Лабораторна робота №2

Логарифмічні алгоритми сортування

У роботі розглядаються алгоритми сортування злиттям та швидкого сортування, які мають логарифмічну обчислювальну складність Аналізується доцільність використання цих алгоритмів, їх переваги та недоліки.

Зміст:

1. Сортування злиттям, ітеративна та рекурсивна версії алгоритму

Алгоритм сортування злиттям (*Merge sort*) – стабільний алгоритм сортування, який використовує O(n) додаткової пам'яті та працює за O($n$log(n)) часу.

Алгоритм ґрунтується на парадигмі «розділяй і володарюй» (Divide and Conquer). Його основна ідея полягає в тому, щоб рекурсивно розділяти масив на менші підмасиви доти, доки кожен з них не буде складатися лише з одного елемента, а потім об'єднувати ці відсортовані підмасиви назад у більший, також відсортований масив

Розглянемо основні етапи рекурсивної реалізації алгоритму, псевдокод якого показаний в таблиці 1.

1. Розділення (Divide). Масив ділиться навпіл на два підмасиви. Цей процес триває рекурсивно для кожного підмасиву, поки не отримаємо масиви розміром 1. Масив з одного елемента за визначенням є відсортованим.
2. Злиття (Conquer) Починаючи з найменших підмасивів, алгоритм послідовно зливає (об'єднує) їх у більші відсортовані масиви. Для злиття двох відсортованих підмасивів, елементи порівнюються між собою і записуються в новий, тимчасовий масив у правильному порядку.
3. Повторення. Цей процес злиття повторюється до тих пір, поки всі підмасиви не будуть об'єднані в один великий, повністю відсортований масив.

Крім рекурсивного підходу до реалізації алгоритму сортування злиттям часто використовують ефективний ітеративний підхід. Його використання

дозволяє уникнути рекурсії та пов'язаного з нею навантаження на стек. Основна ідея алгоритму реалізована у вигляді функції mergeSortIterative(a), завдання якої полягає в контролюванні процесу злиття підмасивів, поступово збільшуючи їх розмір.

Розглянемо основні етапи ітеративної реалізації алгоритму сортування злиттям, псевдокод якої показаний в таблиці 1.

1. Зовнішній цикл **for i = 1 to n, i *= 2** відповідає за **розмір підмасивів**, які будуть об'єднуватися на кожному кроці. Цикл починається з i = 1, що означає, що спочатку об'єднуються масиви розміром 1. На кожній ітерації i подвоюється (i *= 2), тому на наступних кроках будуть об'єднуватися масиви розміром 2, потім 4, 8 і так далі, доки i не перевищить довжину масиву n.

2. Внутрішній цикл **for j = 0 to n - i, j += 2 * i** відповідає за **початкові індекси** підмасивів, які будуть об'єднуватися. Цикл починається з j = 0 і збільшується на 2 * i на кожній ітерації. Це дозволяє пересуватися по масиву парами підмасивів, які потрібно злити. Наприклад, якщо i=1, то j буде 0, 2, 4,... (зливаються елементи 0 і 1, 2 і 3, 4 і 5 і т.д.). Якщо i=2, то j буде 0, 4, 8,... (зливаються підмасиви 0-1 і 2-3, 4-5 і 6-7 і т.д.).

3. Виклик допоміжної функції злиття **merge(a, j, j + i, min(j + 2 * i, n))**, де

- a – масив, в якому відбувається злиття.
- j – початковий індекс першого підмасиву.
- j + i – початковий індекс другого підмасиву.
- min(j + 2 * i, n) - кінцевий індекс другого підмасиву. Використання min гарантує, що ми не вийдемо за межі масиву, особливо на останньому кроці.

В таблиці 1 наведено псевдокоди ітеративного та рекурсивного алгоритму сортування злиттям

Розглянемо більш детально дії допоміжних функцій злиття для ітеративного та рекурсивного алгоритму сортування злиттям.

Ітеративний алгоритм. Функція merge_iterative (a; left, mid, right) є частиною, яка виконує безпосереднє злиття двох відсортованих підмасивів (a[left...mid-1] та a[mid...right-1]) в наступний спосіб:

1. Створюється допоміжний масив result розміром right - left. Він буде тимчасово зберігати злиті елементи.
2. За допомогою двох індексів it1 і it2, які вказують на поточний елемент у кожному з підмасивів, алгоритм порівнює елементи a[left + it1] та a[mid + it2]. Менший елемент копіюється в result, а відповідний індекс (it1 або it2) збільшується.
3. Після того, як один з підмасивів повністю скопійований у result, елементи, що залишилися в іншому підмасиві, просто докопіюються в кінець result.
4. Наприкінці, в циклі всі елементи з result копіюються назад в оригінальний масив a у правильні позиції (від left до right).

Рекурсивний алгоритм. Функція Merge(Left, Right) є частиною, яка виконує безпосереднє злиття двох відсортованих підмасивів (Left [] та Right []) в наступний спосіб:

1.Створюється порожній список з назвою Result. Саме в нього будуть додаватися відсортовані елементи, індекс i для першого списку (Left), який вказує на його поточний елемент та індекс j для другого списку (Right), який вказує на його поточний елемент.
2.Основний цикл злиття продовжується, поки в обох списках (Left та Right) є елементи для порівняння. Якщо елемент з Left менший або дорівнює елементу з Right, він додається до Result та індекс i збільшується, щоб перейти до наступного елемента в Left. Якщо елемент з Right менший, він додається до Result, та індекс j збільшується.
3.Після завершення основного циклу один зі списків може містити елементи, які ще не були додані. Тоді цикл додає всі елементи, що залишилися в Left, до Result. Це можливо, оскільки Left вже відсортований.
4. Аналогічно, наступний цикл додає всі елементи, що залишилися в Right, до Result.
5. return Result – Функція повертає новий, повністю відсортований список, який є результатом злиття двох вхідних списків.

Таблиця 1 Приклади псевдокоду ітеративного та рекурсивного алгоритму сортування злиттям

Ітеративний алгоритм	Рекурсивний алгоритм

function **mergeSortIterative(a : int[n])**:
  for i = 1 to n, i *= 2
  for j = 0 to n - i, j += 2 * i
  merge_iterative (a, j, j + i, min(j + 2 * i, n))
	MergeSortRecursive (A)
	n ← length(A)
	if n ≤ 1 then
	return A
	mid ← n / 2
	left_half ← A[0...mid-1] // Розділяємо масив
	right_half ← A[mid...n-1]
	sorted_left ← **MergeSortRecursive (left_half)**
	sorted_right ← **MergeSortRecursive(right_half)**
	return **Merge(sorted_left, sorted_right)** // Зливаємо відсортовані половини

// Допоміжна функція для злиття
function **merge_iterative (a : int[n]; left, mid, right : int):**
it1 = 0
it2 = 0
result : int[right - left]
 	
while left + it1 < mid and mid + it2 < right
if a[left + it1] < a[mid + it2]
result[it1 + it2] = a[left + it1]
it1 += 1
else
result[it1 + it2] = a[mid + it2]
it2 += 1
 	
while left + it1 < mid
result[it1 + it2] = a[left + it1]
it1 += 1
 	
while mid + it2 < right
result[it1 + it2] = a[mid + it2]
it2 += 1
 	
for i = 0 to it1 + it2
a[left + i] = result[i]
	// Допоміжна функція для злиття
	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 // Додаємо елементи, що залишилися
	append Left[i] to Result
	i ← i + 1
	while j < length(Right) do
	append Right[j] to Result
	j ← j + 1
	return Result

2. Реалізація ітеративної та рекурсивної версій алгоритму сортування злиттям мовою Python

Як і попередній лабораторній роботі ускладнимо завдання необхідністю порахувати кількості порівнянь `comparisons` та присвоювань `assignments`, які виконує алгоритм сортування злиттям в ітеративній (Лістинг 2) та рекурсивній (Лістинг 2) реалізації.

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

```
def merge_sort_iterative(a):
    n = len(a)
    comparisons = 0
    assignments = 0
    i = 1
    while i < n:
        j = 0
        while j < n - i:
            left = j
            mid = j + i
            right = min(j + 2 * i, n)
            # Викликаємо допоміжну функцію merge, яка повертає підраховані операції
            c, a_count = merge(a, left, mid, right)
            comparisons += c
            assignments += a_count
            j += 2 * i
        i *= 2
    return a, comparisons, assignments

def merge(a, left, mid, right):
    comparisons = 0
    assignments = 0
    n1 = mid - left
    n2 = right - mid
    # Створюємо тимчасові підмасиви
    L = a[left:mid]
    R = a[mid:right]
    assignments += n1 + n2
    it1 = 0
    it2 = 0
    k = left
    assignments += 2 # присвоєння it1, it2
    assignments += 1 # присвоєння k
    # Зливаємо елементи, порівнюючи їх
    while it1 < n1 and it2 < n2:
        comparisons += 1
        if L[it1] < R[it2]:
            a[k] = L[it1]
            it1 += 1
            assignments += 1
        else:
            a[k] = R[it2]
            it2 += 1
            assignments += 1
        k += 1
        assignments += 1
    # Копіюємо елементи, що залишилися з першого підмасиву
    while it1 < n1:
        a[k] = L[it1]
        it1 += 1
```

```
            k += 1
            assignments += 1
        # Копіюємо елементи, що залишилися з другого підмасиву
        while it2 < n2:
            a[k] = R[it2]
            it2 += 1
            k += 1
            assignments += 1
        return comparisons, assignments

# Приклад використання
my_list = [89, 45, 68, 90, 29, 34, 17]
print("Оригінальний список:", my_list)
sorted_list, comps, assigs = merge_sort_iterative(my_list.copy())
print("Відсортований список:", sorted_list)
print(f"Кількість порівнянь: {comps}")
print(f"Кількість присвоювань: {assigs}")
```

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

```
def merge_sort_recursive(arr):
    comparisons = 0
    assignments = 0
    if len(arr) <= 1:
        return arr, comparisons, assignments
    mid = len(arr) // 2
    assignments += 1
# Підрахунок рекурсивних викликів
    recursive_calls += 2
    # Рекурсивно ділимо масив на дві половини
    # Рекурсивно ділимо масив на дві половини
    left_half, c1, a1, r1 = merge_sort_recursive_with_counters(arr[:mid])
    right_half, c2, a2, r2 = merge_sort_recursive_with_counters(arr[mid:])
    comparisons += c1 + c2
    assignments += a1 + a2
    recursive_calls += r1 + r2
    # Зливаємо відсортовані половини
    merged_arr, c_merge, a_merge = merge(left_half, right_half)
    comparisons += c_merge
    assignments += a_merge
    return merged_arr, comparisons, assignments, recursive_calls

def merge(left, right):
    merged_arr = []
    comparisons = 0
    assignments = 0
    i = 0
    j = 0
    # Зливаємо елементи з обох масивів
    while i < len(left) and j < len(right):
        comparisons += 1
        if left[i] <= right[j]:
            merged_arr.append(left[i])
            i += 1
        else:
            merged_arr.append(right[j])
            j += 1
        assignments += 1
    # Додаємо елементи, що залишилися
    while i < len(left):
        merged_arr.append(left[i])
        i += 1
        assignments += 1
    while j < len(right):
        merged_arr.append(right[j])
```

```
            j += 1
            assignments += 1
    return merged_arr, comparisons, assignments

# Приклад використання
my_list = [89, 45, 68, 90, 29, 34, 17]
print("Оригінальний список:", my_list)

sorted_list, total_comparisons, total_assignments =
merge_sort_recursive(my_list)
print("Відсортований список:", sorted_list)
print(f"Загальна кількість порівнянь: {total_comparisons}")
print(f"Загальна кількість присвоювань: {total_assignments}")
print(f"Загальна кількість рекурсивних викликів: {total_recursive_calls}")
```

Результати виконання Python-коду, який наведено в Лістінг1 та Лістінг2 показано в таблиці 2

Таблиця 2 Результати ітеративної та рекурсивної реалізації алгоритму сортування злиттям

Ітеративний алгоритм	Рекурсивний алгоритм
Оригінальний список: [89, 45, 68, 90, 29, 34, 17] Відсортований список: [17, 29, 34, 45, 68, 89, 90] Кількість порівнянь: 10 Кількість присвоювань: 68	Оригінальний список: [89, 45, 68, 90, 29, 34, 17] Відсортований список: [17, 29, 34, 45, 68, 89, 90] Загальна кількість порівнянь: 14 Загальна кількість присвоювань: 26 Оригінальний список: [89, 45, 68, 90, 29, 34, 17] Відсортований список: [17, 29, 34, 45, 68, 89, 90] Загальна кількість порівнянь: 14 Загальна кількість присвоювань: 40 Загальна кількість рекурсивних викликів: 12

При аналізі рекурсивної версії алгоритму сортування злиттям варто окрім операцій порівнянь та присвоювань підраховувати кількість рекурсивних викликів. При цьому в Лістинг 2 необхідно додати змінну `recursive_calls` (див. червоний колір). При виконанні модифікованої `merge_sort_recursive` коректно підраховано 12 рекурсивних викликів. Загальна кількість викликів для сортування масиву з 7 елементів має становити приблизно $2n-1$ у найгіршому випадку, де n - кількість елементів. Тоді у нашому випадку, $2 \cdot 7 - 1 = 13$. Отриманий результат 12 є дуже близьким до теоретичного. Кількість рекурсивних викликів є важливим показником навантаження на стек, який не використовується в ітеративному алгоритмі. А збільшення кількості присвоювань при підрахунку в модифікованій рекурсивній версії, пов'язано з тим, що було додано операції по підрахунку рекурсивних викликів.

3.Трасування ітеративної та рекурсивної версій Python-коду алгоритму сортування злиттям

Для виконання трасування Python-коду (Лістинг 1 та 2) необхідно самостійно додати команди print у відповідних місцях коду.

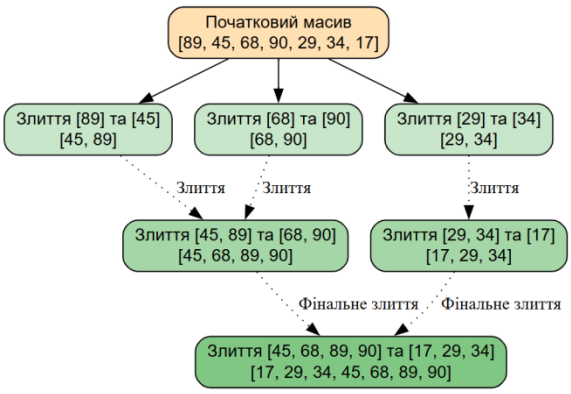
Таблиця 3 Результати трасування алгоритму сортування злиттям.

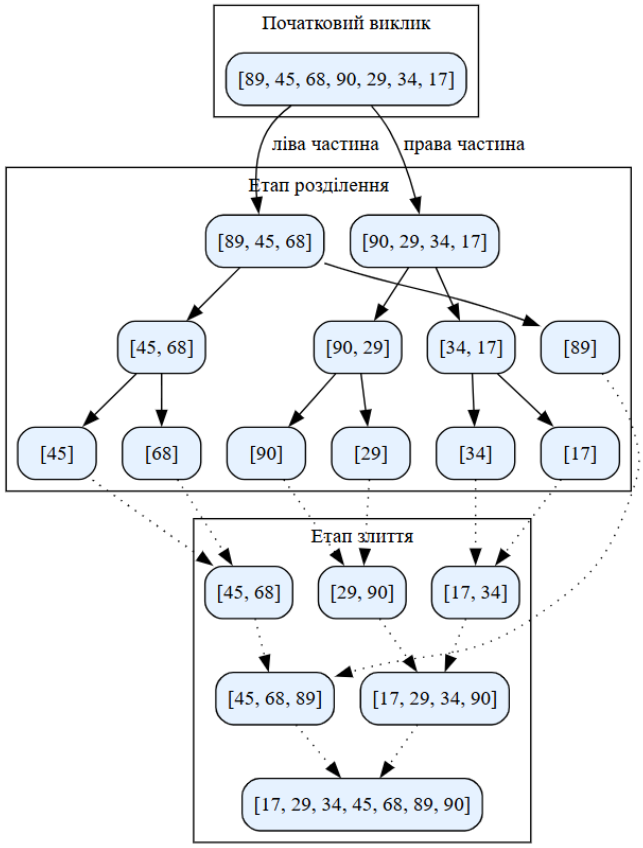
Трасування Python-коду Лістінг 1	Трасування Python-коду Лістінг 2

```
--- ІТЕРАТИВНА ВЕРСІЯ ---
Початковий масив: [89, 45, 68, 90, 29, 34, 17]
--------------------------------------------
----
Об'єднуємо підмасиви: a[0:1] ([89]) і a[1:2] (
[45])
  Порівняння: 89 < 45
Масив після об'єднання: [45, 89, 68, 90, 29, 3
4, 17]
--------------------------------------------
----
Об'єднуємо підмасиви: a[2:3] ([68]) і a[3:4] (
[90])
  Порівняння: 68 < 90
Масив після об'єднання: [45, 89, 68, 90, 29, 3
4, 17]
--------------------------------------------
----
Об'єднуємо підмасиви: a[4:5] ([29]) і a[5:6] (
[34])
  Порівняння: 29 < 34
Масив після об'єднання: [45, 89, 68, 90, 29, 3
4, 17]
--------------------------------------------
----
Об'єднуємо підмасиви: a[0:2] ([45, 89]) і a[2:
4] ([68, 90])
  Порівняння: 45 < 68
  Порівняння: 89 < 68
  Порівняння: 89 < 90
Масив після об'єднання: [45, 68, 89, 90, 29, 3
4, 17]
--------------------------------------------
----
Об'єднуємо підмасиви: a[4:6] ([29, 34]) і a[6:
7] ([17])
  Порівняння: 29 < 17
Масив після об'єднання: [45, 68, 89, 90, 17, 2
9, 34]
--------------------------------------------
----
Об'єднуємо підмасиви: a[0:4] ([45, 68, 89, 90]
) і a[4:7] ([17, 29, 34])
  Порівняння: 45 < 17
  Порівняння: 45 < 29
  Порівняння: 45 < 34
Масив після об'єднання: [17, 29, 34, 45, 68, 8
9, 90]
--------------------------------------------
----
Фінальний відсортований список: [17, 29, 34, 4
5, 68, 89, 90]
Загальна кількість порівнянь: 10
Загальна кількість присвоювань: 98
```

```
--- РЕКУРСИВНА ВЕРСІЯ ---
Розділяємо масив: [89, 45, 68, 90, 29, 34, 17]
  Розділяємо масив: [89, 45, 68]
    Розділяємо масив: [89]
    Розділяємо масив: [45, 68]
      Розділяємо масив: [45]
      Розділяємо масив: [68]
      Зливаємо [45] та [68]
        Порівняння: 45 <= 68 -> True. Додаємо 45
        Додаємо залишок з правого масиву: 68
    Злиття завершено. Результат: [45, 68]
    Зливаємо [89] та [45, 68]
      Порівняння: 89 <= 45 -> False. Додаємо 45
      Порівняння: 89 <= 68 -> False. Додаємо 68
      Додаємо залишок з лівого масиву: 89
  Злиття завершено. Результат: [45, 68, 89]
  Розділяємо масив: [90, 29, 34, 17]
    Розділяємо масив: [90, 29]
      Розділяємо масив: [90]
      Розділяємо масив: [29]
      Зливаємо [90] та [29]
        Порівняння: 90 <= 29 -> False. Додаємо 29
        Додаємо залишок з лівого масиву: 90
    Злиття завершено. Результат: [29, 90]
    Розділяємо масив: [34, 17]
      Розділяємо масив: [34]
      Розділяємо масив: [17]
      Зливаємо [34] та [17]
        Порівняння: 34 <= 17 -> False. Додаємо 17
        Додаємо залишок з лівого масиву: 34
    Злиття завершено. Результат: [17, 34]
    Зливаємо [29, 90] та [17, 34]
      Порівняння: 29 <= 17 -> False. Додаємо 17
      Порівняння: 29 <= 34 -> True. Додаємо 29
      Порівняння: 90 <= 34 -> False. Додаємо 34
      Додаємо залишок з лівого масиву: 90
  Злиття завершено. Результат: [17, 29, 34, 90]
  Зливаємо [45, 68, 89] та [17, 29, 34, 90]
    Порівняння: 45 <= 17 -> False. Додаємо 17
    Порівняння: 45 <= 29 -> False. Додаємо 29
    Порівняння: 45 <= 34 -> False. Додаємо 34
    Порівняння: 45 <= 90 -> True. Додаємо 45
    Порівняння: 68 <= 90 -> True. Додаємо 68
    Порівняння: 89 <= 90 -> True. Додаємо 89
    Додаємо залишок з правого масиву: 90
Злиття завершено. Результат: [17, 29, 34, 45, 68,
89, 90]
Фінальний відсортований список: [17, 29, 34, 45,
68, 89, 90]
Загальна кількість порівнянь: 14
Загальна кількість присвоювань: 40
Загальна кількість рекурсивних викликів: 12
```

Для візуалізаціїї результатів трасування скористайтесь ресурсом Tree Diagram за посиланням https://dreampuf.github.io/GraphvizOnline/?engine. Цей ресурс є особливо корисним візуалізації викликів рекурсивних алгоритмів, таких як сортування злиттям або швидке сортування. Кожен рекурсивний виклик представлений вузлом у дереві, який містить стан масиву. Це допомагає візуалізувати розбиття та злиття. На рисунку 1 показано результати візуалізації трасування ітеративної (Рис. 1,а) та рекурсивної (Рис. 2,б) версій Python-коду алгоритму сортування злиттям. Порівняйте графи на рисунку1 та надайте пояснення стосовно псевдокоду (Тблиця 1), Python-коду (Лістинг 1 та 2) та результатів трасування (Таблиця 3) для ітеративного та рекурсивного алгоритму сортування злиттям





а б

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

2. Швидке сортування

2.1 Рекурсивна версія алгоритму швидкого сортування

Швидке сортування (*quick sort*) чи сортування Хоара – один із найбільш широко використовуваних алгоритмів сортування. Середній час роботи $O(n\log(n))$

В основі швидкого сортування також лежить парадигма «розділяй та володарюй». Але у швидкому сортуванні поділ і сортування відбуваються одночасно.

Розділення. З масиву вибирається *опорний елемент* (pivot). Після цього елементи масиву переставляються таким чином, щоб усі, менші за опорний, знаходилися перед ним, а всі, більші – після нього.

Володарювання (Рекурсія). Алгоритм рекурсивно викликається для двох підмасивів – одного з елементами, меншими за опорний, і іншого – з більшими.

На відміну від сортування злиттям, *етап явного об'єднання відсутній*. Масив вже відсортований, коли завершуються всі рекурсивні виклики.

Більш формально процес розділення та рекурсії представимо у вигляді трьох кроків та псевдокоду в таблиці 4

1. Масив $a[l…r]$ розбивається на два (можливо порожніх) під масиви $a[l…q]$ та $a[q+1…r]$, таких, що кожний елемент $a[l…q]$менше чи дорівнює $a[q]$, який у свою чергу, менше будь-якого елемента підмасиву $a[q+1…r]$. Індекс q обчислюється під час процедури розбиття.

2. Підмасиви $a[l…q]$ та $a[q+1…r]$ сортуються за допомогою рекурсивного виклику процедури швидкого сортування.

3. Оскільки підмасиви *сортуються на місці*, для їхнього об'єднання не потрібні жодні дії: весь масив $a[l…r]$ виявляється відсортованим.

Таблиця 4 Приклади псевдокоду рекурсивної процедури та процедури розбиття алгоритму швидкого сортування за схемою Хоара

Рекурсивна процедура quicksort quicksort(a, l, r)	Процедура розбиття partition

`def quicksort(a, l, r):`
`  if l < 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
	`def partition(a, l, r):`
	`  pivot = a[l]`
	`  i = l - 1`
	`  j = r + 1`
	`  while True:`
	`  i += 1`
	`  while a[i] < pivot:`
	`  i += 1`
	`  j -= 1`
	`  while a[j] > pivot:`
	`  j -= 1`
	`  if i >= j:`
	`  return j`
	`  a[i], a[j] = a[j], a[i]`

Для сортування всього масиву необхідно виконати процедуру *quicksort*(*a*,0,*length*[*a*]−1). Основний крок алгоритму сортування – процедура розбиття `partition(a, l, r)`. Розбиття виконується за схемою Хоара (Hoare partition scheme), яка є фундаментальною частиною алгоритму швидкого сортування (Quicksort). Основна мета цієї функції – переставити елементи в підмасиві від індексу *l* до *r* таким чином, щоб усі елементи, *менші* або рівні за *вибраний опорний* (*pivot*), знаходилися *зліва* від нього, а всі, *більші* або рівні – *справа*. На відміну від розбиття за схемою Ломуто, опорний елемент не обов'язково буде на своїй кінцевій позиції після виконання функції. Розглянемо псевдокод `partition` більш детальною

1. Як опорний елемент вибирається *перший елемент* підмасиву `pivot = a[l]`. Таке припущення є недоліком схеми Хоара оскільки може призвести до найгіршого випадку продуктивності Quicksort, якщо масив вже відсортований.
2. Індекси `i` (лівий) та `j` (правий) ініціалізуються за межами підмасиву `i = l - 1 j = r + 1`. Це дозволяє їм рухатися до центру і правильно знаходити елементи для обміну.
3. Основний цикл (`while True`) – нескінченний цикл, який завершиться тільки тоді, коли індекси `i` та `j` перетнуться.

Пошук зліва `i += 1 while a[i] < pivot: i += 1`

Індекс `i` рухається вправо, доки не знайде елемент, який більший або дорівнює опорному. Цей елемент знаходиться не на своєму місці, оскільки він повинен бути праворуч від опорного.
Пошук справа `j -= 1 while a[j] > pivot: j -= 1`
Індекс `j` рухається вліво, доки не знайде елемент, який менший або дорівнює опорному. Цей елемент також знаходиться не на своєму місці, бо він повинен бути ліворуч.
Перевірка та обмін `if i >= j: return j` Якщо індекси `i` та `j` перетнулися, це означає, що весь підмасив був відсканований, і поділ завершено. Функція повертає індекс `j`, який є точкою, що розділяє масив на два підмасиви для наступних рекурсивних викликів. .
Якщо індекси ще не перетнулися, знайдені елементи міняються місцями `a[i], a[j] = a[j], a[i]`.

2.2 Реалізація алгоритму швидкого сортування мовою Python

При кодуванні алгоритму швидкого сортування за схемою Хоара мовою Python на основі псевдокоду із таблиці 4 ускладнимо завдання необхідністю порахувати кількості порівнянь `comparisons`, присвоювань `assignments` та рекурсивних викликів `recursive_calls` (Лістинг 3).

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

```
def quicksort(a, l, r):
    """
    Рекурсивна реалізація алгоритму швидкого сортування
    з підрахунком операцій, за схемою Хоара.
    """
    comparisons = 0
    assignments = 0
    recursive_calls = 1
    if l < r:
        q, c1, a1 = partition(a, l, r)
        comparisons += c1
        assignments += a1
        # Рекурсивні виклики для лівого та правого підмасивів
        c2, a2, r2 = quicksort(a, l, q)
        c3, a3, r3 = quicksort(a, q + 1, r)
        comparisons += c2 + c3
        assignments += a2 + a3
        recursive_calls += r2 + r3
    else:
        return 0, 0, 0
    return comparisons, assignments, recursive_calls

def partition(a, l, r):
    """
    Допоміжна функція для розділення масиву за схемою Хоара.
    """
    comparisons = 0
    assignments = 0

    pivot = a[l]
    assignments += 1
```

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

    while True:
        # Шукаємо елемент, що більший за опорний
        i += 1
        assignments += 1
        while a[i] < pivot:
            comparisons += 1
            i += 1
            assignments += 1

        # Порівняння для виходу з циклу
        comparisons += 1

        # Шукаємо елемент, що менший за опорний
        j -= 1
        assignments += 1
        while a[j] > pivot:
            comparisons += 1
            j -= 1
            assignments += 1

        # Порівняння для виходу з циклу
        comparisons += 1

        comparisons += 1
        if i >= j:
            return j, comparisons, assignments

        # Обмін елементів
        a[i], a[j] = a[j], a[i]
        assignments += 3

# Приклад використання
my_list = [89, 45, 68, 90, 29, 34, 17]
original_list = my_list.copy()

print("Оригінальний список:", original_list)

total_comparisons,       total_assignments,       total_recursive_calls       =
quicksort(my_list, 0, len(my_list) - 1)

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

Результати виконання Python-коду, який наведено в Лістінг3 показано далі:

Оригінальний список: [89, 45, 68, 90, 29, 34, 17]
Відсортований список: [17, 29, 34, 45, 68, 89, 90]
Загальна кількість порівнянь: 44
Загальна кількість присвоювань: 68
Загальна кількість рекурсивних викликів: 6??

2.3 Трасування рекурсивної версій Python-коду алгоритму швидкого сортування за схемою Хоара

Для виконання трасування Python-коду (Лістінг 3) необхідно самостійно додати команди print у відповідних місцях коду.

Таблиця 5 Результати трасування алгоритму швидкого сортування

Трасування Python-коду (Лістінг 3)

```
--- Quicksort за схемою Хоара з трасуванням ---
Оригінальний список: [89, 45, 68, 90, 29, 34, 17]
Quicksort виклик: масив = [89, 45, 68, 90, 29, 34, 17], l = 0, r = 6
  Вибираємо опорний елемент (pivot): 89
  Поточні індекси: i = 0, j = 6. Обмінюємо a[0] (17) і a[6] (89). Масив: [17,
45, 68, 90, 29, 34, 89]
  Поточні індекси: i = 3, j = 5. Обмінюємо a[3] (34) і a[5] (90). Масив: [17,
45, 68, 34, 29, 90, 89]
  Поточні індекси: i = 5, j = 4. Масив: [17, 45, 68, 34, 29, 90, 89]
  Індекси перетнулися. Поділ завершено. Повертаємо j=4.
  Quicksort виклик: масив = [17, 45, 68, 34, 29, 90, 89], l = 0, r = 4
    Вибираємо опорний елемент (pivot): 17
    Поточні індекси: i = 0, j = 0. Масив: [17, 45, 68, 34, 29]
    Індекси перетнулися. Поділ завершено. Повертаємо j=0.
    Quicksort виклик: масив = [17, 45, 68, 34, 29, 90, 89], l = 0, r = 0
    Quicksort виклик: масив = [17, 45, 68, 34, 29, 90, 89], l = 1, r = 4
      Вибираємо опорний елемент (pivot): 45
      Поточні індекси: i = 1, j = 4. Обмінюємо a[1] (29) і a[4] (45).
Масив: [29, 68, 34, 45]
      Поточні індекси: i = 2, j = 3. Обмінюємо a[2] (34) і a[3] (68).
Масив: [29, 34, 68, 45]
      Поточні індекси: i = 3, j = 2. Масив: [29, 34, 68, 45]
      Індекси перетнулися. Поділ завершено. Повертаємо j=2.
      Quicksort виклик: масив = [17, 29, 34, 68, 45, 90, 89], l = 1, r = 2
        Вибираємо опорний елемент (pivot): 29
        Поточні індекси: i = 1, j = 1. Масив: [29, 34]
        Індекси перетнулися. Поділ завершено. Повертаємо j=1.
        Quicksort виклик: масив = [17, 29, 34, 68, 45, 90, 89], l = 1, r = 1
        Quicksort виклик: масив = [17, 29, 34, 68, 45, 90, 89], l = 2, r = 2
      Після Quicksort для l=1, r=2: масив = [17, 29, 34, 68, 45, 90, 89]
      Quicksort виклик: масив = [17, 29, 34, 68, 45, 90, 89], l = 3, r = 4
        Вибираємо опорний елемент (pivot): 68
        Поточні індекси: i = 3, j = 4. Обмінюємо a[3] (45) і a[4] (68). Масив:
[45, 68]
        Поточні індекси: i = 4, j = 3. Масив: [45, 68]
        Індекси перетнулися. Поділ завершено. Повертаємо j=3.
        Quicksort виклик: масив = [17, 29, 34, 45, 68, 90, 89], l = 3, r = 3
        Quicksort виклик: масив = [17, 29, 34, 45, 68, 90, 89], l = 4, r = 4
      Після Quicksort для l=3, r=4: масив = [17, 29, 34, 45, 68, 90, 89]
    Після Quicksort для l=1, r=4: масив = [17, 29, 34, 45, 68, 90, 89]
  Після Quicksort для l=0, r=4: масив = [17, 29, 34, 45, 68, 90, 89]
  Quicksort виклик: масив = [17, 29, 34, 45, 68, 90, 89], l = 5, r = 6
    Вибираємо опорний елемент (pivot): 90
    Поточні індекси: i = 5, j = 6. Обмінюємо a[5] (89) і a[6] (90). Масив: [89
, 90]
    Поточні індекси: i = 6, j = 5. Масив: [89, 90]
    Індекси перетнулися. Поділ завершено. Повертаємо j=5.
    Quicksort виклик: масив = [17, 29, 34, 45, 68, 89, 90], l = 5, r = 5
    Quicksort виклик: масив = [17, 29, 34, 45, 68, 89, 90], l = 6, r = 6
  Після Quicksort для l=5, r=6: масив = [17, 29, 34, 45, 68, 89, 90]
Після Quicksort для l=0, r=6: масив = [17, 29, 34, 45, 68, 89, 90]

Фінальний відсортований список: [17, 29, 34, 45, 68, 89, 90]
Загальна кількість порівнянь: 44
Загальна кількість присвоювань: 68
Загальна кількість рекурсивних викликів: 6
```

Для візуалізаціїї результатів трасування скористайтесь ресурсом Tree Diagram за посиланням https://dreampuf.github.io/GraphvizOnline/?engine. На рисунку 2 показано результати візуалізації трасування рекурсивної версій Python-коду алгоритму швидкого сортування. Порівняйте граф на рисунку 2 та надайте пояснення стосовно псевдокоду (Тблиця 4), Python-коду (Лістинг 3) та результатів трасування (Таблиця 5) для алгоритму швидкого сортування.

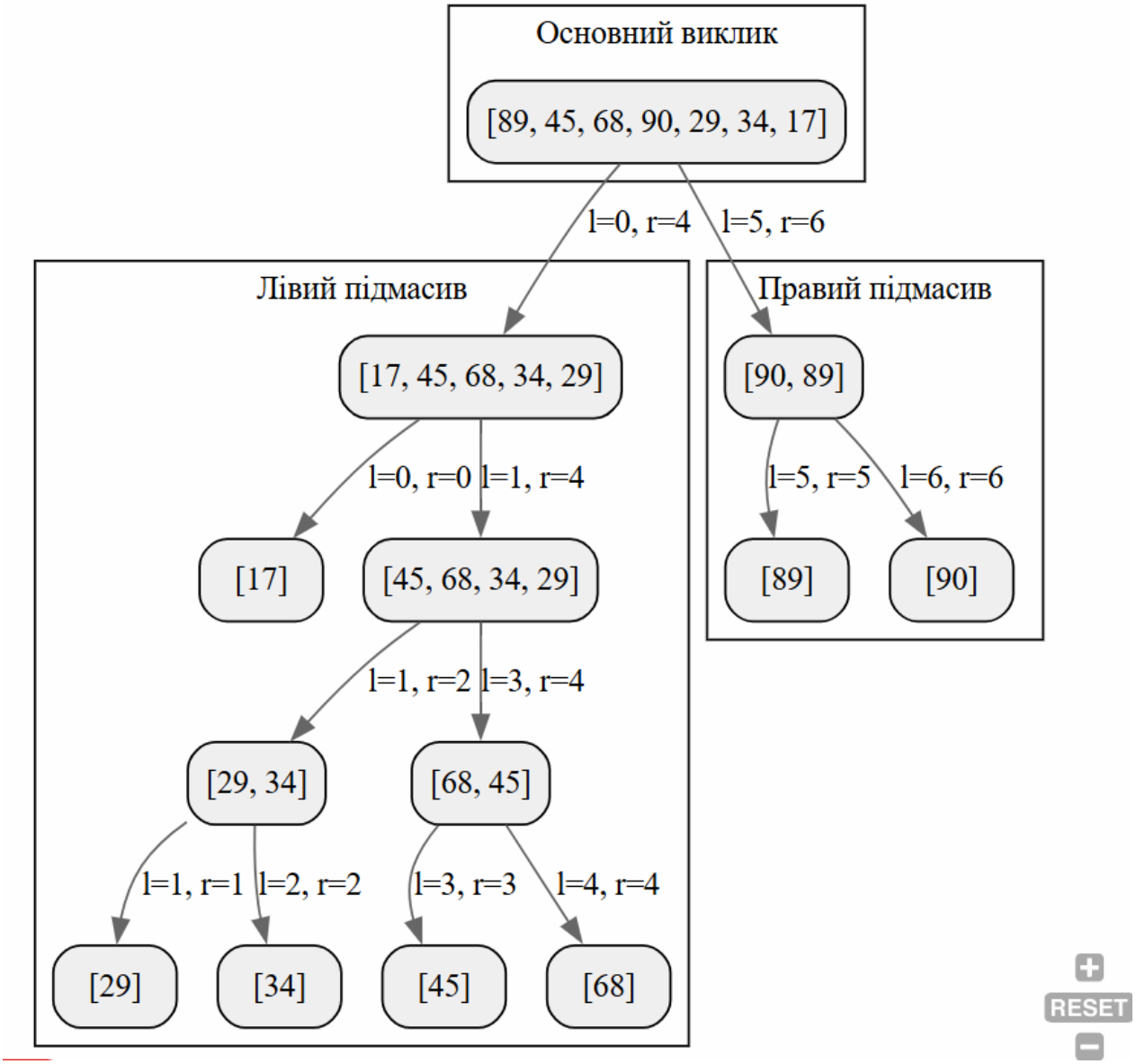


Рисунок 2 – Результати візуалізації трасування Python-коду рекурсивного алгоритму швидкого сортування

3.Порівняння логарифмічних алгоритмів сортування

Стійкість алгоритму сортування означає, що відносний порядок однакових елементів зберігається після сортування. Швидке сортування не може гарантувати цього через те, як воно переставляє елементи відносно **опорного елемента (pivot)**.

Розглянемо приклад: Маємо масив: `[5a, 3, 5b]`, де `5a` і `5b` — це два однакових елементи. Відносний порядок: `5a` іде перед `5b`.

1. Виберемо **опорним елементом** `3`.

2. Під час розділення алгоритм перемістить `5a` і `5b` так, щоб вони були праворуч від опорного елемента `3`.
3. Один з варіантів перестановки може виглядати так: `[3, 5b, 5a]`.
4. Після цієї перестановки, хоча масив ще не відсортовано повністю, **відносний порядок елементів `5a` та `5b` порушено** — `5b` тепер іде перед `5a`.

У класичній реалізації швидкого сортування, коли алгоритм переставляє елементи, менші за опорний, до його лівого боку, а більші — до правого, він не відстежує початковий порядок однакових елементів. Це призводить до потенційного порушення стійкості.

3 Завдання:

Варіанти завдань – співпадають із порядковим номером студента в групі.

Варіант	Послідовність
1	38, 15, 50, 99, 41, 52, 47, 65, 95
2	79, 97, 82, 18, 20, 2, 88, 61, 17
3	68, 97, 12, 15, 31, 40, 22, 50, 53
4	35, 95, 16, 33, 28, 76, 27, 10, 5
5	46, 11, 49, 78, 77, 4, 62, 8, 69
6	58, 5, 50, 99, 61, 32, 27, 45, 75
7	41, 68, 67, 10, 7, 69, 95, 43, 98
8	69, 52, 97, 27, 10, 88,29, 1, 24
9	12, 23, 67, 65, 50, 70, 80, 61, 92
10	87, 79, 97, 82,98, 40, 42, 88, 61
11	47, 50, 61, 41,53, 12, 68, 63, 3
12	53, 100, 44, 74, 53, 38, 82, 65, 28
13	90, 10, 15, 80, 100, 6, 57, 5, 29
14	10, 90, 95, 30, 45, 60, 57, 28, 5
15	50, 80, 19, 86, 35, 7, 60, 48, 51
16	54, 65, 7, 33, 86, 29, 11, 91, 12
17	7, 89, 4, 68, 70, 49, 10, 62, 51
18	21, 44, 22, 50, 63, 68, 97, 12, 15
19	80, 27, 37, 36, 91, 53, 86, 66, 98
20	86, 36, 14, 50, 64, 21, 2, 83, 82
21	50, 57, 78, 34, 41, 68, 47, 61, 38
22	19, 75, 43, 31, 5, 66, 62, 34, 76
23	77, 89, 74, 68, 70, 49, 5, 62, 51
24	53, 5, 44, 47, 35, 83, 82, 85, 28
25	11, 42, 67, 55, 65, 78, 25, 50, 69
26	41, 52, 47, 65, 95, 38, 15, 50, 99
27	18, 20, 2, 88, 61, 17, 79, 97, 82
28	31, 40, 22, 50, 53, 68, 97, 12, 15
29	28, 76, 27, 10, 5, 35, 95, 16, 33

30	78, 77, 4, 62, 8, 69, 46, 11, 49
31	61, 32, 27, 45, 75, 58, 5, 50, 99
32	95, 43, 98, 41, 68, 67, 10, 7, 69
33	29, 1, 24, 69, 52, 97, 27, 10, 88
34	70, 80, 61, 92, 12, 23, 67, 65, 50
35	98, 40, 42, 88, 61, 87, 79, 97, 82
36	53, 12, 68, 63, 3, 47, 50, 61, 41

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

4.Вимоги до представлення звіту

Звіт повинен містити:

1. Титульну частину (лист) з вказівкою на назву лабораторної роботи, ПІБ студента, групу, номер варіанту та послідовність, яку необхідно отсортувати.
2. Псевдокоди сортування злиттям та швидкого сортування, результат їх чисельного моделювання за варіантами завдань
3. Результати порівнянь алгоритмів сортування злиттям та швидкого сортування між собою за кількістю операцій порівняння та присвоювання
4. Висновки

https://dreampuf.github.io/GraphvizOnline/?engine