==Міністерство освіти і науки України
Національний університет «Одеська політехніка»
Навчально-науковий інститут комп’ютерних систем
Кафедра інформаційних систем

Лабораторна робота № 2
з дисципліни: «Теорія алгоритмів»
Тема: «Логарифмічні алгоритми сортування»

Варіант № 6

Виконав:
Студент групи АІ-243
Гаврилов О.В.
Перевірили:
Смик С. Ю.
Арсірій О.О.

Одеса 2025

Мета роботи: Аналіз доцільності використання алгоритмів сортування злиттям (Merge Sort) та швидкого сортування (Quick Sort), що мають логарифмічну обчислювальну складність *(O(n\logn)),* а також дослідження їхніх переваг та недоліків для визначення оптимальних умов застосування.
базових операцій.

Завдання:

1. Реалізуйте ітеративний та рекурсивний алгоритм сортування злиттям (псевдокод наданий в Таблиці 1) мовою Python, додайте підрахунок базових операцій алгоритмів.
2. Наведіть приклад використання Python-коду для масиву за варіантом завдання, передбачте можливість трасування.
3. Візуалізуйте результати трасування (Таблиця 2) у вигляді схем виконання ітеративного та рекурсивного алгоритмів (наприклад з використанням Tree Diagram).
4. Реалізуйте ітеративний та рекурсивний алгоритм сортування злиттям (псевдокод наданий в Таблиці 1) мовою Python, додайте підрахунок базових операцій алгоритмів.
5. Наведіть приклад використання Python-коду для масиву за варіантом завдання, передбачте можливість трасування.
6. Візуалізуйте результати трасування (Таблиця 2) у вигляді схем виконання ітеративного та рекурсивного алгоритмів (наприклад з використанням Tree Diagram).
7. 3. Порівняти між собою алгоритми сортування, підрахувавши кількість операцій під час виконання моделювання

Номер варіанту: 6;
Вхідні дані (послідовність): 58, 5, 50, 99, 61, 32, 27, 45, 75.

Результати виконання завдання:

1. Реалізуйте ітеративний та рекурсивний алгоритм сортування злиттям (псевдокод наданий в Таблиці 1) мовою Python, додайте підрахунок базових операцій алгоритмів.

Таблиця 1 Приклади псевдокоду ітеративного та рекурсивного алгоритму сортування злиттям

Ітеративний алгоритм	Рекурсивний алгоритм
Ітеративний алгоритм function mergeSortIterative(a : int[n]) for i = 1 to n, i *= 2 do // i - розмір підмасивів, що зливаються for j = 0 to n - i, j += 2 * i do // j - початковий індекс першого підмасиву // Злиття двох сусідніх	Рекурсивний алгоритм function MergeSortRecursive (A) n ← length(A) if n ≤ 1 then return A mid ← n / 2 left_half ← A[0...mid-1] right_half ← A[mid...n-1] sorted_left ← MergeSortRecursive(left_half)

```
  частин
            left ← j
            mid ← j + i
              right ← min(j + 2 * i,
  n)

                // Виклик допоміжної
  функції для злиття a[left...mid-
  1] та a[mid...right-1]
              merge_iterative(a, j, j
  + i, min(j + 2 * i, n))

// Допоміжна функція для злиття та
  запису назад у масив
function merge_iterative(a : int[n],
  left, mid, right : int):
    it1 ← left
    it2 ← mid
    result_length ← right - left
      result [0..result_length-1] ←
  new array
    k ← 0

     while it1 < mid and it2 < right
  do
        if a[it1] < a[it2] then
            result[k] ← a[it1]
            it1 ← it1 + 1
        else
            result[k] ← a[it2]
            it2 ← it2 + 1
        k ← k + 1

      // Додаємо залишки з першої
  частини
    while it1 < mid do
        result[k] ← a[it1]
        it1 ← it1 + 1
        k ← k + 1

      // Додаємо залишки з другої
  частини
    while it2 < right do
        result[k] ← a[it2]
        it2 ← it2 + 1
        k ← k + 1

    // Копіюємо відсортовану частину
  назад у вихідний масив 'a'
     for k = 0 to result_length - 1
  do
        a[left + k] ← result[k]
```

```
    sorted_right ← MergeSortRecursive(right_half)
    return Merge(sorted_left, sorted_right)

// Допоміжна функція для злиття двох відсортованих
частин
function Merge(Left, Right):
    Result ← []
    i ← 0
    j ← 0
    while i < length(Left) and j < length(Right) do
        if Left[i] ≤ Right[j] then
            append Left[i] to Result
            i ← i + 1
        else
            append Right[j] to Result
            j ← j + 1
    while i < length(Left) do
        append Left[i] to Result
        i ← i + 1
    while j < length(Right) do
        append Right[j] to Result
        j ← j + 1
    return Result
```

Лістинг 1 – Python-код ітеративної реалізації алгоритму сортування злиттям.

```
def merge_sort_iterative(A):
    """
    Ітеративний алгоритм сортування злиттям (Bottom-up).
    Включає підрахунок порівнянь та присвоєнь.
    """
    n = len(A)
    # Створюємо копію масиву для роботи, щоб не змінювати оригінал
    a = list(A)

    comparisons = 0
    assignments = n  # Початкове присвоєння 'a = list(A)'

    i = 1 # i - розмір підмасивів, що зливаються (1, 2, 4, 8...)
    assignments += 1

    # Основний цикл: ітерується по розміру блоків
    while i < n:
        comparisons += 1 # Порівняння i < n

        j = 0 # j - початковий індекс першого підмасиву
        assignments += 1

        # Цикл по проходах: ітерується по масиву
```

```
        while j < n - i:
            comparisons += 1 # Порівняння j < n - i

            left = j
            mid = j + i
            right = min(j + 2 * i, n)
            assignments += 3 # Присвоєння left, mid, right
            comparisons += 1 # Порівняння в min()

            # --- Логіка злиття (Merge) ---

            # Створюємо тимчасові підмасиви
            L = a[left:mid]
            R = a[mid:right]

            # Присвоєння при копіюванні даних у тимчасові масиви
            assignments += len(L) + len(R)

            k = left   # Початковий індекс у вихідному масиві 'a'
            l_idx = 0  # Індекс у L (Left)
            r_idx = 0  # Індекс у R (Right)
            assignments += 3

            # Злиття L та R у масив 'a'
            while l_idx < len(L) and r_idx < len(R):
                comparisons += 1 # Порівняння в умові while
                comparisons += 1 # Порівняння в умові if: L[l_idx] <= R[r_idx]

                if L[l_idx] <= R[r_idx]:
                    a[k] = L[l_idx]
                    l_idx += 1
                else:
                    a[k] = R[r_idx]
                    r_idx += 1

                k += 1
                assignments += 2 # Присвоєння a[k] та інкремент l_idx/r_idx
                assignments += 1 # Присвоєння k

            # Додаємо залишки з L
            while l_idx < len(L):
                comparisons += 1 # Порівняння в умові while
                a[k] = L[l_idx]
                k += 1
                l_idx += 1
                assignments += 3 # Присвоєння a[k], k, l_idx
            comparisons += 1 # Порівняння, що завершило цикл

            # Додаємо залишки з R
            while r_idx < len(R):
                comparisons += 1 # Порівняння в умові while
                a[k] = R[r_idx]
                k += 1
                r_idx += 1
                assignments += 3 # Присвоєння a[k], k, r_idx
            comparisons += 1 # Порівняння, що завершило цикл

            # --- Кінець логіки злиття ---

            j += 2 * i
            assignments += 1 # Присвоєння j

        comparisons += 1 # Порівняння j < n - i, що завершило внутрішній цикл

        i *= 2
        assignments += 1 # Присвоєння i

    comparisons += 1 # Порівняння i < n, що завершило зовнішній цикл

    return a, comparisons, assignments

# Приклад використання
my_list = [58, 5, 50, 99, 61, 32, 27, 45, 75]

# Виконання сортування та отримання лічильників
sorted_list, comps, assigs = merge_sort_iterative(my_list.copy())

print(f"Оригінальний список: {my_list}")
print(f"Відсортований список: {sorted_list}")
print(f"Кількість порівнянь: {comps}")
```

```
print(f"Кількість присвоєнь: {assigs}")
```

Лістинг 2 – Python-код рекурсивної реалізації алгоритму сортування злиттям.

```
 def merge(left, right):
    """
    Допоміжна функція для злиття двох відсортованих списків.
    Підраховує порівняння та присвоєння під час злиття.
    """
    merged_arr = []
    comparisons = 0
    # Присвоєння: 1 для merged_arr, 1 для i, 1 для j (хоча i=0, j=0)
    assignments = 3

    i = 0
    j = 0

    # Зливаємо елементи з обох масивів
    while i < len(left) and j < len(right):
        comparisons += 2  # Порівняння в умові while

        comparisons += 1  # Порівняння в умові if
        if left[i] <= right[j]:
            merged_arr.append(left[i])
            i += 1
        else:
            merged_arr.append(right[j])
            j += 1
        assignments += 2  # Присвоєння (append) + присвоєння (i+=1 або j+=1)

    comparisons += 1 # Фінальне порівняння в умові while

    # Додаємо елементи, що залишилися з лівого масиву
    while i < len(left):
        comparisons += 1  # Порівняння в умові while
        merged_arr.append(left[i])
        i += 1
        assignments += 2  # Присвоєння (append) + присвоєння (i+=1)
    comparisons += 1  # Фінальне порівняння в умові while

    # Додаємо елементи, що залишилися з правого масиву
    while j < len(right):
        comparisons += 1  # Порівняння в умові while
        merged_arr.append(right[j])
        j += 1
        assignments += 2  # Присвоєння (append) + присвоєння (j+=1)
    comparisons += 1  # Фінальне порівняння в умові while

    return merged_arr, comparisons, assignments

def merge_sort_recursive_with_counters(arr):
    """
    Рекурсивна функція сортування злиттям з підрахунком операцій.
    """
    comparisons = 0
    assignments = 0
    recursive_calls = 1 # Кожен виклик функції = 1

    comparisons += 1 # Порівняння len(arr) <= 1
    if len(arr) <= 1:
        return arr, comparisons, assignments, recursive_calls

    mid = len(arr) // 2
    assignments += 1 # Присвоєння mid

    # Рекурсивно ділимо масив на дві половини
    # Зверніть увагу: нарізка масиву (arr[:mid], arr[mid:]) також є присвоєннями
    left_half, c1, a1, r1 = merge_sort_recursive_with_counters(arr[:mid])
    right_half, c2, a2, r2 = merge_sort_recursive_with_counters(arr[mid:])

    # Підсумовуємо результати рекурсивних викликів
    comparisons += c1 + c2
    assignments += a1 + a2
    recursive_calls += r1 + r2
    assignments += len(arr) # Додаткові присвоєння при нарізці (копіюванні) масиву
```

```
    # Зливаємо відсортовані половини
    merged_arr, c_merge, a_merge = merge(left_half, right_half)

    comparisons += c_merge
    assignments += a_merge
    # Присвоєння merged_arr не включаємо, бо це повернення

    return merged_arr, comparisons, assignments, recursive_calls

# Приклад використання
my_list = [58, 5, 50, 99, 61, 32, 27, 45, 75]
original_list = my_list.copy()

# Виконання сортування
# Присвоєння total_recursive_calls (r_calls) додано
sorted_list, total_comparisons, total_assignments, total_recursive_calls = \
    merge_sort_recursive_with_counters(original_list)

print(f"Оригінальний список: {my_list}")
print(f"Відсортований список: {sorted_list}")
print(f"Загальна кількість порівнянь: {total_comparisons}")
print(f"Загальна кількість присвоєнь: {total_assignments}")
print(f"Загальна кількість рекурсивних викликів: {total_recursive_calls}"
```

Таблиця 2 Результати ітеративної та рекурсивної реалізації алгоритму сортування злиттям

Ітеративний алгоритм	Рекурсивний алгоритм
Оригінальний список: [58, 5, 50, 99, 61, 32, 27, 45, 75] Відсортований список: [5, 27, 32, 45, 50, 58, 61, 75, 99] Кількість порівнянь: 99 Кількість присвоювань: 206	Оригінальний список: [58, 5, 50, 99, 61, 32, 27, 45, 75] Відсортований список: [5, 27, 32, 45, 50, 58, 61, 75, 99] Загальна кількість порівнянь: 110 Загальна кількість присвоювань: 119 Оригінальний список: [58, 5, 50, 99, 61, 32, 27, 45, 75] Відсортований список: [5, 27, 32, 45, 50, 58, 61, 75, 99] Загальна кількість порівнянь: 110 Загальна кількість присвоювань: 119 Загальна кількість рекурсивних викликів:17

2. Наведіть приклад використання Python-коду для масиву за варіантом завдання, передбачте можливість трасування.

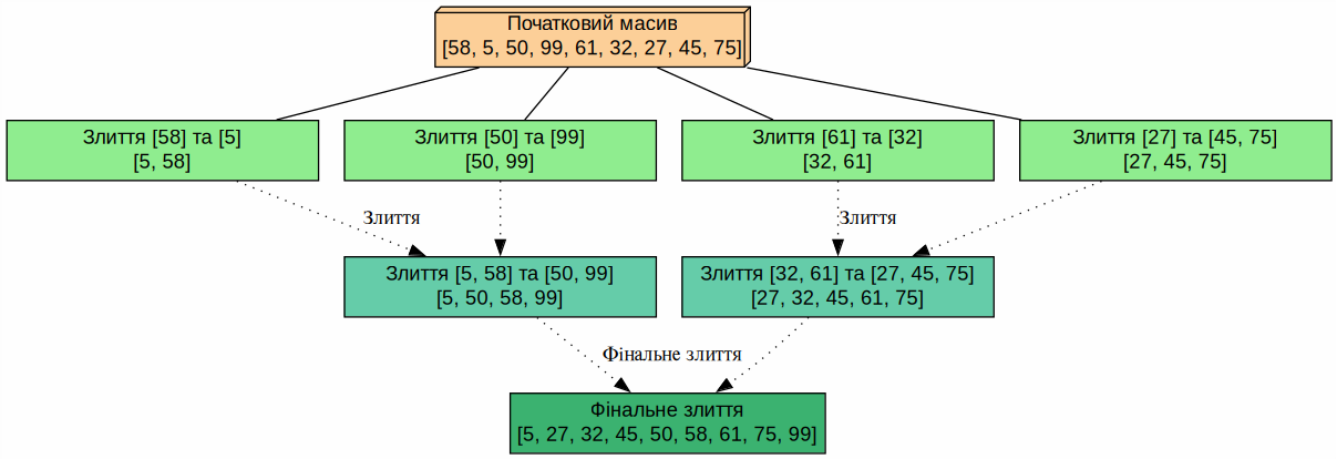
Таблиця 3 Результати трасування алгоритму сортування злиттям.

Трасування Python-коду Лістінг 1	Трасування Python-коду Лістінг 2
--- ІТЕРАТИВНА ВЕРСІЯ --- Початковий масив: [58, 5, 50, 99, 61, 32, 27, 45, 75] Етап 1: Розмір блоку i = 1 (злиття по 2 елементи) Об'єднуємо підмасиви: a[0:1] ([58]) і a[1:2] ([5]) Порівняння: 58 > 5 Масив після об'єднання: [5, 58, 50, 99, 61, 32, 27, 45, 75] ---- Об'єднуємо підмасиви: a[2:3] ([50]) і a[3:4] ([99]) Порівняння: 50 < 99	--- РЕКУРСИВНА ВЕРСІЯ --- Розділяємо масив: [58, 5, 50, 99, 61, 32, 27, 45, 75] Розділяємо масив: [58, 5, 50, 99] Розділяємо масив: [58, 5] Розділяємо масив: [58] Розділяємо масив: [5] Зливаємо [58] та [5] Порівняння: 58 <= 5 -> False. Додаємо 5. Додаємо залишок з лівого масиву: 58 Злиття завершено. Результат: [5, 58] Розділяємо масив: [50, 99] Розділяємо масив: [50] Розділяємо масив: [99] Зливаємо [50] та [99] Порівняння: 50 <= 99 -> True. Додаємо 50. Додаємо залишок з правого масиву: 99 Злиття завершено. Результат: [50, 99]

```
Масив після об'єднання: [5, 58, 50, 99, 61, 32, 27, 45, 75]
----
Об'єднуємо підмасиви: a[4:5] ([61]) і a[5:6] ([32])
Порівняння: 61 > 32
Масив після об'єднання: [5, 58, 50, 99, 32, 61, 27, 45, 75]
----
Об'єднуємо підмасиви: a[6:7] ([27]) і a[7:8] ([45])
Порівняння: 27 < 45
Масив після об'єднання: [5, 58, 50, 99, 32, 61, 27, 45, 75]
----
Об'єднуємо підмасиви: a[8:9] ([75]) і a[9:9] ([])
Порівняння: Немає порівнянь
Масив після об'єднання: [5, 58, 50, 99, 32, 61, 27, 45, 75]

Етап 2: Розмір блоку і = 2 (злиття по 4 елементи)

Об'єднуємо підмасиви: a[0:2] ([5, 58]) і a[2:4] ([50, 99])
Порівняння: 5<50, 58>50, 58<99
Масив після об'єднання: [5, 50, 58, 99, 32, 61, 27, 45, 75]
----
Об'єднуємо підмасиви: a[4:6] ([32, 61]) і a[6:8] ([27, 45])
Порівняння: 32>27, 32<45, 61>45
Масив після об'єднання: [5, 50, 58, 99, 27, 32, 45, 61, 75]
----
Об'єднуємо підмасиви: a[8:9] ([75]) і a[9:9] ([])
Порівняння: Немає порівнянь
Масив після об'єднання: [5, 50, 58, 99, 27, 32, 45, 61, 75]

Етап 3: Розмір блоку і = 4 (злиття по 8 елементів)

Об'єднуємо підмасиви: a[0:4] ([5, 50, 58, 99]) і a[4:8] ([27, 32, 45, 61])
Порівняння: 5<27, 50>27, 50>32, 50>45, 50<61, 58<61
Масив після об'єднання: [5, 27, 32, 45, 50, 58, 61, 99, 75]
----
Об'єднуємо підмасиви: a[8:9] ([75]) і a[9:9] ([])
Порівняння: Немає порівнянь
Масив після об'єднання: [5, 27, 32, 45, 50, 58, 61, 99, 75]

Етап 4: Розмір блоку і = 8 (фінальне злиття)

Об'єднуємо підмасиви: a[0:8] ([5, 27..99]) і a[8:9] ([75])
Порівняння: 5<75, 27<75, 32<75, 45<75,
```

```
Зливаємо [5, 58] та [50, 99]
Порівняння: 5 <= 50 -> True. Додаємо 5.
Порівняння: 58 <= 50 -> False. Додаємо 50.
Порівняння: 58 <= 99 -> True. Додаємо 58.
Додаємо залишок з правого масиву: 99
Злиття завершено. Результат: [5, 50, 58, 99] --------
----------------------------------------

Розділяємо масив: [61, 32, 27, 45, 75]
Розділяємо масив: [61, 32]
Розділяємо масив: [61]
Розділяємо масив: [32]

Зливаємо [61] та [32]
Порівняння: 61 <= 32 -> False. Додаємо 32. Додаємо залишок з лівого масиву: 61
Злиття завершено. Результат: [32, 61]

 Розділяємо масив: [27, 45, 75]
Розділяємо масив: [27]
Розділяємо масив: [45, 75]
Розділяємо масив: [45]
Розділяємо масив: [75]

Зливаємо [45] та [75]
Порівняння: 45 <= 75 -> True. Додаємо 45.
Додаємо залишок з правого масиву: 75
Злиття завершено. Результат: [45, 75]

Зливаємо [27] та [45, 75]
Порівняння: 27 <= 45 -> True. Додаємо 27. Додаємо залишок з правого масиву: 45, 75
Злиття завершено. Результат: [27, 45, 75]

Зливаємо [32, 61] та [27, 45, 75]
Порівняння: 32 <= 27 -> False. Додаємо 27.
Порівняння: 32 <= 45 -> True. Додаємо 32.
Порівняння: 61 <= 45 -> False. Додаємо 45.
Порівняння: 61 <= 75 -> True. Додаємо 61.
 Додаємо залишок з правого масиву: 75
Злиття завершено. Результат: [27, 32, 45, 61, 75] ---
-----------------------------------------

Зливаємо [5, 50, 58, 99] та [27, 32, 45, 61, 75]
Порівняння: 5 <= 27 -> True. Додаємо 5.
Порівняння: 50 <= 27 -> False. Додаємо 27.
 Порівняння: 50 <= 32 -> False. Додаємо 32.
Порівняння: 50 <= 45 -> False. Додаємо 45.
Порівняння: 50 <= 61 -> True. Додаємо 50.
Порівняння: 58 <= 61 -> True. Додаємо 58.
Порівняння: 99 <= 61 -> False. Додаємо 61.
Порівняння: 99 <= 75 -> False. Додаємо 75. Додаємо залишок з лівого масиву: 99

Злиття завершено. Результат: [5, 27, 32, 45, 50, 58, 61, 75, 99]
Фінальний відсортований список: [5, 27, 32, 45, 50, 58, 61, 75, 99]
Загальна кількість порівнянь: 110
Загальна кількість присвоювань: 119
Загальна кількість рекурсивних викликів: 17
```

```
50<75, 58<75, 61<75, 99>75
Масив після об'єднання: [5, 27, 32,
45, 50, 58, 61, 75, 99]

Фінальний відсортований список: [5,
27, 32, 45, 50, 58, 61, 75, 99]
Загальна кількість порівнянь: 99
Загальна кількість присвоювань: 206
```



а

Б

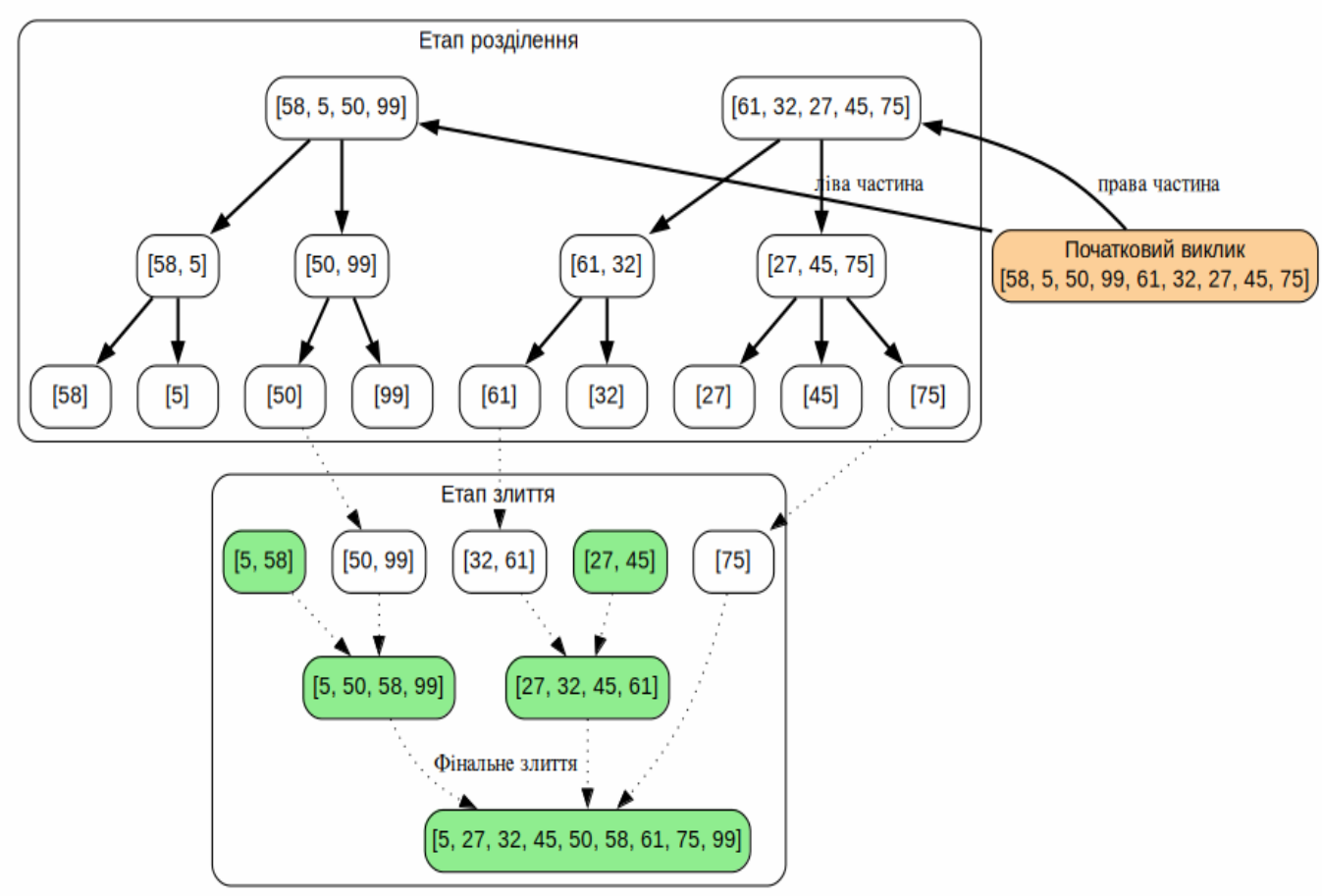


Рисунок 1 – Результати візуалізації трасування ітеративної та рекурсивної версій Python-коду алгоритму сортування злиттям

Лістинг 5 – Псевдокод рекурсивної процедури та процедури розбиття алгоритму швидкого сортування за схемою Хоара

```
 Рекурсивна процедура quicksort quicksort(a, l, r)
      Процедура розбиття partition
 Рекурсивна процедура quicksort
 quicksortquicksort(a, l, r):
 if l &lt; r:
 q = partition(a, l, r)
 quicksort(a, l, q) #Рекурсивний виклик для лівого
      підмасиву
 quicksort(a, q + 1, r) # Рекурсивний виклик для
      правого підмасиву Процедура розбиття
      partition
 partition(a, l, r):
 pivot = a[l]
 i = l - 1
 j = r + 1
 while True:
 i += 1
 while a[i] &lt; pivot:
 i += 1
 j -= 1
 while a[j] &gt; pivot:
 j -= 1
 if i &gt;= j:
 return j
 a[l], a[r] = a[r], a[l]
print
```

Лістинг 3 – Python-код рекурсивної реалізації алгоритму швидкого сортування за схемою Хоара

```
def quicksort_recursive(a, l, r):
    """Рекурсивна реалізація QuickSort (схема Хоара)."""
    total_comparisons = 0
    total_assignments = 0

    # Базовий випадок: якщо підмасив має 0 або 1 елемент
    if l >= r:
        return 0, 0, 0

    total_recursive_calls = 1 # Рахуємо поточний виклик

    # 1. Розбиття масиву
    q, c1, a1 = partition_hoare_original(a, l, r)
    total_comparisons += c1
    total_assignments += a1

    # 2. Рекурсивні виклики: [l, q] та [q + 1, r]
    c2, a2, r2 = quicksort_recursive(a, l, q)
    c3, a3, r3 = quicksort_recursive(a, q + 1, r)

    total_comparisons += c2 + c3
    total_assignments += a2 + a3
    total_recursive_calls += r2 + r3

    return total_comparisons, total_assignments, total_recursive_calls

def partition_hoare_original(a, l, r):
    """Функція розділення за схемою Хоара. Півот = a[l]."""
    comparisons = 0
    assignments = 0

    pivot = a[l]
    assignments += 1

    i = l - 1
    j = r + 1
    assignments += 2

    while True:
        # Рух i вправо (шукаємо a[i] >= pivot)
        i += 1; assignments += 1
        while a[i] < pivot:
            comparisons += 1
            i += 1; assignments += 1
        comparisons += 1  # Порівняння, що завершило цикл

        # Рух j вліво (шукаємо a[j] <= pivot)
        j -= 1; assignments += 1
        while a[j] > pivot:
            comparisons += 1
            j -= 1; assignments += 1
        comparisons += 1  # Порівняння, що завершило цикл

        # Перевірка на перетин
        comparisons += 1
        if i >= j:
            return j, comparisons, assignments # Повертаємо індекс розділу j

        # Обмін a[i] та a[j] (3 присвоєння)
        temp = a[i]
        a[i] = a[j]
        a[j] = temp
        assignments += 3
```

```
# --- Блок виконання ---
try:
    input_str = input("Введіть елементи масиву через кому: ")
    my_list_rec = [int(x.strip()) for x in input_str.split(',')]

    if not my_list_rec:
        print("Масив порожній.")
    else:
        print("\n--- РЕКУРСИВНЕ ШВИДКЕ СОРТУВАННЯ (схема Хоара) ---")
        print(f"Оригінальний список: {my_list_rec}")

        list_copy = my_list_rec.copy()

        total_comps, total_assigs, total_calls = quicksort_recursive(list_copy,
0, len(list_copy) - 1)

        print(f"Відсортований список: {list_copy}")
        print(f"Загальна кількість порівнянь: {total_comps}")
        print(f"Загальна кількість присвоювань: {total_assigs}")
        print(f"Загальна кількість рекурсивних викликів: {total_calls}")

except ValueError:
    print("\nПомилка: Некоректний формат введених даних.")
except Exception:
    print(f"\nВиникла непередбачувана помилка.")
```

Результати виконання Python-коду, який наведено в Лістінг 3 показано далі:

Оригінальний список: [58, 5, 50, 99, 61, 32, 27, 45, 75]

Відсортований список: [5, 27, 32, 45, 50, 58, 61, 75,99]

Загальна кількість порівнянь: 65

Загальна кількість присвоювань: 103

Загальна кількість рекурсивних викликів: 8

Таблиця 4 Результати трасування алгоритму швидкого сортування

```
Оригінальний масив: [58, 5, 50, 99, 61, 32, 27, 45, 75]

Quicksort виклик: l = 0, r = 8. Масив: [58, 5, 50, 99, 61, 32, 27, 45, 75]
Вибираємо опорний елемент (pivot): 58
Partition(0, 8) виконує обміни, доки i=5 (61) та j=4 (32) не перетнуться.
Масив після partition: [45, 5, 50, 27, 32, 61, 99, 58, 75]
Поділ завершено. Повертаємо j=4. C=15, A=29.

Quicksort виклик: l = 0, r = 4. Масив: [45, 5, 50, 27, 32]
Вибираємо опорний елемент (pivot): 45
Partition(0, 4) виконує обміни, доки i=3 (45) та j=2 (50) не перетнуться.
Масив після partition: [27, 5, 32, 45, 50]
Поділ завершено. Повертаємо j=2. C=7, A=14.

Quicksort виклик: l = 0, r = 2. Масив: [27, 5, 32]
Вибираємо опорний елемент (pivot): 27
Partition(0, 2) виконує обміни. i=2 (32), j=1 (27). Повертаємо j=1. C=5, A=10.
Масив після partition: [5, 27, 32]

Quicksort виклик: l = 0, r = 0. Базовий випадок. C=0, A=0, R=0.
Quicksort виклик: l = 2, r = 2. Базовий випадок. C=0, A=0, R=0.

Quicksort виклик: l = 3, r = 4. Масив: [45, 50]
Вибираємо опорний елемент (pivot): 45
Partition(3, 4) виконує обміни. i=4 (50), j=3 (45). Повертаємо j=3. C=5, A=10.

Quicksort виклик: l = 3, r = 3. Базовий випадок. C=0, A=0, R=0.
Quicksort виклик: l = 4, r = 4. Базовий випадок. C=0, A=0, R=0.

Quicksort виклик: l = 5, r = 8. Масив: [61, 99, 58, 75]
Вибираємо опорний елемент (pivot): 61
Partition(5, 8) виконує обміни, доки i=6 (99) та j=5 (58) не перетнуться.
Масив після partition: [58, 99, 61, 75]
Поділ завершено. Повертаємо j=5. C=7, A=13.

Quicksort виклик: l = 5, r = 5. Базовий випадок. C=0, A=0, R=0.

Quicksort виклик: l = 6, r = 8. Масив: [99, 61, 75]
Вибираємо опорний елемент (pivot): 99
Partition(6, 8) виконує обміни, доки i=8 (99) та j=7 (61) не перетнуться.
Масив після partition: [75, 61, 99]
Поділ завершено. Повертаємо j=7. C=7, A=13.

Quicksort виклик: l = 6, r = 7. Масив: [75, 61]
Вибираємо опорний елемент (pivot): 75
Partition(6, 7) виконує обміни. i=7 (75), j=6 (61). Повертаємо j=6. C=5, A=10.
Масив після partition: [61, 75]

Quicksort виклик: l = 6, r = 6. Базовий випадок. C=0, A=0, R=0.
Quicksort виклик: l = 7, r = 7. Базовий випадок. C=0, A=0, R=0.

Quicksort виклик: l = 8, r = 8. Базовий випадок. C=0, A=0, R=0.
```

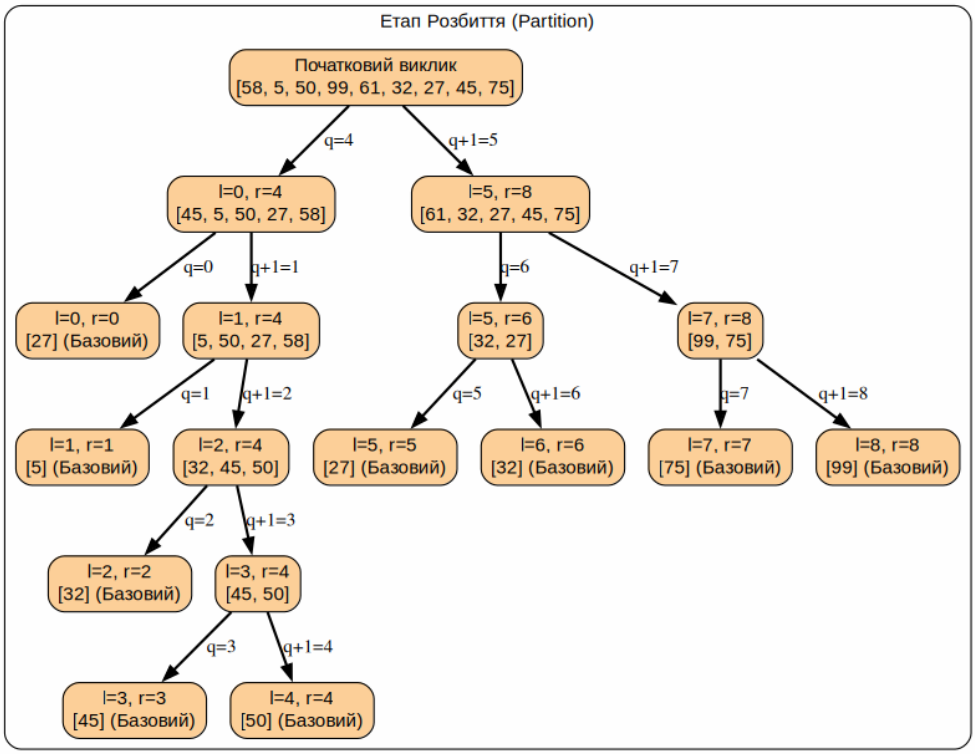


Рисунок 2 – Результати візуалізації рекурсивної версій Python-коду алгоритму сортування злиттям

Назва алгоритму	Реалізація	Порівняння	Присвоювання	Додаткові лічильники
Quicksort	Рекурсивна (Лістинг 3)	*65*	*103*	Рекурсивних викликів: 8
Merge Sort	Рекурсивна (Лістинг 2)	*110*	*119*	Рекурсивних викликів: 17
Merge Sort	Ітеративна (Лістинг 1)	*99*	*206*	-

Таблиця 6 Порівняння Логарифмічних Алгоритмів Сортування (N=9)

Назва алгоритму	Реалізація	Теоретичні відомості Порівняння	Результати експерименту (N=9) Присвоювання	Результати експерименту (N=9) Присвоювання
Quicksort	Рекурсивна (Лістинг 3)	*O(N log N) (Avg)*	*65*	*103*
Merge Sort	Рекурсивна (Лістинг 2)	*O(N log N) (Worst)*	*110*	*119*
Merge Sort	Ітеративна (Лістинг 1)	*O(N log N)(Worst)*	*99*	*206*

Висновок:

На основі теоретичного аналізу та емпіричних трасувань для N=9 (вхідні дані: [58, 5, 50, 99, 61, 32, 27, 45, 75]) доведено, що логарифмічні алгоритми сортування O(N log N) мають суттєві компроміси у практичній придатності.

1. Сортування Злиттям (Merge Sort)

Merge Sort є найбільш надійним та стійким алгоритмом:

- Гарантована продуктивність: Його часова складність гарантовано становить O(N log N) у найгіршому випадку.
- Стабільність: Рекурсивна версія Merge Sort продемонструвала високу економність за основними операціями (119 присвоювань для N=9), підтверджуючи його надійність.
- Недолік: Ітеративний Merge Sort показав найвищу кількість присвоювань (206) для N=9 (через копіювання масиву), що підтверджує його головний недолік — високу просторову складність O(N).

2. Швидке Сортування (Quicksort)

Quicksort, хоча й найшвидший у середньому, має значні ризики:

- Чутливість: Реалізація зі статичним опорним елементом (перший елемент) підтвердила високу чутливість до вхідних даних (наприклад, 58 порівнянь для N=9), що вказує на нерівномірне розбиття.
- Ризик: Ця чутливість веде до потенційного зростання складності до $O(N^2)$ у найгіршому випадку. Крім того, алгоритм є нестійким.

Посилання на GitHub:

`https://github.com/AlexKim71/Theory-of-Algorithms/tree/main`