АЛГОРИТМИ СОРТУВАННЯ

Лекція 3

Дисципліна «Алгоритми та структури даних»

Типи алгоритмів сортування

Сортування елементів списку - впорядкування їх за деякою ознакою

За замовчуванням сортування відбувається за неспаданням (зростанням)

arr[1] ≤ arr[2] ≤ ... ≤ arr[n]

1. Прості алгоритми. Складність O(n^2)

Ці алгоритми прості для розуміння та реалізації, але є неефективними для великих списків, оскільки їхній час виконання зростає квадратично

Сортування бульбашкою (Bubble Sort) - послідовно переглядає список, порівнюючи сусідні елементи та міняючи їх місцями, якщо вони розташовані в неправильному порядку

Сортування вставками (Insertion Sort) - проходить за списком, беручи кожен елемент і вставляючи його у вже відсортовану частину списку на правильну позицію

Сортування вибором (Selection Sort) - знаходить найменший елемент у невідсортованій частині списку і міняє його місцями з першим елементом цієї частини; процес повторюється для решти списку

Типи алгоритмів сортування

2. Ефективні алгоритми. Складність O(nlogn)

Ці алгоритми значно швидші за прості, особливо для великих наборів даних. Вони використовують складніші підходи, як правило, на основі "розділяй і володарюй".

Сортування злиттям (Merge Sort) - рекурсивно ділить список навпіл, поки не отримає списки з одного елемента. Потім здійснює злиття, сортуючи елементи на кожному кроці. Цей алгоритм є стабільним, але потребує додаткової пам'яті

Швидке сортування (Quick Sort) - вибирає опорний елемент (pivot), розділяє список на дві частини (елементи менші за опорний та більші за нього), а потім рекурсивно застосовує цю ж процедуру до обох частин. Один із найшвидших алгоритмів на практиці

3. Спеціалізовані алгоритми O(n)

Ці алгоритми мають лінійну складність, що робить їх надзвичайно швидкими, але вони можуть працювати лише з певними типами даних, наприклад, з цілими числами в обмеженому діапазоні

Сортування підрахунком (Counting Sort) - створює допоміжний масив, який зберігає кількість входжень кожного елемента. Потім використовує цю інформацію для створення відсортованого вихідного масиву

Бульбашкове сортування (Bubble sort)

Бульбашкове сортування (також, використовується **термін сортування обміном**) є найпростішим, з алгоритмічної точки зору, алгоритмів сортування

Ідея алгоритму полягає у тому, що здійснюється кілька проходів по списку, під час кожного з яких порівнюють пари сусідніх елементів. Якщо елементи стоять не правильно, вони міняються місцями. Кожен прохід по списку ставить наступне найбільше значення на його правильну позицію

Розглянемо кілька проходів бульбашкового сортування для списку елементів:

22	10	15	34	12

Бульбашкове сортування

1-й прохід

22	**10**	15	34	12
10	22	**15**	34	12
10	15	22	34	12
10	15	22	34	**12**
10	15	22	12	**34**

2-й прохід

10	15	22	12	34
10	15	22	12	34
10	15	22	**12**	34
10	15	12	22	**34**
10	15	12	**22**	**34**

3-й прохід

10	15	12	22	34
10	15	**12**	22	34
10	12	15	22	34
10	12	15	22	34
10	**12**	**15**	**22**	**34**

4-й прохід

10	12	15	22	34
10	12	15	22	34
10	12	15	22	34
10	12	15	22	34
10	**12**	**15**	**22**	**34**

Чому потрібен четвертий прохід у бульбашковому сортуванні, якщо масив вже виглядає відсортованим?

Четвертий прохід у бульбашковому сортуванні необхідний для того, щоб **гарантувати**, що масив дійсно відсортований

Для підтвердження сортування потрібен повний прохід без обмінів - четвертий прохід служить саме для цього. Він проходить по всьому масиву, порівнює сусідні елементи, і якщо жодного обміну не відбувається, це означає, що **масив повністю відсортований**, і роботу алгоритму можна завершити

У більшості реалізацій бульбашкового сортування використовується так званий **"прапорець" (flag)**. Це змінна, яка стає **true** на початку кожного проходу. Якщо під час проходу відбувся хоча б один обмін, прапорець стає **false**. Якщо ж після повного проходу прапорець залишився **true**, алгоритм зупиняється, оскільки це означає, що масив вже відсортований і подальші ітерації не потрібні

Реалізація на Python:

```
def bubble_sort(arr):
 n = len(arr)
  # Зовнішній цикл для проходів по масиву
 for i in range(n):
  # Внутрішній цикл для порівняння сусідніх елементів
  # (n-i-1) зменшує кількість порівнянь, оскільки
  # найбільші елементи вже "спливли" в кінець масиву
     for j in range(0, n - i - 1):
      # Якщо елемент більший за наступний, міняємо їх місцями
         if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
return arr
```

Алгоритм використовує **два вкладених цикли**:

Зовнішній цикл проходить по масиву n разів.

Внутрішній цикл також проходить по масиву майже n разів.

Отже, загальна кількість операцій порівняння та обміну приблизно дорівнює $\mathbf{n^2}$

Складність бульбашкового сортування становить $\mathbf{O(n^2)}$, що робить його одним із найповільніших алгоритмів сортування

Сортування вибором (Selection sort)

Алгоритм сортування вибором - це простий, але неефективний алгоритм сортування. Він працює шляхом послідовного знаходження найменшого (або найбільшого) елемента з невідсортованої частини масиву і переміщення його на початок (або кінець) відсортованої частини

22	10	15	34	12
	частина, що залишилася			
10	22	15	34	12
		частина, що залишилася		
10	12	15	34	22
			частина, що залишилася	
10	12	15	34	22
10	12	15	22	34

Сортування вибором. Реалізація на Python

```
def selection_sort(arr):
    n = len(arr)
    # Проходимо по всьому масиву, крім останнього елемента
    for i in range(n - 1):
    # Припускаємо, що поточний елемент є найменшим
        min_pos = i
    # Знаходимо індекс найменшого елемента в
    # невідсортованій частині масиву
        for j in range(i + 1, n):
            if arr[j] < arr[min_pos]:
                min_pos = j
    # Міняємо місцями знайдений найменший елемент
    # з першим елементом невідсортованої частини
        arr[i], arr[min_pos] = arr[min_pos], arr[i]
return arr
```

Як в алгоритмі бульбашкового сортування, при сортування вибором для списку з n елементами, здійснюється $n - 1$ прохід алгоритму по списку, на кожному з яких буде здійснено $n - j$ операцій порівняння, де j – це номер проходу

Хоча за кількістю операцій цей алгоритм оптимальніший за бульбашкове сортування, проте його асимптотична складність така ж, як у бульбашкового сортування – $\mathbf{O(n^2)}$

Сортування вставкою (Insertion sort)

Алгоритм сортування вставкою - це простий алгоритм сортування, який будує фінальний відсортований список по одному елементу за раз. Він працює шляхом взяття елементів з невідсортованої частини масиву і вставки їх у правильну позицію у вже відсортовану частину

Перший елемент списку вважається відсортованою частиною. Далі, кожен наступний елемент, на кожному проході вставляється у відповідну позицію. При цьому, може виникнути ситуація, коли для вставки необхідно зсунути частину елементів списку

Приклад сортування вставкою

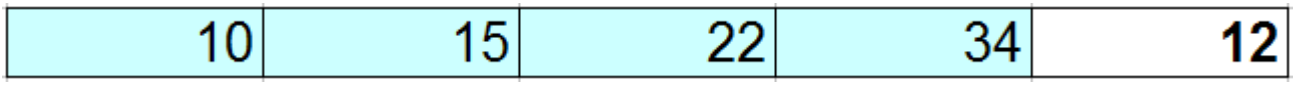
Дано:

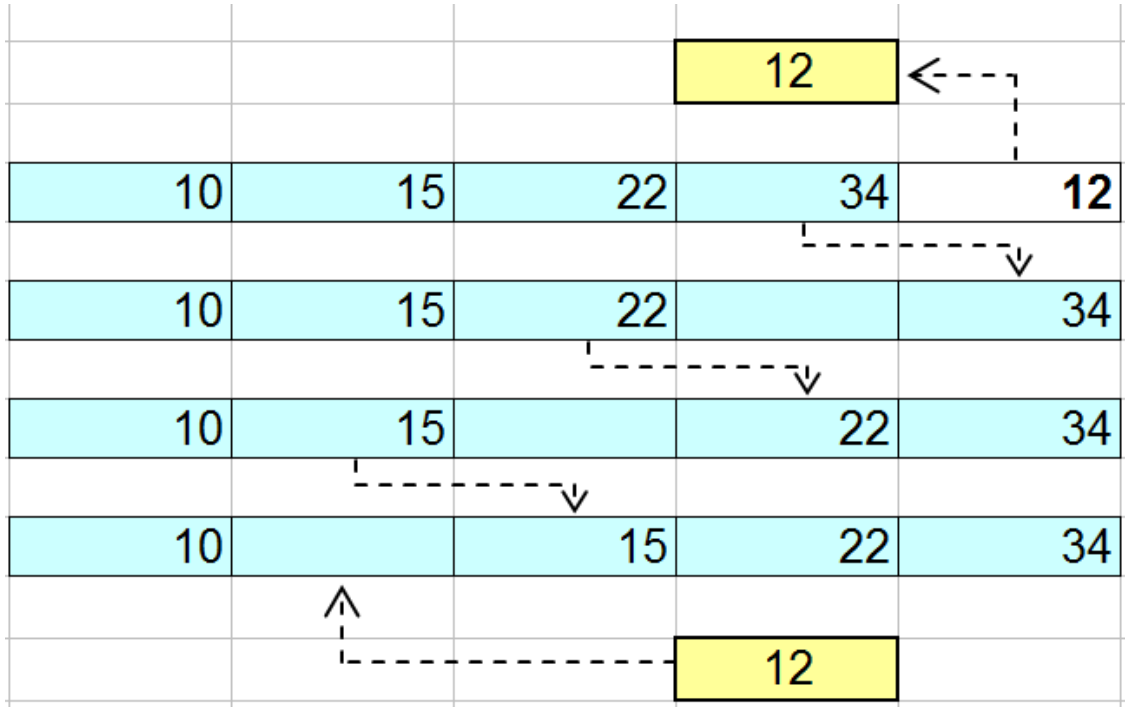
22	10	15	34	12

	22	10	15	34	12
1-й прохід	**10**	22	15	34	12
2-й прохід	10	**15**	22	34	12
3-й прохід	10	15	22	**34**	12
4-й прохід	10	**12**	15	22	34

Вставка елементу на потрібну позицію

Наприклад, розглянемо 4-й прохід алгоритму, а саме стан списку під час вставки елемента 12





Розглядаємо елемент **12**. Порівнюємо його з елементами відсортованої частини справа наліво: **34, 22, 15**
12 < 34, переміщуємо **34** вправо
12 < 22, переміщуємо **22** вправо
12 < 15, переміщуємо **15** вправо
Порівнюємо з **10**: **12 > 10**, зупиняємося
Вставляємо **12** на звільнену позицію

Сортування вставкою. Реалізація на Python

```
def insertion_sort(arr):
 # Проходимо по списку, починаючи з другого елемента (індекс 1)
    for i in range(1, len(arr)):
        # Зберігаємо поточний елемент для вставки
              key = arr[i]
        # Індекс попереднього елемента
              j = i - 1
       # Рухаємо елементи відсортованої частини масиву,
       # які більші за key, на одну позицію вперед
            while j >= 0 and key < arr[j]:
                arr[j + 1] = arr[j]
                j -= 1
       # Вставляємо key на його правильну позицію
        arr[j + 1] = key
    return arr
```

Як і у випадках бульбашкового сортування і сортування вибором, алгоритм здійснює $n - 1$ прохід для списку з n елементів. Якщо частина, у яку вставляється елемент складається з i елементів, то у найгіршому випадку для вставки нового елементу потрібно i операцій зсуву. Отже, асимптотична складність алгоритму буде $O(n^2)$.

Проте, у найкращому випадку, а це буде якщо список вже відсортований, алгоритм буде виконуватися за $O(n)$

Сортування Шелла (більш докладно – самостійно)

Сортування Шелла - це алгоритм сортування, що є узагальненням сортування вставкою

Алгоритм сортування Шелла дозволяє уникати великих зрушень, як у випадку сортування вставкою, коли менше значення знаходиться наприкінці масиву і має бути переміщено в крайню ліву частину

Ідея:

Розбити масив на групи елементів, що знаходяться на певній відстані один від одного, і здійснити незалежне сортування цих груп (як правило, методом вставки). На кожній ітерації крок між елементами групи зменшується і на останній ітерації він дорівнює одиниці. Складність сортування залежить від способу вибору кроку

Цей алгоритм є доволі ефективним для наборів даних середніх розмірів. У найгіршому випадку він має складність **O(n)**

Сортування злиттям (Merge sort)

Сортування злиттям – це рекурсивний алгоритм, який працює за стратегією **«розділяй і володарюй»:**

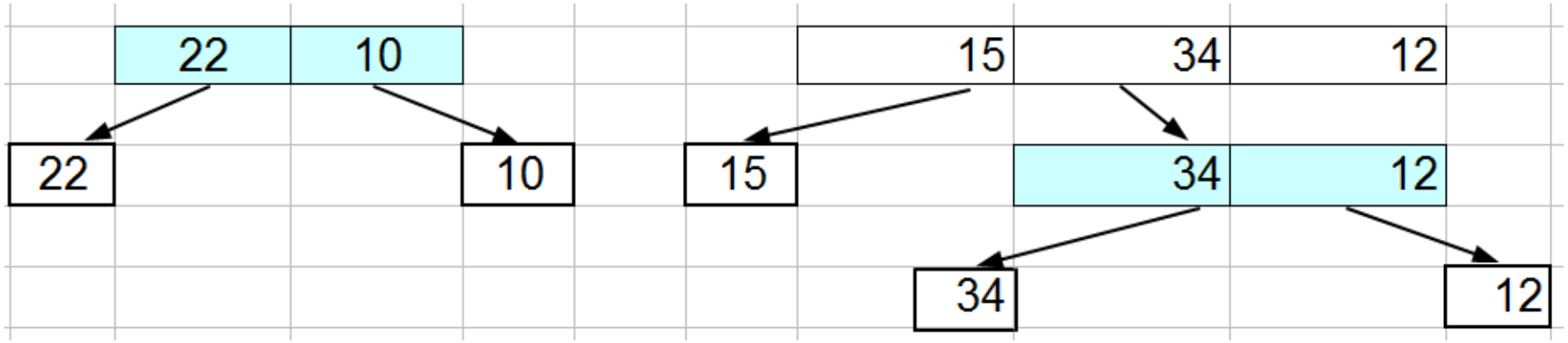
1. Якщо список порожній або складається з одного елементу, то він вважається відсортований

2. Якщо список складається більше ніж з двох елементів, то він розділяється навпіл, після чого для кожної з половин рекурсивно викликається сортування злиттям. Далі два відсортовані списки об’єднуються (зливаються) у один так, щоб утворений список був відсортованим. Рисунок нижче демонструє операцію розбиття списку навпіл

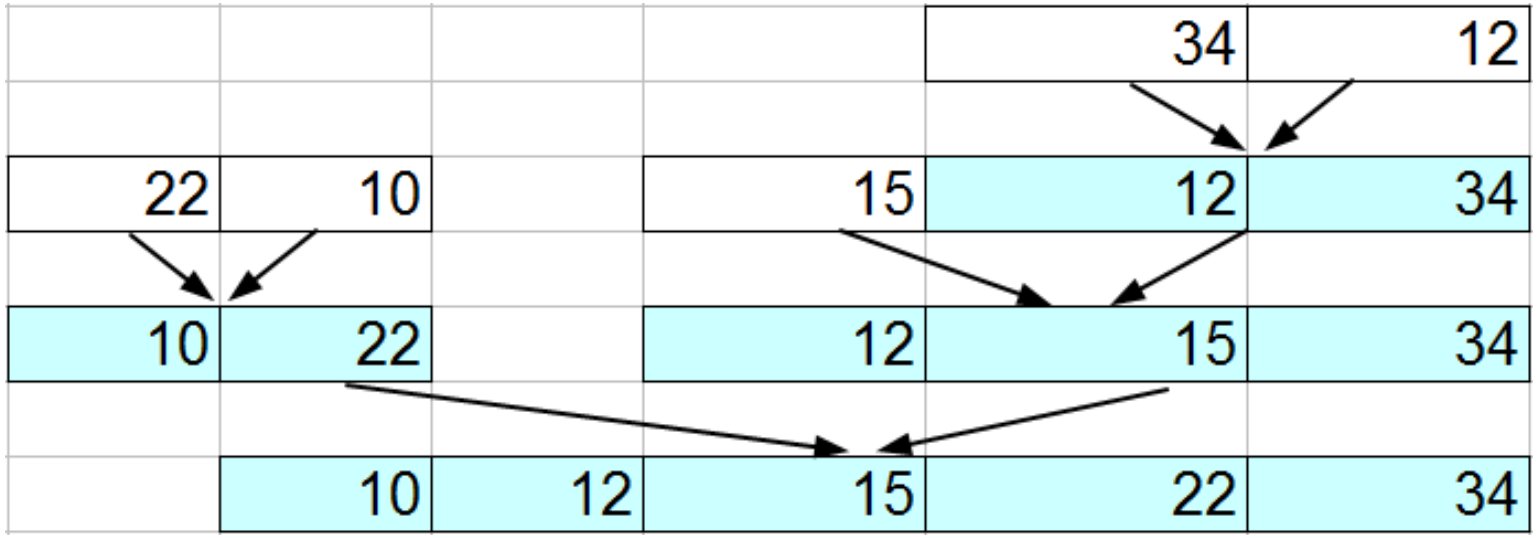
Дано:

22	10	15	34	12

Розділяємо:



Злиття і сортування:



Сортування злиттям. Реалізація на Python

```
def merge_sort(arr):
# Базовий випадок рекурсії: якщо список
# містить 0 або 1 елемент, він вже
# відсортований
    if len(arr) <= 1:
        return arr
# Знаходимо середину масиву
    mid = len(arr) // 2
# Рекурсивно ділимо масив на дві частини
    left_half = arr[:mid]
    right_half = arr[mid:]
    # Рекурсивно сортуємо кожну частину
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)
# Зливаємо відсортовані частини
    return merge(left_half, right_half)
```

```
def merge(left, right):
    result = []
    i = 0  # Індекс для лівого масиву
    j = 0  # Індекс для правого масиву

# Порівнюємо елементи з обох масивів і додаємо
# менший до результату
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
# Додаємо залишки, якщо один з масивів вичерпано
    result.extend(left[i:])
    result.extend(right[j:])
return result
```

Складність алгоритму сортування злиттям

На кожній ітерації рекурсивного алгоритму ділимо список навпіл

Це означає, що для списку з ***n*** елементів буде ***logn*** рекурсивних викликів. Крім цього на кожній ітерації буде здійснюватися операція об’єднання двох списків, асимптотична складність якої є **O(n)**

Складність алгоритму сортування злиттям є O(n log n)

Недоліком алгоритму сортування злиттям є потреба у використанні додаткової пам’яті, яка виникає під час ділення списку навпіл, тобто під час створення двох половинок списку, які використовуються для рекурсивного виклику алгоритму сортування

Швидке сортування (Quick sort)

Алгоритм швидкого сортування, використовується для того, щоб отримати ті ж переваги по швидкості, що і сортування злиттям, не використовуючи при цьому додатку пам’ять

Однією з особливостей алгоритму є те, що він використовує значно меншу кількість порівнянь і перестановок елементів, ніж інші алгоритми

Як і алгоритм сортування злиттям, швидке сортування виконується за час **O(nlogn)**. Проте цей час виконання досягається в середньому, а у окремих випадках, алгоритм швидкого сортування може давати навіть гірші результати за бульбашковий алгоритм або алгоритм вставкою

Розглянемо алгоритм швидкого сортування на прикладі сортування списку:

36	42	17	91	12	25	72	49

Обирається еталонний елемент списку, що називається **опорним елементом (медіаной, pivot)**. Найпростіший спосіб - завжди вибирати перший або останній елемент масиву як опорний:

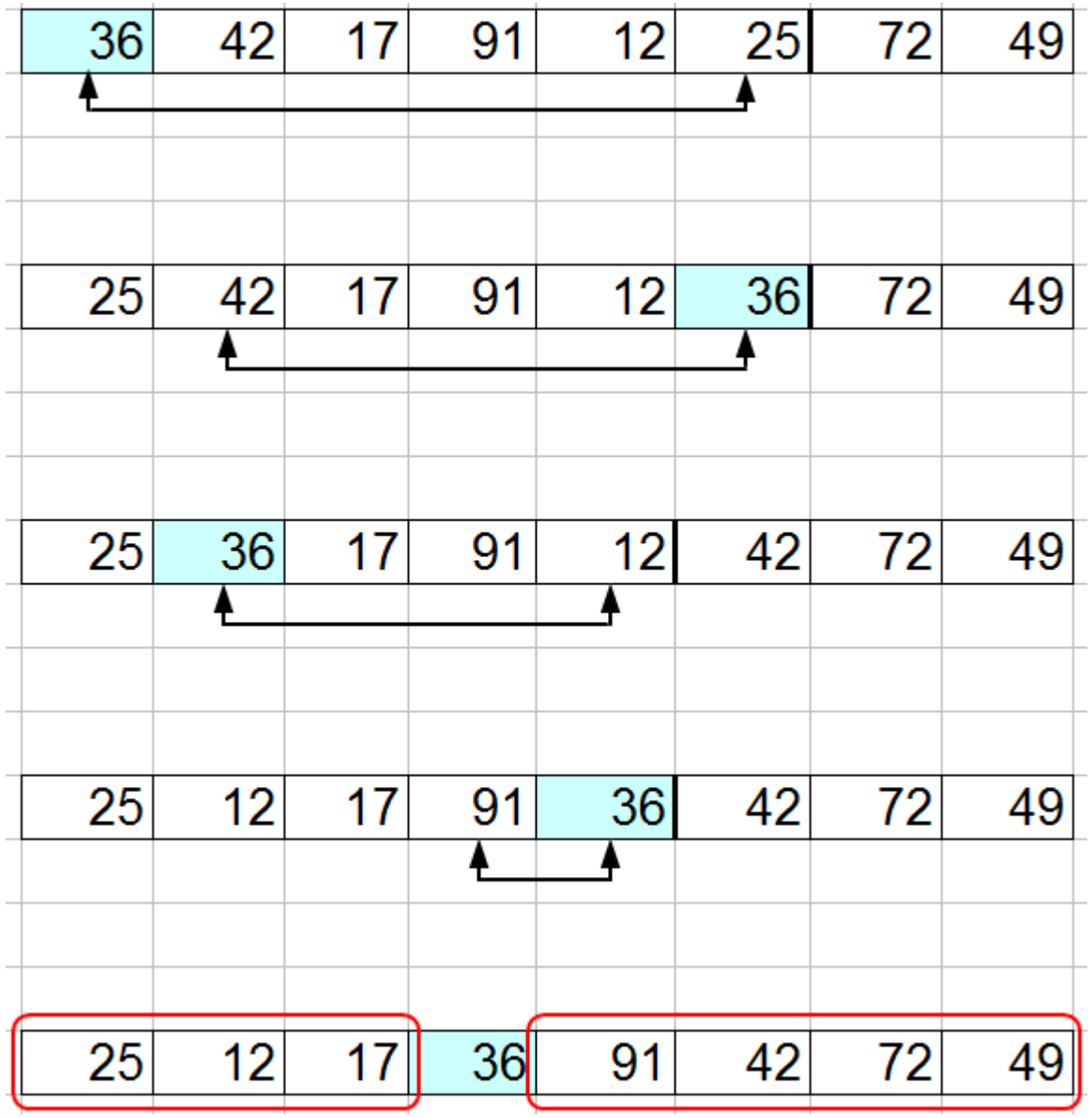
36	42	17	91	12	25	72	49

Опорний елемент **завжди один для кожного конкретного виклику алгоритму** швидкого сортування. Проте **він змінюється для кожного нового рекурсивного виклику**, який відбувається на підсписку

Візьмемо за опорний елемент (медіана) a[1]. Порівняємо його з останнім елементом a[n]. Якщо вони впорядковані один відносно одного, то a[1] порівнюємо з a[n-1], a[n-2], ..., поки не зустрінеться елемент, менший від опорного. Тоді їх треба поміняти місцями

Далі слід порівнювати опорний з наступним елементом, що стоїть за попереднім місцем розташування опорного. Якщо впорядкованість порушується, треба здійснити обмін

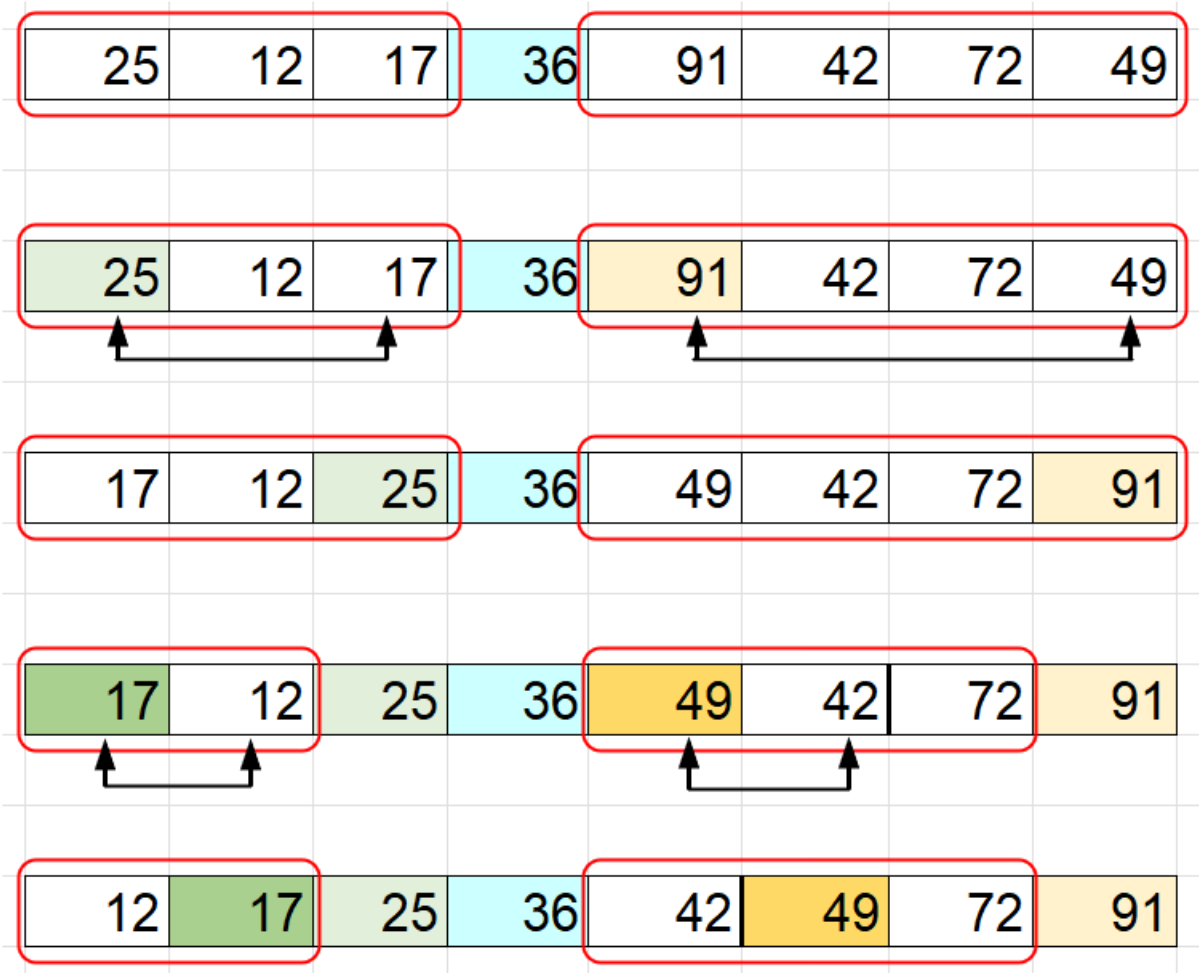
Після того, як опорний порівняно з усіма елементами у списку, він, стрибаючи, стане на своє місце, тобто зліва від опорного стануть всі елементи, менші від нього, а справа - більші. Далі достатньо лише окремо відсортувати праву та ліву частини списку, бо положення опорного елементу вже не мінятиметься



Рекурсія для підсписків

Після розбиття, алгоритм рекурсивно викликається для двох нових, менших підсписків: один з елементами, що менші за опорний елемент, і другий - з тими, що більші

Кожен з цих рекурсивних викликів вибере свій **власний, новий опорний елемент** з відповідного підсписку



Порівняння деяких методів сортування

Алгоритм	Складність	Час сортування (сек.)	Рекомендації до застосування
Бульбашкове сортування	$O(n^2)$	0.0023	Невеликі або майже відсортовані масиви даних
Сортування вибором	$O(n^2)$	0.0018	Невеликі масиви даних
Швидке сортування	$O(n \cdot \log(n))$	0.0006	Великі масиви даних, випадково розподілені
Шейкерне сортування	$O(n^2)$	0.0024	Двостороння обробка даних
Сортування вставками	$O(n^2)$	0.0012	Невеликі або майже відсортовані масиви даних
Сортування злиттям	$O(n \cdot \log(n))$	0.0009	Великі масиви даних. Стабільність сортування