішній цикл):

```
def myBubbleSort_v1(myList):
    for i in range(len(myList)-1):
        sortedFlag=True
```

```

for j in range(len(myList)-i-1):
    if myList[j]<myList[j+1]:
        temp=myList[j]
        myList[j]=myList[j+1]
        myList[j+1]=temp
        sortedFlag=False

if sortedFlag:
    break

```

1.2. Сортвання Шелла (Shell Sort)

Алгоритм сортвання **Шелла** — це вдосконалена версія сортвання вставлянням. Тому спочатку розглянемо базовий алгоритм — сортвання вставлянням.

Як і бульбашкове сортання, алгоритм сортання вставлянням дуже простий у реалізації та легкий у розумінні. Основна ідея — це віртуальний поділ списку на відсортовану частину (на початку списку) і невідсортовану. На кожному кроці алгоритм бере елемент із невідсортованої частини і вставляє його у правильну позицію у відсортованій частині списку.

Розглянемо процес сортання списку із N елементів за зростанням. Першим кроком передбачається, що відсортована частина списку складається лише з першого елемента (з індексом 0). Далі йдемо за списком від `list[1]` до `list[N-1]` і порівнюємо кожен елемент з його «попередниками», розташованими лівіше.

Якщо поточний елемент менший від лівого сусіда, то порівнюємо його з попереднім елементом лівого сусіда. І так робимо, доки не знайдемо його правильну позицію

у відсортованій лівій частині списку. Також потрібно переміщати (зрушувати) великі елементи на одну позицію вгору (після кожного порівняння з результатом «поточний менше лівого сусіда»), щоб звільнити місце для вставляння (рис. 6).

	0	1	2	3	4
1	4	3	2	10	12
	1 зсув				
2	3	4	2	10	12
	2 зсуви				
3	2	3	4	10	12
	залишається				
4	2	3	4	10	12
	залишається				

Рисунок 6

Зеленим на рисунку виділено відсортовану частину масиву, яка змінюється (доповнюється) на кожному кроці.

Червоним показано поточний елемент, для якого ми на поточному кроці визначаємо правильну позицію (виділена жовтим).

Алгоритм Шелла покращує середню часову складність сортування вставлянням, коли елементи можуть переміщатися тільки на одну позицію. Якщо потрібна позиція у відсортованій частині знаходиться далеко від поточної позиції елемента, потрібно багато переміщень (зсувів), які збільшують час виконання сортування.

Алгоритм Шелла дозволяє переміщати та міняти місцями віддалені елементи. Цей алгоритм спочатку сортує елементи, які знаходяться далеко один від одного, а потім елементи, розташовані на найближчій відстані.

Ця відстань, у межах якої проводиться сортвання, називається інтервалом.

Як інтервали для сортвання списку довжиною N , зазвичай використовується така послідовність: $N/2$, $N/4$, ..., 1.

Фактично, на кожному кроці ми визначаємо набір списків, які вміщують елементи початкового списку, що знаходяться на зазначеному інтервалі один від одного. І проводимо сортвання вставлянням всередині цих списків. Далі, при скороченні інтервалу, ми скорочуємо кількість цих списків (вони стають довшими і відсортованими). Останнім кроком (коли інтервал дорівнює 1) також використовується сортання вставлянням, але вже усього списку, більша частина якого вже відсортована.

Розглянемо з прикладу.

На першому кроці N дорівнює 8 (кількість елементів у списку), тому елементи знаходяться один від одного на інтервалі 4 ($N/2 = 4$) (рис. 7).

N=8	0	1	2	3	4	5	6	7
1	33	31	40	8	12	17	25	42
N/2=4								

Рисунок 7

Ми отримали 4 списки: жовтий, помаранчевий, блакитний та зелений. Елементи всередині кожного з цих списків будуть порівнюватися та змінюватись місцями, якщо вони розміщені не по черзі за зростанням.

Тут у першому циклі елемент на 0-й позиції порівнюватиметься з елементом на 4-й позиції. 0-й елемент

більший, тому він буде замінений елементом на 4-й позиції. В іншому випадку (наприклад, при порівнянні 3-го та 7-го елементів) порядок залишається незмінним. Цей процес продовжиться для інших елементів кожного з утворених списків.

Результат після першого кроку:

0	1	2	3	4	5	6	7
12	17	25	8	33	31	40	42

Рисунок 8

Далі у другому циклі елементи одного підписку знаходяться на інтервалі 2 ($N/4 = 2$). Тому отримаємо лише два підписки: жовтий та блакитний.

	2	0	1	2	3	4	5	6	7
$N/4=2$	12	17	25	8	33	31	40	42	

Рисунок 9

Кожен із цих підписків сортується алгоритмом вставлення.

Результат після другого кроку:

0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42

Рисунок 10

Третім кроком елементи знаходяться один від одного на інтервалі 1, тобто ми отримуємо один список довжиною N , який також сортується методом вставлення (рис. 11).

Як ви вже помітили, зсуви при сортуванні вставленням відбуваються на невеликій відстані (на 1 елемент у прикладі).

0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
зсув							
8	12	25	17	33	31	40	42
залишається							
8	12	25	17	33	31	40	42
зсув							
8	12	17	25	33	31	40	42
залишається							
8	12	17	25	33	31	40	42
зсув							
8	12	17	25	33	31	40	42
залишається							
8	12	17	25	33	31	40	42
залишається							
8	12	17	25	33	31	40	42

Рисунок 11

Створимо окрему функцію для реалізації сортування вставлянням `InsertionSort()` для елементів на відстані, яка дорівнює інтервалу один від одного, і викличемо її у циклі функції сортування Шелла `ShellSort()`:

```
def ShellSort(myList):
    subListN = len(myList)//2
    step=1
    while subListN > 0:
        for startInd in range(subListN):
            InsertionSort(myList, startInd, subListN)
            print("Interval={}. After step {}: {}".format(subListN, step, myList))
            subListN = subListN // 2
    def InsertionSort(myList, startInd, gapValue):
        for i in range(startInd + gapValue, len(myList), gapValue):
            currentElem = myList[i]
            currentInd = i
```

```

while currentInd >= gapValue and
    myList[currentInd-gapValue] >
        currentElem:
    myList[currentInd] = myList[currentInd -
                                gapValue]
    currentInd = currentInd-gapValue
myList[currentInd]=currentElem

numbers=[33,31,40,8,12,17,25,42]
print("Original list: {}".format(numbers))

ShellSort(numbers)
print("Sorted list: {}".format(numbers))

```

Результат:

```

Original list: [33, 31, 40, 8, 12, 17, 25, 42]
Interval=4. After step 1: [12, 17, 25, 8, 33, 31, 40, 42]
Interval=2. After step 1: [12, 8, 25, 17, 33, 31, 40, 42]
Interval=1. After step 1: [8, 12, 17, 25, 31, 33, 40, 42]
Sorted list: [8, 12, 17, 25, 31, 33, 40, 42]

```

Рисунок 12

1.3. Сортування злиттям

Тепер ми розглянемо алгоритм, який належить до категорії швидких алгоритмів сортування. Ці алгоритми призначені для роботи з великими наборами даних і мають кращі часові характеристики, порівняно з розглянутими раніше алгоритмами.

В основі сортування злиттям знаходиться принцип «розділяй і володарюй». Список поділяється на рівні, або практично рівні, частини, кожна з яких сортується окремо. Після чого вже впорядковані частини об'єднуються

в одну. Така процедура розділення списку повторюється рекурсивно.

Якщо список порожній або містить лише один елемент, він відсортований (базовий випадок — зупинка рекурсивного поділу). Інакше відбувається рекурсивний виклик сортування злиттям для кожної з частин окремо.

Потім відсортовані фрагменти (які містяться в окремих списках) необхідно зібрати у правильному порядку — тобто виконати злиття. При цьому, на кожному кроці злиття з кожного з двох списків вибирається менший елемент пари (якщо сортування за зростанням) і записується зліва у вільну клітинку додаткового списку. Якщо один із списків закінчиться раніше, то елементи іншого будуть записані (додані в кінець) до утвореного списку.

Опишемо загальні кроки рекурсивного сортування, на вхід якого подається список (або його частини на наступних кроках рекурсії):

Якщо довжина списку більше 1:

1. Знайти точку для розділення списку на 2 половини:
 $m = (\text{поточна довжина списку})/2$.
2. Викликати сортування для першої (лівої) половини списку (від початку до m -го елемента).
3. Викликати сортування для другої (правої) половини списку (від m -го елемента до останнього елемента).
4. Виконати злиття двох відсортованих списків (які були повернені рекурсивними викликами після кроків 2 і 3):
 - 4.1. Доки є елементи у лівому і правому списку:
 - Якщо поточний елемент лівого списку менший, ніж поточний елемент із правого, то до утворе-

ного списку вставляється елемент лівого списку, якщо ні — з правого списку.

- Після вставляння елемента зі списку перейти до наступного елемента, а для іншого списку (з якого вставляння не відбувалося) поточний елемент не змінюється.

4.2. Перевіряємо, якщо в якомусь списку залишився елемент, додаємо його кінець з результуючого списку.

Розглянемо з прикладу (рис. 13).

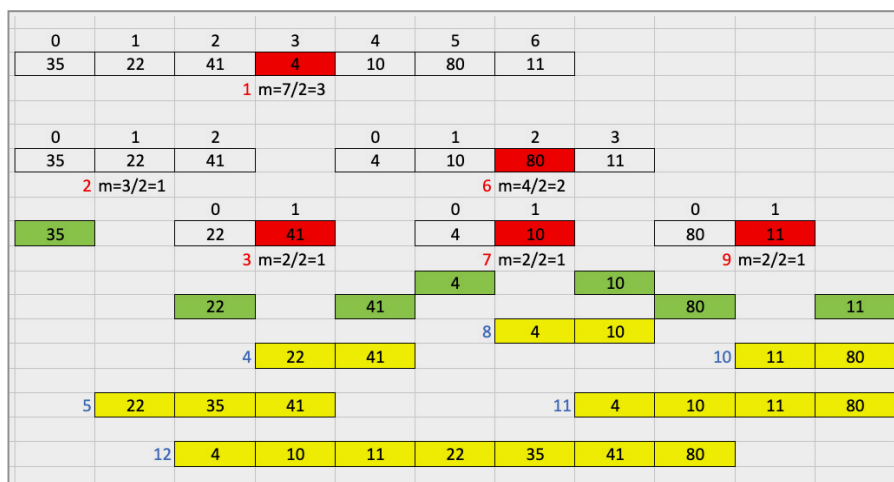


Рисунок 13

На рисунку 13 показані усі 12 кроків сортування злиттям: номери кроків виділені червоним кольором шрифту і відповідають крокам розділення списку, а номери блакитного кольору — крокам злиття. Червоним кольором виділено точки розбиття списку, а зеленим — точки зупинки рекурсії і початку «підйому» — злиття.

Результати проміжних і фінального злиття виділені жовтим кольором.

Реалізуємо нашу функцію сортування злиттям:

```
def MergeSort(myList):
    if len(myList) > 1:
        # Finding the middle of the list
        m = len(myList)//2
        print("m: {}".format(m))

        # Splitting a list into left and right parts
        leftPart = myList[:m]
        rightPart = myList[m:]
        print("left: {}".format(leftPart))
        print("right: {}".format(rightPart))

        # Sorting the left part
        MergeSort(leftPart)

        # Sorting the right part
        MergeSort(rightPart)

        # merge steps
        i = j = k = 0

        # Copy data to sorted list from leftPart-list
        # and rightPart-list
        while i < len(leftPart) and
            j < len(rightPart):
            if leftPart[i] < rightPart[j]:
                myList[k] = leftPart[i]
                i += 1
            else:
                myList[k] = rightPart[j]
                j += 1
                k += 1

        # Checking if any element was left in
        # leftPart-list
```

```

while i < len(leftPart):
    myList[k] = leftPart[i]
    i += 1
    k += 1

# Checking if any element was left in
rightPart-list
while j < len(rightPart):
    myList[k] = rightPart[j]
    j += 1
    k += 1

print("temp merge: {}".format(myList))

numbers=[35, 22, 41, 4, 10, 80, 11]
print("Original list: {}".format(numbers))
MergeSort(numbers)
print("Sorted list: {}".format(numbers))

```

Результати тестування:

```

Original list: [35, 22, 41, 4, 10, 80, 11]
m: 3
left: [35, 22, 41]
right: [4, 10, 80, 11]
m: 1
left: [35]
right: [22, 41]
m: 1
left: [22]
right: [41]
temp merge: [22, 41]
temp merge: [22, 35, 41]
m: 2
left: [4, 10]
right: [80, 11]
m: 1
left: [4]

```

Рисунок 14

```

right: [10]
temp merge: [4, 10]
m: 1
left: [80]
right: [11]
temp merge: [11, 80]
temp merge: [4, 10, 11, 80]
temp merge: [4, 10, 11, 22, 35, 41, 80]
Sorted list: [4, 10, 11, 22, 35, 41, 80]

```

Рисунок 14 (продовження)

1.4. Швидке сортування (Quick Sort)

Як і розглянуте раніше сортування злиттям, швидке сортування відноситься до категорії алгоритмів, що використовують стратегію «розділяй і владарюй».

Quick Sort — один із найпопулярніших методів сортування. Більшість готових бібліотек і методів сортування у різних мовах програмування використовують цей алгоритм як основу.

Основна ідея полягає в розділенні набору даних на дві частини. Елемент, який знаходиться у точці поділу, називається опорним (*pivot*).

Далі відбувається переміщення елементів набору (наприклад, послідовність потрібно впорядкувати за зростанням): елементи менші, ніж опорний, переміщуються ліворуч від нього, а великі елементи — у праву частину набору.

Така процедура повторюється рекурсивно — окремо для лівої та правої частини, доки не відсортується весь набір, тобто алгоритм сягне такої ситуації, що у кожній частині залишиться один елемент. Пам'ятаймо, що

порожній список та список, який складається з одного елемента, є відсортованим.

Існує багато різних версій, як вибрати опорну точку: можна вибрати перший елемент як опорний, останній елемент, випадковий елемент набору або медіану (елемент, розташований посередині набору).

Найбільш вдалим (і тому рекомендованим) є вибір медіани як опорного елемента: тоді по обидві сторони від опорного елемента виявляється приблизно рівна кількість елементів.

Розглянемо роботу алгоритму покроково (при сортуванні за зростанням).

1. Введемо дві змінні для зберігання індексів першого (`firstInd`) та останнього (`lastInd`) елементів сортованого набору (на першому кроці це весь набір `myList`) та змінну для зберігання індексу (`midInd`) опорного елемента.

Розрахуємо значення `midInd`: $\text{midInd} = (\text{firstInd} + \text{lastInd})/2$ або це половина набору (число елементів / 2).

2. Розбиваємо набір і переміщуємо елементи лівіше або правіше від опорного елемента: елементи з правої частини менші, ніж опорний, переміщуються вліво від нього, а великі елементи з лівої частини — у праву частину набору.

- 2.1. Переміщення відбувається шляхом обміну двох елементів: збільшуємо значення `firstInd` на 1, доки `myList[firstInd] < myList[midInd]`, таким чином, виявляючи індекс першого кандидата для перенесення до правої частини.

2.2. У такий спосіб (зсуваючи `lastInd` вліво, зменшуючи його значення на 1, виявляємо позицію елемента-кандидата для перенесення ліворуч від опорного.

2.3. Знайдені ліворуч та праворуч від опорного «кандидати» змінюються місцями.

Кроки 2.1-2.3 повторюються, доки `firstInd < lastInd`.

3. Перевіряємо: якщо у лівій частині більше одного елемента, то повторюємо рекурсивне впорядкування цієї частини (кроки 1-2). Таку ж перевірку та виклик сортування виконуємо потім і для правої частини набору.

Розглянемо на прикладі сортування за зростанням такого списку (рис. 15):

0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42

Рисунок 15

При першому виклику нашої функції-сортування ми опрацюємо весь масив.

Визначаємо позиції `firstInd`, `lastInd` та `midInd` (рис. 16).

1	0	1	2	3	4	5	6	7
	12	8	25	17	33	31	40	42
	firstInd			midInd				lastInd

Рисунок 16

Далі починаємо пересувати `firstInd` праворуч та `lastInd` ліворуч, щоб знайти можливих кандидатів на обмін (рис. 17).

0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
на місці							
0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
		на місці					
0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
		обмін					
0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
		обмін				на місці	
0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
		обмін			на місці		
0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
		обмін		на місці			
0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
		обмін		на місці			
0	1	2	3	4	5	6	7
12	8	25	17	33	31	40	42
		обмен	обмен				
0	1	2	3	4	5	6	7
12	8	17	25	33	31	40	42

Рисунок 17

Як бачимо, перший кандидат на обмін у лівій частині — це число 25 (2-й елемент). А от у частині справа усі елементи виявилися більшими за опорний (тобто в правильній частині списку). Таким чином, `lastInd` дійшов до позиції опорного елемента, і оскільки опорний виявився меншим за кандидата ліворуч, то стався обмін.

Оскільки в обох частинах кількість елементів більше 1, то описана процедура сортування буде запущена для кожної частини окремо.

Розглянемо приклад подальшої обробки лівої частини списку (рис. 18).

2	0	1	2	3
	12	8	17	25
обмін				
	0	1	2	3
	12	8	17	25
обмін			на місці	
	0	1	2	3
	12	8	17	25
обмін		на місці		
	0	1	2	3
	12	8	17	25
обмін	обмін			
	0	1	2	3
	8	12	17	25

Рисунок 18

Сталася аналогічна ситуація: кандидат із лівої частини обмінявся з опорним елементом місцями. У лівій частині один елемент. Тому цей фрагмент вже готовий-відсортований. А от у частині справа кількість елементів більша за 1.

Виконуємо процедуру сортування окремо для правої частини.

3	2	3
	17	25
firstInd		
	на місці	

Рисунок 19

Як бачимо, **firstInd** та **midInd** збігаються, і кандидатів для обміну у правій частині також немає. При цьому кількість

елементів правої та лівої частини дорівнює 1, тобто на цьому етапі отримуємо готовий відсортований фрагмент.

Оскільки при використанні алгоритму швидкого сортування копії списку або його частин не створюються, то ми вже отримали на поточному етапі половину відсортованого списку:

0	1	2	3
8	12	17	25

Рисунок 20

Після виконання аналогічних кроків для правої частини отримаємо повністю відсортований набір:

0	1	2	3	4	5	6	7
8	12	17	25	31	33	40	42

Рисунок 21

Реалізуємо нашу функцію швидкого сортування:

```
def QuickSort(myList, firstInd, lastInd):
    global C
    global E
    i = firstInd
    j = lastInd
    # pivot element in the middle
    pivotElem = myList[firstInd +
                        (lastInd - firstInd) // 2]
    while i <= j:
        while myList[i] < pivotElem:
            i += 1
            C+=1
        while myList[j] > pivotElem:
            j -= 1
            C+=1
```

```

C+=1
if i <= j: # swap
    E+=1
    myList[i], myList[j] = myList[j], myList[i]
    i += 1
    j -= 1
if firstInd < j: # sort left part
    QuickSort(myList, firstInd, j)
if i < lastInd: # sort right part
    QuickSort(myList, i, lastInd)
print("C={}, E={}".format(C,E))
return myList

```

Результати тестування (рис. 22, 23):

Original list: [12, 8, 25, 17, 33, 31, 40, 42]

Рисунок 22

Sorted list: [8, 12, 17, 25, 31, 33, 40, 42]

Рисунок 23

1.5. Порівняння алгоритмів сортування

Кожен із розглянутих нами алгоритмів сортування має як переваги, так і недоліки. І тепер, коли ми вивчили особливості найпоширеніших алгоритмів сортування, проведемо їх порівняльний аналіз.

Чому і коли це важливо? Уявімо, що ми перейшли від невеликих розмірів наборів даних у кількості до 1000 елементів до наборів, що містять 100000 або навіть більше мільйона елементів (цілком реальна цифра навіть для звичайного інтернет-магазину). У таких ситуаціях ефективність алгоритму сортування (яка є лише однією операцією серед

одночасно запущених функцій програмного забезпечення) може серйозно вплинути на роботу усього додатка.

Часто оцінку складності алгоритмів проводять за часом виконання або використовуюваної у процесі сортування пам'яті

Однак час виконання (швидкість роботи алгоритму) не є правильною мірою оцінки алгоритму, оскільки виконання залежить від параметрів конкретного комп'ютера, типів даних, що використовуються алгоритмом, особливостей мови програмування і т. д.

Більш надійним показником оптимальності сортування є обчислювальна складність, тобто кількість елементарних операцій (просто присвоєння, індексація елемента масиву, арифметичні операції, операції порівняння, логічні операції), які має виконати алгоритм сортування (адже кожна така операція займає певний час).

Але й цей показник, заснований на кількості виконаних операторів, не є ідеальною мірою оптимальності, оскільки кількість операторів у реалізації алгоритму залежить як від мови програмування, так і від стилю розробника.

Найкращим рішенням буде представити час роботи алгоритму функцією $f(n)$, яка залежить від розміру вхідних даних (n елементів у наборі). Такий вид порівняння алгоритмів не залежить ані від машинного часу, ані від використовуваних технологій і стилю програмування.

Для позначення такої часової складності використовується велика літера **O-Big-O Notation**.

Можна сказати, що нотація **Big-O** — це спосіб відстеження того, наскільки швидко зростає час виконання по відношенню до розміру вхідних даних.

Нотація **Big-O** може виражати найкращий, найгірший і середній час роботи алгоритму. Більшість обговорень **Big-O** для різних алгоритмів зосереджено на «верхній межі» часу виконання алгоритму, які часто називають найгіршим випадком. Про особливості цих трьох випадків тестування алгоритмів поговоримо трохи згодом.

А зараз давайте розглянемо можливі значення оцінки складності у нотації **Big-O** на прикладах.

O(1) — час роботи алгоритму — константа (тобто постійно) незалежно від обсягу вхідних даних.

Припустимо, що ми маємо список оцінок студентів. Якщо нам потрібно дізнатися оцінку певного студента (ми чітко знаємо його номер у списку групи, тобто індекс елемента відомий), то ми отримаємо його оцінку зі списку за індексом за той самий час, як і для списку із 10 студентів, так і для списку із 1000. Таким чином, отримуємо **O(1)**.

O(n) — лінійна складність: час виконання алгоритму збільшуватиметься з тією ж швидкістю, що й кількість вхідних даних.

Наприклад, у нашій групі 50 студентів і в кожного потрібно запитати, чи хоче він записатися на додаткове заняття з програмування (кожний відповідає тільки «так» чи «ні»). Припустимо, час відповіді одного студента становить одну хвилину. Тоді на опитування групи із 50 студентів піде 50 хвилин. А знаючи, що ми маємо опитати 200 студентів, ми відразу ж виділимо у своєму графіку 200 хвилин.

У програмуванні одним із найпоширеніших завдань із лінійною складністю є обхід масиву. Наприклад, виведення усіх елементів масиву або зміна значення кожного

елемента. Таким чином, складність фрагмента коду з циклом, який виконується n разів, — $O(n)$.

Алгоритм лінійного пошуку в наборі, при якому ми перевіряємо кожен елемент на рівність ключа, також має лінійну складність.

$O(n^2)$ — квадратична складність означає, що обчислення виконується за квадратичний час, тобто дорівнює квадрату розміру вхідних даних.

Повернемося до нашого прикладу опитування 10 студентів. Ми запитуємо першого студента, чи записується він на додаткове заняття. Після цього запитуємо його ж про думку решти 9-ти студентів: як він вважає, чи запишеться другий студент, третій і т. д. Цей процес опитування про себе та інших повторимо з кожним студентом групи. Загальна кількість разів запитання буде $10 \cdot 10$, тобто 10^2 , оскільки код, що реалізує процес опитування таким способом міститиме два цикли: один цикл (опитування студента про всіх) буде вкладеним у інший (повторити таке опитування про себе та про всіх для кожного студента групи).

У програмуванні наявність двох вкладених циклів часто є ознакою того, що цей фрагмент коду має складність $O(N^2)$.

$O(\log n)$ — логарифмічна складність означає, що час виконання зростає пропорційно до логарифму розміру вхідних даних, тобто час виконання збільшується повільно зі збільшенням кількості вхідних даних.

Прикладом є бінарний пошук, де на кожному кроці ми скорочуємо простір пошуку вдвічі. Ми порівнюємо центральний елемент набору з ключем пошуку, якщо елемент більший за ключ, то виключаємо з пошуку усі

елементи правіше центрального, якщо ні — усі елементи лівіше центрального і т. д., доки не знайдемо ключ або ліва та права межі не зімнуться.

Тепер ми можемо визначити оцінку складності у нотації **Big-O** для кожного із вивчених алгоритмів сортуння.

Також додатково проведемо порівняння за кількістю елементарних операцій, які має виконати кожен алгоритм. В якості таких операцій (число яких ми розрахуємо під час виконання кожного алгоритму) будемо використовувати операцію порівняння (**C**) та операцію обміну (**E**).

Однак, коли ми говоримо про обчислювальну складність, то пам'ятаємо про можливі три ситуації:

- найгірший випадок (коли набір елементів відсортовано у зворотному порядку) — тоді кожен елемент знаходиться не на своєму місці і його потрібно перемістити, тобто виконати максимальну кількість елементарних операцій;
- середній випадок (коли початковий порядок сортованих вхідних даних довільний);
- найкращий випадок (набір вхідних даних забезпечує мінімально можливу кількість елементарних операцій, наприклад, вже відсортований список).

Таким чином, тестування проведемо на трьох наборах даних (по 20 елементів у кожному):

- набір із випадково згенерованими числовими значеннями (середній випадок);
- набір, в якому числові значення відсортовані у порядку, зворотному порядку сортуння (найгірший випадок);
- набір, в якому половина значень вже відсортована у правильному порядку (найкращий випадок).

Внесемо розрахунок кількості порівнянь та перестановок елементів у наш код алгоритмів сортування (відповідно до його особливостей).

Також нам потрібно створити функції для генерації набору середнього випадку та найкращого.

```
from random import randint

def printList(myList):
    for index, elem in enumerate(myList):
        print("element {}: {}".format(index+1, elem))

def generateNum(n):
    nList=[]
    for i in range(n):
        nList.append(randint(11,100))
    return nList
```

Для найгіршого випадку братимемо дані після сортування в режимі «середнього» випадку, розгорнувши їх у зворотному напрямку за допомогою функції `sorted()`.

Почнемо із сортування бульбашкою.

Ми бачимо у коді два вкладені цикли, за допомогою яких алгоритм двічі проходить по усьому списку. Таким чином, маємо тут складність $O(n^2)$.

```
def myBubbleSort(myList):
    global C
    global E
    print("C={}, E={}".format(C,E))

    for i in range(len(myList)-1):
        sortedFlag=True
```



```

    for j in range(len(myList)-i-1):
        C+=1
        if myList[j]<myList[j+1]:
            E+=1
            temp=myList[j]
            myList[j]=myList[j+1]
            myList[j+1]=temp
            sortedFlag=False
    if sortedFlag:
        break
    print("C={}, E={}".format(C,E))

```

```

C=0
E=0
numbers1=generateNum(20)

print("Original list1:")
printList(numbers1)
myBubbleSort(numbers1)
print("Sorted list1:")
printList(numbers1)

C=0
E=0

numbers2=sorted(numbers1)
print("Original list2:")
printList(numbers2)
myBubbleSort(numbers2)
print("Sorted list2:")
printList(numbers2)

C=0
E=0

numbers3=generateNum(10)+[10,9,8,7,6,5,4,3,2,1]

```

```

print("Original list3:")
printList(numbers3)
myBubbleSort(numbers3)
print("Sorted list3:")
printList(numbers3)

```

Результати тестування алгоритму сортування бульбашкою (табл. 1).

Таблиця 1

Початкові дані	Відсортовані
Середній випадок	
original list1: element 1: 31 element 2: 82 element 3: 100 element 4: 89 element 5: 62 element 6: 56 element 7: 69 element 8: 94 element 9: 59 element 10: 98 element 11: 29 element 12: 47 element 13: 38 element 14: 34 element 15: 51 element 16: 91 element 17: 71 element 18: 43 element 19: 33 element 20: 77	C=175, E=78 Sorted list1: element 1: 100 element 2: 98 element 3: 94 element 4: 91 element 5: 89 element 6: 82 element 7: 77 element 8: 71 element 9: 69 element 10: 62 element 11: 59 element 12: 56 element 13: 51 element 14: 47 element 15: 43 element 16: 38 element 17: 34 element 18: 33 element 19: 31 element 20: 29
Найгірший випадок	
	C=190, E=190

Початкові дані	Відсортовані
Original list2: element 1: 29 element 2: 31 element 3: 33 element 4: 34 element 5: 38 element 6: 43 element 7: 47 element 8: 51 element 9: 56 element 10: 59 element 11: 62 element 12: 69 element 13: 71 element 14: 77 element 15: 82 element 16: 89 element 17: 91 element 18: 94 element 19: 98 element 20: 100	Sorted list2: element 1: 100 element 2: 98 element 3: 94 element 4: 91 element 5: 89 element 6: 82 element 7: 77 element 8: 71 element 9: 69 element 10: 62 element 11: 59 element 12: 56 element 13: 51 element 14: 47 element 15: 43 element 16: 38 element 17: 34 element 18: 33 element 19: 31 element 20: 29
Найкращий випадок	
Original list3: element 1: 70 element 2: 44 element 3: 28 element 4: 65 element 5: 22 element 6: 16 element 7: 29 element 8: 65 element 9: 42 element 10: 46 element 11: 10	C=112, E=20 Sorted list3: element 1: 70 element 2: 65 element 3: 65 element 4: 46 element 5: 44 element 6: 42 element 7: 29 element 8: 28 element 9: 22 element 10: 16 element 11: 10

Початкові дані	Відсортовані
element 12: 9	element 12: 9
element 13: 8	element 13: 8
element 14: 7	element 14: 7
element 15: 6	element 15: 6
element 16: 5	element 16: 5
element 17: 4	element 17: 4
element 18: 3	element 18: 3
element 19: 2	element 19: 2
element 20: 1	element 20: 1

1.5.1. Сортування Шелла

Використовує алгоритм сортування вставлянням, який також у найгіршому випадку (дані відсортовані у зворотному порядку) має $O(n^2)$.

Проте, у найкращому випадку (коли масив спочатку відсортований правильно) цей спосіб забезпечить $O(n)$.

```
def InsertionSort(myList, startInd, gapValue):
    global C
    global E

    for i in range(startInd+gapValue, len(myList),
                    gapValue):
        currentElem = myList[i]
        currentInd = i
        C+=1
        while currentInd >= gapValue and
              myList[currentInd-gapValue] > currentElem:
            myList[currentInd] = myList[currentInd-
            gapValue]
            currentInd = currentInd-gapValue
            E+=1
        myList[currentInd] = currentElem
        E+=1
```

```

def ShellSort(myList):
    global C
    global E
    print("C={}, E={}".format(C,E))

    subListN = len(myList)//2
    step=1
    while subListN > 0:

        for startInd in range(subListN):
            InsertionSort(myList,startInd,subListN)

        #print("Interval={}. After step {}: {}".format(subListN,step,myList))

        subListN = subListN // 2
    print("C={}, E={}".format(C,E))

```

```

C=0
E=0
numbers1=generateNum(20)

print("Original list1:")
printList(numbers1)
ShellSort(numbers1)
print("Sorted list1:")
printList(numbers1)

C=0
E=0

numbers2=sorted(numbers1,reverse=True)
print("Original list2:")
printList(numbers2)
ShellSort(numbers2)

```

```

print("Sorted list2:")
printList(numbers2)

C=0
E=0

numbers3=[1,2,3,4,5,6,7,8,9,10]+generateNum(10)

print("Original list3:")
printList(numbers3)
ShellSort(numbers3)
print("Sorted list3:")
printList(numbers3)

```

Результати тестування алгоритму сортування Шелла
(табл. 2)

Таблиця 2

Початкові дані	Відсортовані
Середній випадок	
Original list1:	C=62, E=95
element 1: 11	Sorted list1:
element 2: 45	element 1: 11
element 3: 70	element 2: 12
element 4: 35	element 3: 21
element 5: 82	element 4: 22
element 6: 12	element 5: 32
element 7: 22	element 6: 33
element 8: 50	element 7: 35
element 9: 32	element 8: 39
element 10: 67	element 9: 45
element 11: 74	element 10: 47
element 12: 33	element 11: 50
element 13: 76	element 12: 63
element 14: 70	element 13: 67
	element 14: 70

Початкові дані	Відсортовані
element 15: 63	element 15: 70
element 16: 21	element 16: 73
element 17: 78	element 17: 74
element 18: 47	element 18: 76
element 19: 39	element 19: 78
element 20: 73	element 20: 82

Найгірший випадок

Original list2:	C=62, E=98
element 1: 82	Sorted list2:
element 2: 78	element 1: 11
element 3: 76	element 2: 12
element 4: 74	element 3: 21
element 5: 73	element 4: 22
element 6: 70	element 5: 32
element 7: 70	element 6: 33
element 8: 67	element 7: 35
element 9: 63	element 8: 39
element 10: 50	element 9: 45
element 11: 47	element 10: 47
element 12: 45	element 11: 50
element 13: 39	element 12: 63
element 14: 35	element 13: 67
element 15: 33	element 14: 70
element 16: 32	element 15: 70
element 17: 22	element 16: 73
element 18: 21	element 17: 74
element 19: 12	element 18: 76
element 20: 11	element 19: 78
	element 20: 82

Найкращий випадок

Original list3:	C=62, E=76
element 1: 1	Sorted list3:
element 2: 2	element 1: 1
element 3: 3	element 2: 2
element 4: 4	element 3: 3
	element 4: 4

Початкові дані	Відсортовані
element 5: 5	element 5: 5
element 6: 6	element 6: 6
element 7: 7	element 7: 7
element 8: 8	element 8: 8
element 9: 9	element 9: 9
element 10: 10	element 10: 10
element 11: 81	element 11: 20
element 12: 86	element 12: 32
element 13: 87	element 13: 39
element 14: 41	element 14: 41
element 15: 57	element 15: 57
element 16: 20	element 16: 74
element 17: 77	element 17: 77
element 18: 39	element 18: 81
element 19: 74	element 19: 86
element 20: 32	element 20: 87

1.5.2. Сортування злиттям

Розбиває сортований набір на кожному кроці на дві частини, таким чином оцінка складності — $O(\log n)$.

```
def MergeSort(myList):
    global C
    global E
    if len(myList) > 1:

        # Finding the middle of the list
        m = len(myList) // 2
        print("m: {}".format(m))

        # Splitting a list into left and right parts
        leftPart = myList[:m]
        rightPart = myList[m:]
        print("left: {}".format(leftPart))
        print("right: {}".format(rightPart))
```



```

# Sorting the left part
MergeSort(leftPart)

# Sorting the right part
MergeSort(rightPart)

# merge steps
i = j = k = 0

# Copy data to sorted list from leftPart-list
and rightPart-list
while i < len(leftPart) and j < len(rightPart):
    C+=1
    if leftPart[i] < rightPart[j]:
        myList[k] = leftPart[i]
        i += 1
        E+=1
    else:
        myList[k] = rightPart[j]
        j += 1
        E+=1
    k += 1

# Checking if any element was left
in leftPart-list
while i < len(leftPart):
    myList[k] = leftPart[i]
    i += 1
    k += 1
    E+=1

# Checking if any element was left
in rightPart-list
while j < len(rightPart):
    myList[k] = rightPart[j]
    j += 1
    k += 1

```

```

        E+=1
    #print("temp merge: {}".format(myList))
    print("C={}, E={}".format(C,E))

```

```

C=0
E=0
numbers1=generateNum(20)

print("Original list1:")
printList(numbers1)
MergeSort(numbers1)
print("Sorted list1:")
printList(numbers1)

C=0
E=0

numbers2=sorted(numbers1,reverse=True)
print("Original list2:")
printList(numbers2)
MergeSort(numbers2)
print("Sorted list2:")
printList(numbers2)

C=0
E=0

numbers3=[1,2,3,4,5,6,7,8,9,10]+generateNum(10)

print("Original list3:")
printList(numbers3)
MergeSort(numbers3)
print("Sorted list3:")
printList(numbers3)

```

Результати тестування алгоритму сортування злиттям (табл. 3).

Таблиця 3

Початкові дані	Відсортовані
Середній випадок	
Original list1: element 1: 57 element 2: 25 element 3: 65 element 4: 57 element 5: 63 element 6: 79 element 7: 27 element 8: 14 element 9: 83 element 10: 71 element 11: 45 element 12: 30 element 13: 40 element 14: 55 element 15: 59 element 16: 94 element 17: 72 element 18: 78 element 19: 15 element 20: 38	C=61, E=88 Sorted list1: element 1: 14 element 2: 15 element 3: 19 element 4: 23 element 5: 35 element 6: 38 element 7: 40 element 8: 42 element 9: 47 element 10: 56 element 11: 61 element 12: 67 element 13: 72 element 14: 74 element 15: 76 element 16: 81 element 17: 87 element 18: 89 element 19: 93 element 20: 96
Найгірший випадок	
Original list2: element 1: 96 element 2: 93 element 3: 89 element 4: 87 element 5: 81 element 6: 76	C=48, E=88 Sorted list2: element 1: 14 element 2: 15 element 3: 19 element 4: 23 element 5: 35 element 6: 38

Початкові дані	Відсортовані
element 7: 74	element 7: 40
element 8: 72	element 8: 42
element 9: 67	element 9: 47
element 10: 61	element 10: 56
element 11: 56	element 11: 61
element 12: 47	element 12: 67
element 13: 42	element 13: 72
element 14: 40	element 14: 74
element 15: 38	element 15: 76
element 16: 35	element 16: 81
element 17: 23	element 17: 87
element 18: 19	element 18: 89
element 19: 15	element 19: 93
element 20: 14	element 20: 96

Найкращий випадок

Original list3:	C=45, E=88
element 1: 1	Sorted list3:
element 2: 2	element 1: 1
element 3: 3	element 2: 2
element 4: 4	element 3: 3
element 5: 5	element 4: 4
element 6: 6	element 5: 5
element 7: 7	element 6: 6
element 8: 8	element 7: 7
element 9: 9	element 8: 8
element 10: 10	element 9: 9
element 11: 78	element 10: 10
element 12: 67	element 11: 17
element 13: 41	element 12: 41
element 14: 51	element 13: 49
element 15: 17	element 14: 51
element 16: 49	element 15: 61
element 17: 61	element 16: 67
element 18: 68	element 17: 68
element 19: 96	element 18: 78
element 20: 98	element 19: 96
	element 20: 98

1.5.3. Швидка сортування

Складність цього алгоритму в середньому та кращому випадку $O(n * \log n)$, а от при гіршому випадку ми отримаємо $O(n^2)$.

```
def QuickSort(myList, firstInd, lastInd):
    global C
    global E
    i = firstInd
    j = lastInd
    pivotElem = myList[firstInd + (lastInd - firstInd)//2]
    # pivot element in the middle
    while i <= j:
        while myList[i] < pivotElem:
            i += 1
            C+=1
        while myList[j] > pivotElem:
            j -= 1
            C+=1
        C+=1
        if i <= j: # swap
            E+=1
            myList[i], myList[j] = myList[j], myList[i]
            i += 1
            j -= 1
    if firstInd < j: # sort left part
        QuickSort(myList, firstInd, j)
    if i < lastInd: # sort right part
        QuickSort(myList, i, lastInd)
    print("C={}, E={}".format(C,E))
    return myList
```

```
C=0
E=0
numbers1=generateNum(20)
```

```

print("Original list1:")
printList(numbers1)
QuickSort(numbers1,0,len(numbers1)-1)
print("Sorted list1:")
printList(numbers1)
C=0
E=0

numbers2=sorted(numbers1,reverse=True)
print("Original list2:")
printList(numbers2)
QuickSort(numbers2,0,len(numbers1)-1)
print("Sorted list2:")
printList(numbers2)
C=0
E=0

numbers3=[1,2,3,4,5,6,7,8,9,10]+generateNum(10)
print("Original list3:")
printList(numbers3)
QuickSort(numbers3,0,len(numbers1)-1)
print("Sorted list3:")
printList(numbers3)

```

Результати тестування алгоритму сортування злиттям (таб. 4).

Таблица 4

Початкові дані	Відсортовані
Середній випадок	
Original list1: element 1: 20 element 2: 74 element 3: 65 element 4: 67	C=70, E=24 Sorted list1: element 1: 11 element 2: 17 element 3: 20 element 4: 21

Початкові дані	Відсортовані
element 5: 31	element 5: 24
element 6: 79	element 6: 28
element 7: 11	element 7: 29
element 8: 63	element 8: 31
element 9: 28	element 9: 40
element 10: 77	element 10: 46
element 11: 46	element 11: 63
element 12: 71	element 12: 65
element 13: 21	element 13: 66
element 14: 99	element 14: 67
element 15: 29	element 15: 71
element 16: 40	element 16: 74
element 17: 66	element 17: 77
element 18: 17	element 18: 79
element 19: 94	element 19: 94
element 20: 24	element 20: 99

Найгірший випадок

Original list2:	C=60, E=22
element 1: 99	Sorted list2:
element 2: 94	element 1: 11
element 3: 79	element 2: 17
element 4: 77	element 3: 20
element 5: 74	element 4: 21
element 6: 71	element 5: 24
element 7: 67	element 6: 28
element 8: 66	element 7: 29
element 9: 65	element 8: 31
element 10: 63	element 9: 40
element 11: 46	element 10: 46
element 12: 40	element 11: 63
element 13: 31	element 12: 65
element 14: 29	element 13: 66
element 15: 28	element 14: 67
element 16: 24	element 15: 71
element 17: 21	element 16: 74
element 18: 20	element 17: 77
element 19: 17	element 18: 79
element 20: 11	element 19: 94
	element 20: 99

Початкові дані	Відсортовані
Найкращий випадок	
Original list3: element 1: 1 element 2: 2 element 3: 3 element 4: 4 element 5: 5 element 6: 6 element 7: 7 element 8: 8 element 9: 9 element 10: 10 element 11: 69 element 12: 88 element 13: 25 element 14: 42 element 15: 33 element 16: 72 element 17: 94 element 18: 85 element 19: 74 element 20: 20	C=70, E=16 Sorted list3: element 1: 1 element 2: 2 element 3: 3 element 4: 4 element 5: 5 element 6: 6 element 7: 7 element 8: 8 element 9: 9 element 10: 10 element 11: 20 element 12: 25 element 13: 33 element 14: 42 element 15: 69 element 16: 72 element 17: 74 element 18: 85 element 19: 88 element 20: 94

Таблиця 5. Підсумкові результати порівняння алгоритмів

Алгоритм	Середній випадок		Найгірший випадок		Найкращий випадок	
	C	E	C	E	C	E
Бульбашкою	175	78	190	190	112	20
Шелла	62	95	62	98	62	76
Злиттям	61	88	48	88	45	88
Швидке сортування	70	24	60	22	70	16

2. Пошук

2.1. Лінійний пошук

Завдання пошуку даних є одним із найпоширеніших, і практично у кожному додатку (незалежно від предметної області його використання і засобів реалізації) є функція пошуку. Спочатку визначимося, а в чому полягає завдання пошуку.

Пошук — це процес виявлення у певному наборі даних таких елементів (об'єктів), характеристики яких (одна чи декілька) відповідають критеріям пошуку.

Критерій пошуку — це умова (обмеження), накладена на значення властивостей (характеристик) даних.

Найпростіший приклад: у нас є список логінів користувача і нам потрібно дізнатися, чи є у цьому списку логін «**admin**». Значення критерію пошуку часто називають *ключем пошуку* (тобто у нашому прикладі ключ — це рядок «**admin**»). У цьому випадку критерій пошуку: значення елемента дорівнює «**admin**» (тобто перевіряємо характеристикою елемента є його значення).

Інший приклад: знайти у списку логінів користувачів такі логіни, що містять менше 7 символів. Тут критерієм пошуку буде кількість символів у логіні (довжина логіну).

Існує безліч алгоритмів пошуку, ідеї яких часто залежать від особливостей набору даних, в яких ми проводимо пошук. Набори даних у загальному випадку можна поділити на невпорядковані та впорядковані (відсортовані). Оскільки про невпорядкований набір даних нам

заздалегідь нічого невідомо, то найпоширенішим алгоритмом пошуку в таких колекціях є прямий перебір або лінійний алгоритм пошуку.

Лінійний пошук — це повний послідовний (один за одним) перебір усіх елементів набору даних (наприклад, деякого списку `myList`), при якому кожен елемент набору перевіряється на відповідність критерію пошуку (у найпростіших випадках порівнюється з ключем пошуку `key`).

Починається лінійний пошук із крайнього, найчастіше, лівого елемента набору. Пошук закінчено, якщо:

- потрібний елемент знайдений (такий, що `myList[i]==key`);
- весь список переглянуто (пошук дійшов до кінця списку) і потрібний елемент не знайдено.

Розглянемо з прикладу.

0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
40==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
62==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
90==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
10==17? False							
0	1	2	3	4	5	6	7
40	62	90	10	17	11	80	25
70==17? True							
Stop searching							

Рисунок 24

У нашому прикладі на п'ятому етапі порівняння поточного елемента (виділений жовтим кольором фону) з ключем пошуку (значення 17) потрібний елемент (з індексом 4) було знайдено. Пошук було зупинено на цьому етапі. Результат пошуку — успішний.

Розглянемо ще один приклад. Припустимо, що ключ пошуку — це число 29. Перебравши послідовно усі елементи нашого списку (виконавши кількість операцій порівнянь, що дорівнює кількості елементів у списку) ми не знайшли число 29. У цьому випадку пошук було зупинено за другою з двох умов — пройдено весь список. Результат пошуку — не успішний.

Реалізуємо нашу функцію лінійного пошуку, яка прийматиме два аргументи: список (у якому проходитиме пошук) та ключ (значення елемента, позицію якого потрібно знайти).

Припустимо, що нам потрібно знайти перше входження ключа до нашого списку (тобто у разі збігу елемента з ключем, функція поверне індекс цього елемента і пошук буде зупинено). Якщо пошук неуспішний (ключа у списку немає), то функція поверне значення -1.

```
def LinearSearch(myList, keyItem):  
    for i in range(len(myList)):  
        if myList[i] == keyItem:  
            return i  
    return -1  
  
numbers=[40,62,90,10,17,11,80,25]  
print("Original list: {}".format(numbers))  
  
key1=17
```

```
print("Key element {} is in List in {} position".
      format(key1, LinearSearch(numbers, key1)))

key2=29
print("Key element {} is in List in {} position".
      format(key2, LinearSearch(numbers, key2)))
```

Результати успішного та неуспішного пошуку:

```
Original list: [40, 62, 90, 10, 17, 11, 80, 25]
Key element 17 is in List in 4 position
Key element 29 is in List in -1 position
```

Рисунок 25

Тепер розглянемо ситуацію, при якій у нашому списку є багато елементів зі значеннями, рівними ключу. І ми маємо знати усі їх індекси їх (усі входження ключа до списку).

Для цього створимо додатковий список (спочатку порожній) і при кожному знаходженні ключа у списку заноситимемо його індекс до списку індексів.

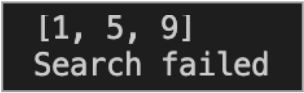
```
def LinearSearchAllKeys(myList, keyItem):
    indList=[]
    for i in range(len(myList)):
        if myList[i] == keyItem:
            indList.append(i)

    return indList

numbers=[40,17,62,90,10,17,11,80,25,17]
key1=17
key2=29
indKey1=LinearSearchAllKeys(numbers, key1)
indKey2=LinearSearchAllKeys(numbers, key2)
```

```
for indKey in (indKey1, indKey2):
    if len (indKey) != 0:
        print(indKey)
    else:
        print("Search failed")
```

Результати для успішного пошуку (ключ 17) і для неуспішного (ключ 29).



[1, 5, 9]
Search failed

Рисунок 26

Даний алгоритм має лише одну перевагу — простота ідеї та реалізації. Однак при великих наборах даних цей алгоритм дуже повільний. І в найгіршому випадку (коли ключ знаходиться у кінці списку) нам потрібно перебрати весь список. Проте, у разі невідсортованого набору даних — це єдиний можливий спосіб реалізації пошуку.

2.2. Бінарний пошук

Такий алгоритм пошуку може використовуватися лише у відсортованих наборах даних. Однак, незважаючи на такі вимоги до даних (навіть у разі їх початкової невпорядкованості, яка вимагатиме виконання сортування перед пошуком), бінарний пошук вважається кращим і найшвидшим алгоритмом.

Алгоритм бінарного пошуку використовує підхід «розділяй і володарюй».

Основний етап бінарного пошуку — це вилучення елемента з середини набору і його перевірка на рівність ключа. Далі, залежно від результатів порівняння, ми забираємо ту чи іншу половину набору з подальшого розгляду. Таким чином, простір пошуку скорочується вдвічі на кожному кроці алгоритму.

При цьому використовувати бінарний пошук виявлення максимального або мінімального значення в наборі немає сенсу, оскільки набір упорядкований і дані елементи знаходяться на початку та в кінці набору відповідно.

Розглянемо кроки бінарного пошуку докладніше. Нехай наш початковий набір даних відсортовано за зростанням.

1. Виконуємо порівняння середнього елемента набору із ключем.
2. Якщо ключ дорівнює середньому елементу, то пошук завершується успішно.
3. Якщо середній елемент більший за ключ, тоді потрібно продовжувати пошук тільки в лівій частині набору (до середнього елемента), інакше — пошук проходить у правій частині, тобто визначаємо нові межі простору пошуку.
4. Кроки 1-3 повторюємо, доки у просторі пошуку є елементи.

Коли ми говоримо про виключення правої або лівої частини із простору пошуку, то, фактично, ми «пересуваємо» ліву або праву межу простору вправо або вліво відповідно.

Спочатку ліва межа (**L**) знаходиться на 0-му елементі списку, а права межа (**R**) — на останньому.

Розглянемо з прикладу.

0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L			m=(L+R)//2			key	R
			12==21? 12<21				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
				L		key	R

Рисунок 27

Після порівняння середнього (3-го елемента, значення 12) з ключем (значення 21) простір пошуку змінився пересуванням лівої межі (L) на позицію $m+1$ ($3+1=4$), тобто наступним кроком ми продовжимо пошук у наборі, починаючи з індексу 4 до 7 включно.

0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
				L		key	R
					m=(L+R)//2		
					18==21? 18<21		
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
						key	R
						L	

Рисунок 28

Отримуємо аналогічну ситуацію (середній елемент менший за ключ) і знову зміщуємо ліву межу. Продовжуємо пошук у просторі з 6-го по 7-й елемент набору включно.

0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
						key	R
						L	
						$m=(L+R)//2$	
						$21==21? \text{ True}$	

Рисунок 29

При цій перевірці середній (6-ий) елемент співпав із ключем. Пошук зупиняється. Результат пошуку успішний.

Тепер розглянемо приклад того, як працюватиме алгоритм, коли ключа в наборі немає (неуспішний пошук)

Нехай набір залишається той самий, а ключ дорівнює 7.

			key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L			$m=(L+R)//2$				R
			$12==7? \text{ 12}>7$				

Рисунок 30

Центральний елемент (12) більший за ключ, тому пересуваємо праву межу ($R=m-1=3-1=2$).

Наступним кроком простір пошуку — це частина списку від 0-го до 2-го елемента включно.

			key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L		R					

Рисунок 31

			key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
L		R					
	5==7? 5<7						

Рисунок 32

Після цього кроку (центральный елемент зі значенням 5 менший за ключ) ми пересуваємо вже ліву межу:

		L	key=7				
0	1	2	3	4	5	6	7
2	5	9	12	17	18	21	32
		R					

Рисунок 33

Отримуємо ситуацію, коли права та ліва межі співпали. Перевіряємо елемент у цій точці — він не дорівнює ключу. Пошук закінчено через відсутність простору пошуку. Результат пошуку невдалий — ключ у наборі не знайдено.

Реалізуємо нашу функцію бінарного пошуку.

```
def BinarySearch(myList, keyItem):
    L=0
    R=len(myList)-1
    keyFound=False

    while (L<=R) and (keyFound==False):
        m=(L+R)//2

        if myList[m]==keyItem:
            keyFound=True
        elif myList[m]>keyItem:
```

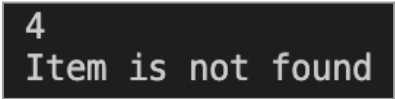
```
        R=m-1
    else:
        L=m+1

    if keyFound:
        return m
    else:
        return -1

numbers=[2,5,9,12,17,18,21,32]
key1=17
key2=29
indKey1=BinarySearch(numbers, key1)
indKey2=BinarySearch(numbers, key2)

for indKey in (indKey1,indKey2):
    if indKey!=-1:
        print(indKey)
    else:
        print("Item is not found")
```

Результат:



```
4
Item is not found
```

Рисунок 34



Урок 5

Сортування, пошук

© STEP IT Academy, www.itstep.org

© Анна Егошина

Усі права на фото-, аудіо- і відеотвори, що охороняються авторським правом і фрагменти яких використані в матеріалі, належать їх законним власникам. Фрагменти творів використовуються в ілюстративних цілях в обсязі, виправданому поставленим завданням, у рамках учбового процесу і в учбових цілях, відповідно до законодавства про вільне використання твору без згоди його автора (або іншої особи, яка має авторське право на цей твір). Обсяг і спосіб цитованих творів відповідає прийнятим нормам, не завдає збитку нормальному використанню об'єктів авторського права і не обмежує законні інтереси автора і правовласників. Цитовані фрагменти творів на момент використання не можуть бути замінені альтернативними аналогами, що не охороняються авторським правом, і відповідають критеріям добросовісного використання і чесного використання.

Усі права захищені. Повне або часткове копіювання матеріалів заборонено. Узгодження використання творів або їх фрагментів здійснюється з авторами і правовласниками. Погоджене використання матеріалів можливе тільки якщо вказано джерело.

Відповідальність за несанкціоноване копіювання і комерційне використання матеріалів визначається чинним законодавством.