МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КПІ»

Кафедра інформаційних систем та технологій

Звіт

з комп'ютерного практикуму 7(11)

«Алгоритми сортування. Методи сортування великих обсягів даних»

з дисципліни

«Теорія алгоритмів»

Бригада – 10

Варіант № 1

Перевірила:

ст. вик. Солдатова М.О.

Виконали:

Бойко Катерина,

Гоголь Софія,

Павлова Софія,

Хіврич Володимир

Київ 2022

Комп’ютерний практикум 11

Тема: Алгоритми сортування. Методи сортування великих обсягів даних

Мета роботи: Дослідження та порівняння алгоритмів сортування.

Завдання

Постановка задачі:

Варіант 1:

Король леприконів вирішив навести лад в своїх володіннях. Для цього він доручив помічникам відсортувати скарби в особистій та державній (там їх було дуже багато) скарбницях. Але на цьому він не зупинився та наказав зібрати дані про скарби всіх родин гномів та скласти впорядкований список. При збиранні таких даних з’ясувалося, що багато родин має частково однакові дані про свої скарби. Допоможіть скласти помічникам короля відповідні списки.

Мета задачі – допомогти помічникам короля скласти відсортовані списки скарбів

Використані структури даних: масиви, покажчики, цілочисельні типи, числа з плаваючою точкою, рядки.

Вибір алгоритму

Сортування злиттям

Сортування злиттям (англ. merge sort) — алгоритм сортування, в основі якого лежить принцип «Розділяй та володарюй».

Принцип роботи

В основі цього способу сортування лежить злиття двох упорядкованих ділянок масиву в одну впорядковану ділянку іншого масиву. Злиття двох упорядкованих послідовностей можна порівняти з перебудовою двох колон солдатів, вишикуваних за зростом, в одну, де вони також розташовуються за зростом. Якщо цим процесом керує офіцер, то він порівнює зріст солдатів, перших у своїх колонах і вказує, якому з них треба ставати останнім у нову колону, а кому залишатися першим у своїй. Так він вчиняє, поки одна з колон не вичерпається — тоді решта іншої колони додається до нової.

Під час сортування в дві допоміжні черги з основної поміщаються перші дві відсортовані підпослідовності, які потім зливаються в одну і результат записується в тимчасову чергу. Потім з основної черги беруться наступні дві відсортовані підпослідовності і так доти, доки основна черга не стане порожньою. Після цього послідовність з тимчасової черги переміщається в основну чергу. І знову продовжується сортування злиттям двох відсортованих підпослідовностей. Сортування триватиме доти, доки довжина відсортованої підпослідовності не стане рівною довжині самої послідовності.

Складність алгоритму сортування злиттям за часом:

Найгірший та середній випадок – **O (n log(n))**

Кращий випадок – **O (n log(n))**

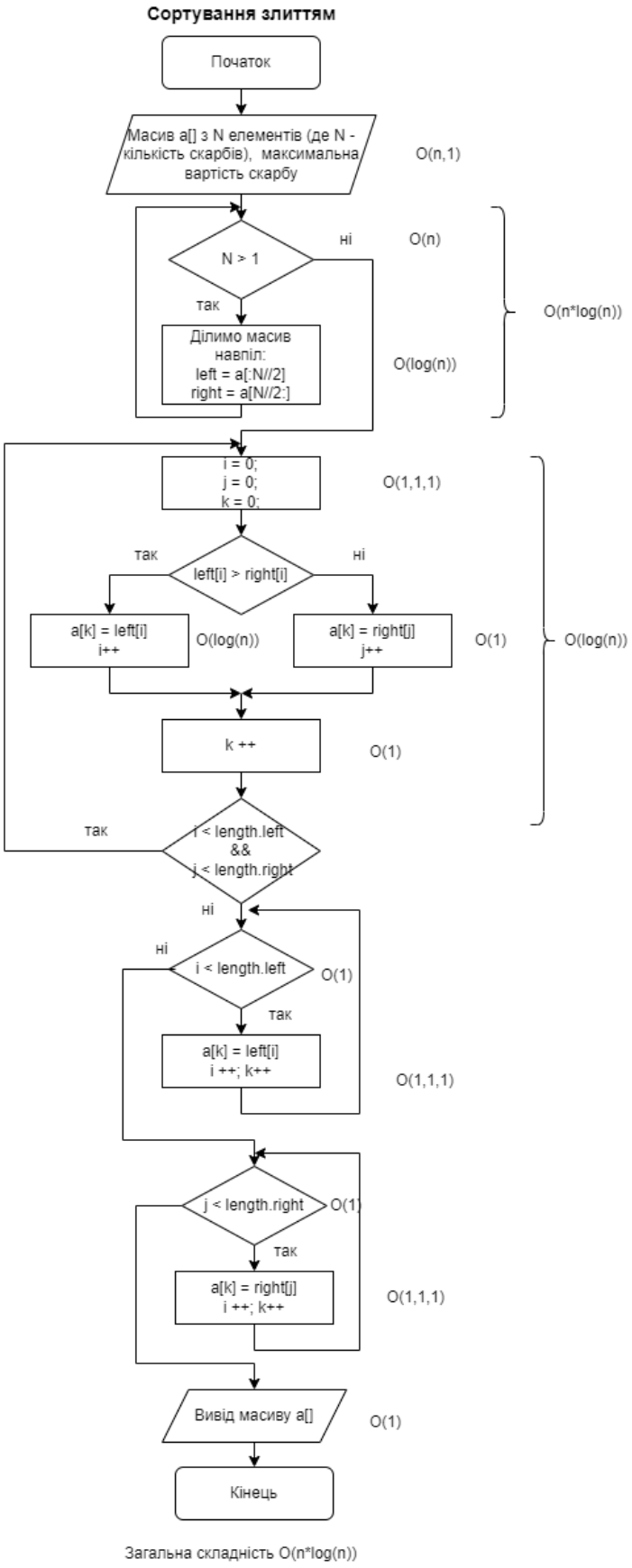


Рис. 1. Блок-схема алгоритму сортування злиттям

Пірамідальне сортування

Пірамідальне сортування (або сортування куп, HeapSort) — це метод сортування порівнянням, заснований на такій структурі даних, як двійкова купа.

Принцип роботи

Цей алгоритм використовує таку структуру даних, яка власне і називається купа. Купа являє собою бінарне дерево, для якого виконується умова, що кожен лист дерева, тобто вузол без нащадків, має глибину d або d-1, де d — це глибина дерева.

Алгоритм складається з трьох кроків:

1. Побудуйте max-heap із вхідних даних.
2. На даному етапі найбільший елемент зберігається в корені купи. Замініть його на останній елемент купи, а потім зменшіть розмір на 1. Нарешті, перетворіть отримане дерево в max-heap з новим коренем.
3. Повторюйте вищезгадані кроки, поки розмір купи більше 1.

Складність алгоритму пірамідального сортування за часом:

Найгірший та середній випадок – **O (n log(n))**

Кращий випадок – **O (n log(n))**

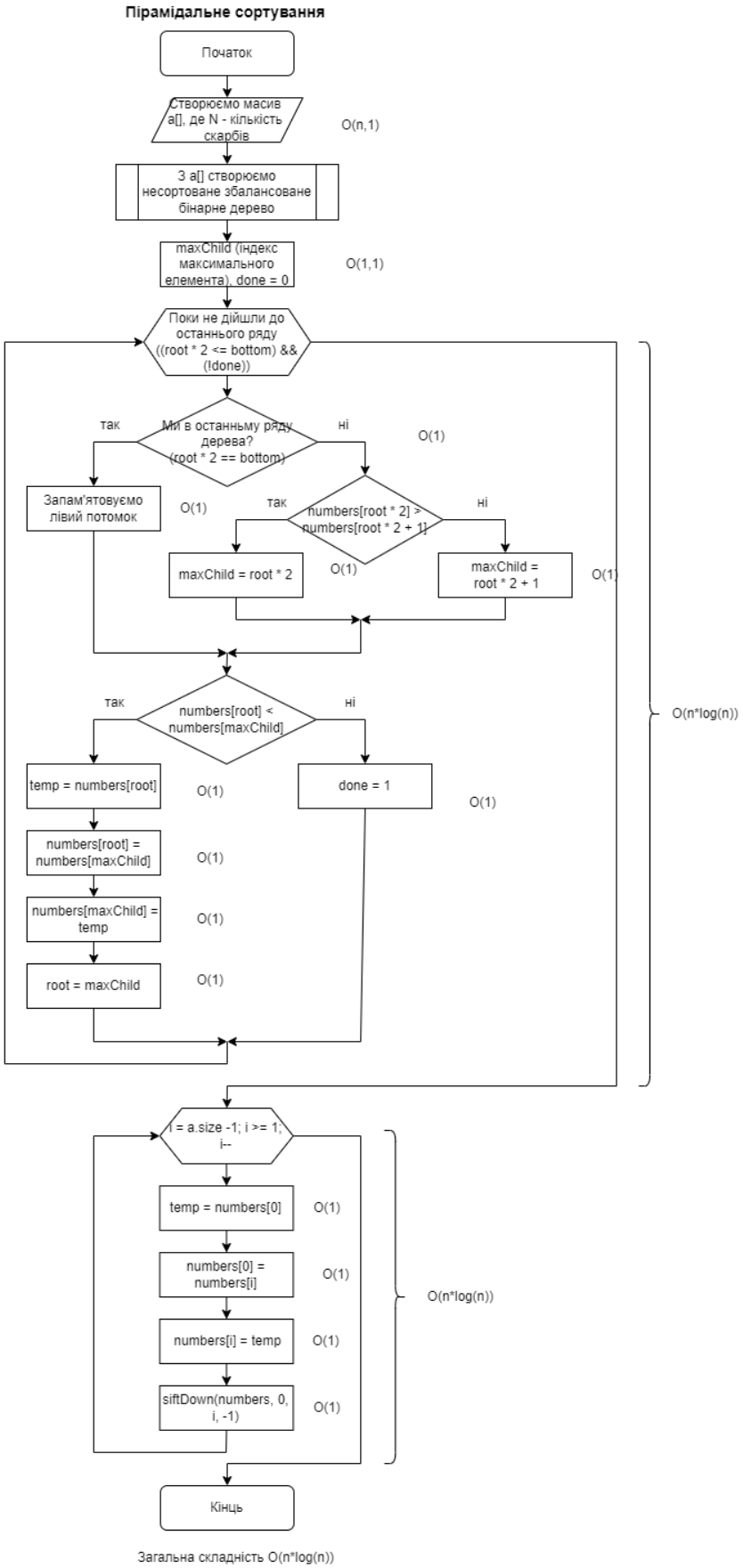


Рис. 2. Блок-схема алгоритму пірамідального сортування

Швидке сортування

Швидке сортування (Quick Sort) — алгоритм сортування, який використовує дуже прості цикли й операції, він працює швидше за інші алгоритми, що мають таку ж асимптотичну оцінку складності. Наприклад, зазвичай більш ніж удвічі швидший у порівнянні з сортуванням злиттям. Швидке сортування не є стабільним.

Принцип роботи

Ідея алгоритму полягає в переставлянні елементів масиву таким чином, щоб його можна було розділити на дві частини та кожний елемент з першої частини був не більший за будь-який елемент з другої. Впорядкування кожної з частин відбувається рекурсивно. Алгоритм швидкого сортування може бути реалізований як у масиві, так і у двозв'язному списку.

Алгоритм складається з трьох кроків:

1. Вибрати елемент з масиву. Має назву опорний.

2. Розбиття: перерозподіл елементів в масиві таким чином, що елементи менше опорного розміщаються перед ним, а більше або рівні після.

3. Рекурсивно застосувати перші два кроки до двох підмасивів зліва і праворуч від опорного елемента. Рекурсія не застосовується до масиву, в якому тільки один елемент або відсутні елементи.

Складність алгоритму швидкого сортування за часом:

Найгірший та середній випадок – **O (n^2)**

Кращий випадок – **O (n log(n))**

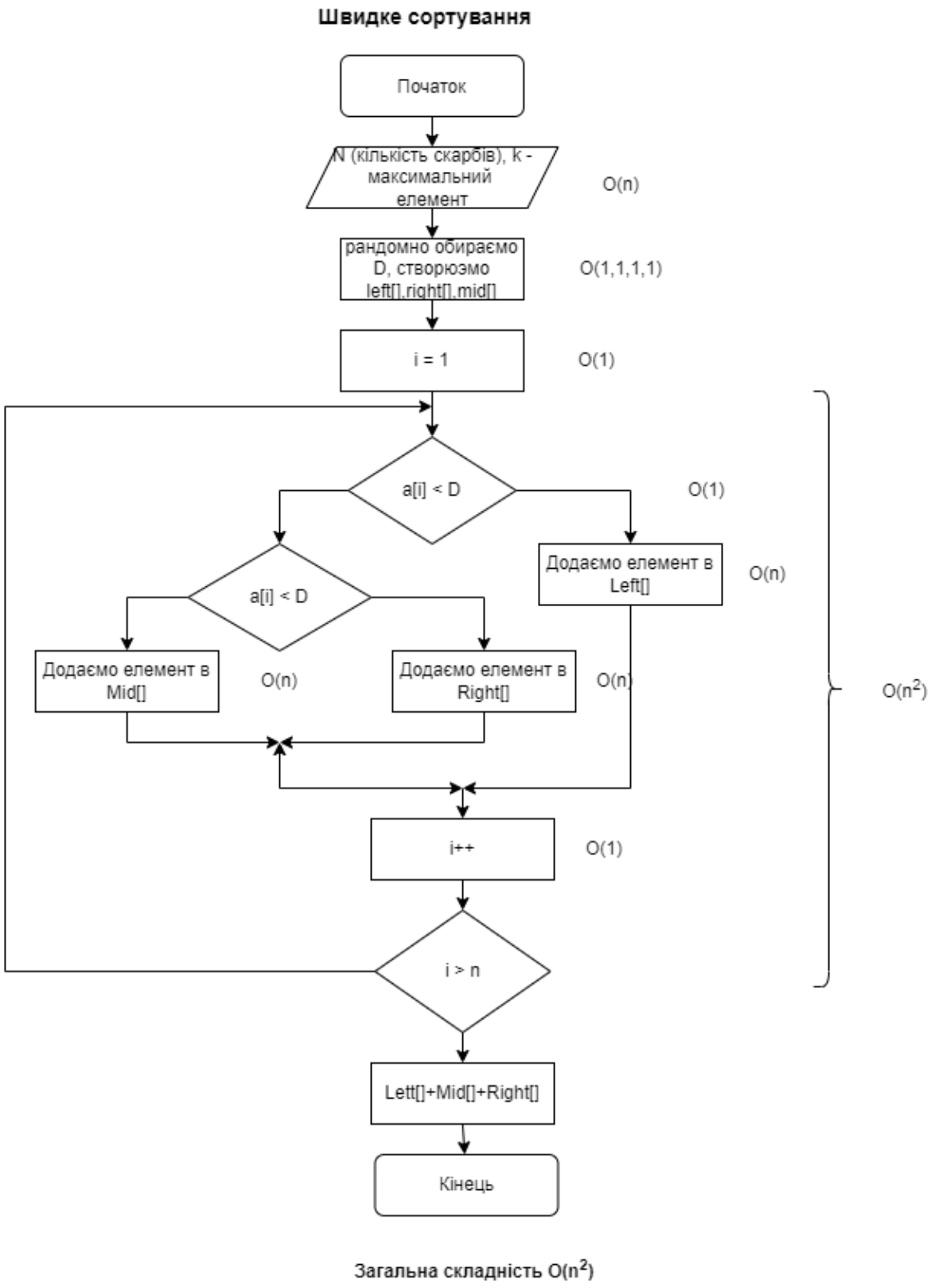


Рис. 3. Блок-схема алгоритму швидкого сортування

Вхідні дані

Назва змінної	Тип змінної	Значення змінної
length0	float	Кількість скарбів гномів королівства/короля
end0	float	Максимальна вартість скарбу
partlySorted0	string	Чи відсортовувати скарби в державній скарбничці
varik	int	Допомагає обрати тип сортування
addLength0	float	Кількість скарбів гномів королівства
var0	string	Обирає, що будемо сортувати (Скарби короля чи всіх родин гномів королівства)

Табл. 1 Таблиця вхідних даних

Модель:

Основні величини:

Назва змінної	Тип змінної	Значення змінної
length	int	Кількість скарбів гномів королівства/короля
end	int	Максимальна кількість золота
addLength	int	Кількість скарбів гномів королівства
partlySorted	int	Чи відсортовувати скарби в державній скарбничці
var	int	Обирає, що будемо сортувати (Скарби короля чи всіх родин гномів королівства)
varik	int	Допомагає обрати алгоритм сортування

Табл. 2. Таблиця основних величин

Допоміжні величини:

Назва змінної	Тип змінної	Значення змінної
subArrayOne	auto const	Розмір лівого підмасиву (сортування злиттям)

subArrayTwo	auto const	Розмір правого підмасиву (сортування злиттям)
leftArray	*auto**	Лівий підмасив
rightArray	*auto**	Правий підмасив
indexOfSubArrayOne	*auto*	Початковий індекс першого підмасиву
indexOfSubArrayTwo	*auto*	Початковий індекс другого підмасиву
indexOfMergedArray	*int*	Початковий індекс об'єднаного масиву
array	*int**	Допоміжний масив
mid	*auto*	Індекс, що знаходиться в середині масиву
largest	*int*	Найбільший елемент масиви (Пірамідальне сортування)
l	*int*	Ліва дитина
r	*int*	Права дитина

Табл 3. Таблиця допоміжних величин

Лістинг програмної реалізації

```
#include <iostream>
#include <windows.h>
#include <ctime>

using namespace std;

// Merges two subarrays of array[].
// First subarray is array[begin..mid]
// Second subarray is array[mid+1..end]
void merge(int array[], int const left, int const mid, int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Create temp arrays
    auto* leftArray = new int[subArrayOne],
            * rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
            leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
            rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne = 0,        // Initial index of first sub-array
            indexOfSubArrayTwo = 0;             // Initial index of second sub-array
    int indexOfMergedArray = left;      // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo <
subArrayTwo)
    {
            if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])
```

```
            {
                array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
                indexOfSubArrayOne++;
            }
            else
            {
                array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
                indexOfSubArrayTwo++;
            }
            indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne)
    {
            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
            indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo)
    {
            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
            indexOfMergedArray++;
    }
}
// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end) return;        // Returns recursively
```

```
    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

// To heapify a subtree rooted with node i which is
// an index in array[]. n is size of heap
void heapify(int array[], int length, int i)
{
    int largest = i;  // Initialize largest as root
    int l = 2 * i + 1;          // left = 2*i + 1
    int r = 2 * i + 2;          // right = 2*i + 2

    // If left child is larger than root
    if (l < length && array[l] > array[largest]) largest = l;
    // If right child is larger than largest so far
    if (r < length && array[r] > array[largest]) largest = r;
    // If largest is not root
    if (largest != i)
    {
            swap(array[i], array[largest]);
            // Recursively heapify the affected sub-tree
            heapify(array, length, largest);
    }
}
// main function to do heap sort
void heapSort(int array[], int length)
{
    // Build heap (rearrange array)
    for (int i = length / 2 - 1; i >= 0; i--)
            heapify(array, length, i);

    // One by one extract an element from heap
    for (int i = length - 1; i >= 0; i--)
```

```
    {
        // Move current root to end
        swap(array[0], array[i]);

        // call max heapify on the reduced heap
        heapify(array, i, 0);
    }
}
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int array[], int low, int high)
{
    int pivot = array[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (array[j] < pivot)
        {
            i++;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[high]);
    return (i + 1);
}
void quickSort(int array[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(array, low, high);
```

```
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
    }
}
void printArray(int array[], int length)
{
    for (auto i = 0; i < length; i++)
            cout << array[i] << " ";
    cout << "\n";
}

void createArray(int array[], int length, int end)
{
    for (int i = 0; i < length; i++)
            array[i] = rand() % end;
}
void addElementsToArray(int array[], int start, int n, int end)
{
    for (int i = start; i < start + n; i++)
            array[i] = rand() % end;
}
int ErorCheck(float a)
{
    if (a <= 0 || (a - int(a) != 0))
    {
            cout << "\n * -------- * ПОМИЛКА * -------- * \n";
            cout << " Упс! Уведено не натуральне число! \n\n";
            return 0;
    }
    else return 1;
}
int FoundError()
{
    cout << "\n    * -------- * ПОМИЛКА * -------- * \n";
    cout << " Неправильний варіант виконання програми!\n";
```

```
    return 0;
}
int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    cout << "\t * -------- * Комп'ютерний практикум 7 * -------- *\n";
    cout << "\t\t\t Бригада 10. Варіант 1  \n";
    cout << "     Реалізація алгоритмів сортування для великих обсягів даних\n\n";

    float length0, end0, addLength0;
    int i, length, end, varik = 0, partlySorted = 0, addLength, var;
    string partlySorted0, var0;

    cout << "\nСортувати скарби короля леприконів чи всіх родин гномів
королівства? (0/1): ";
    cin >> var0;
    if (var0 == "0" || var0 == "1") var = atoi(var0.c_str());
    else return FoundError();

    if (!var) cout << "\nУ короля леприконів усього N скарбів\nN = ";
    else cout << "\nУ родин гномів усього N скарбів\nN = ";
    cin >> length0;
    ErorCheck(length0);
    length = int(length0);
    int* arr = new int[length];
    int* array = new int[length];
    cout << "\nМаксимальна вартість скарбу = ";
    cin >> end0;
    ErorCheck(end0);
    end = int(end0);
    createArray(arr, length, end);

    if (!var)
```

```
    {
        cout << "\nВідсортувати скарби в державній скарбничці? (0/1): ";
        cin >> partlySorted0;
        if (partlySorted0 == "0" || partlySorted0 == "1") partlySorted =
atoi(partlySorted0.c_str());
        else return FoundError();
    }
    if(!var) cout << "\nСкарби короля леприконів: \n";
    else cout << "\nСкарби всіх родин гномів: \n";
    printArray(arr, length);
    cout << "\n----------Менюшка----------";
    cout << "\n0 - Сортування злиттям\n1 - Пірамідальне сортування\n2 - Швидке
сортування";
    cout << "\n----------------------------";
    cout << "\nОберіть алгоритм сортування (0/1/2): ";
    cin >> varik;
    //float start = clock();
    switch (varik)
    {
    case 0:
        if (partlySorted == 0) cout << "\nСОРТУВАННЯ ЗЛИТТЯМ:";
        mergeSort(arr, 0, length - 1);
        break;
    case 1:
        if (partlySorted == 0) cout << "\nПІРАМІДАЛЬНЕ СОРТУВАННЯ:";
        heapSort(arr, length);
        break;
    case 2:
        if (partlySorted == 0) cout << "\nШВИДКЕ СОРТУВАННЯ:";
        quickSort(arr, 0, length - 1);
        break;
    default:
        cout << "\n    * -------- * ПОМИЛКА * -------- * \n";
        cout << " Неправильний варіант виконання програми!\n";
        return 0;
```

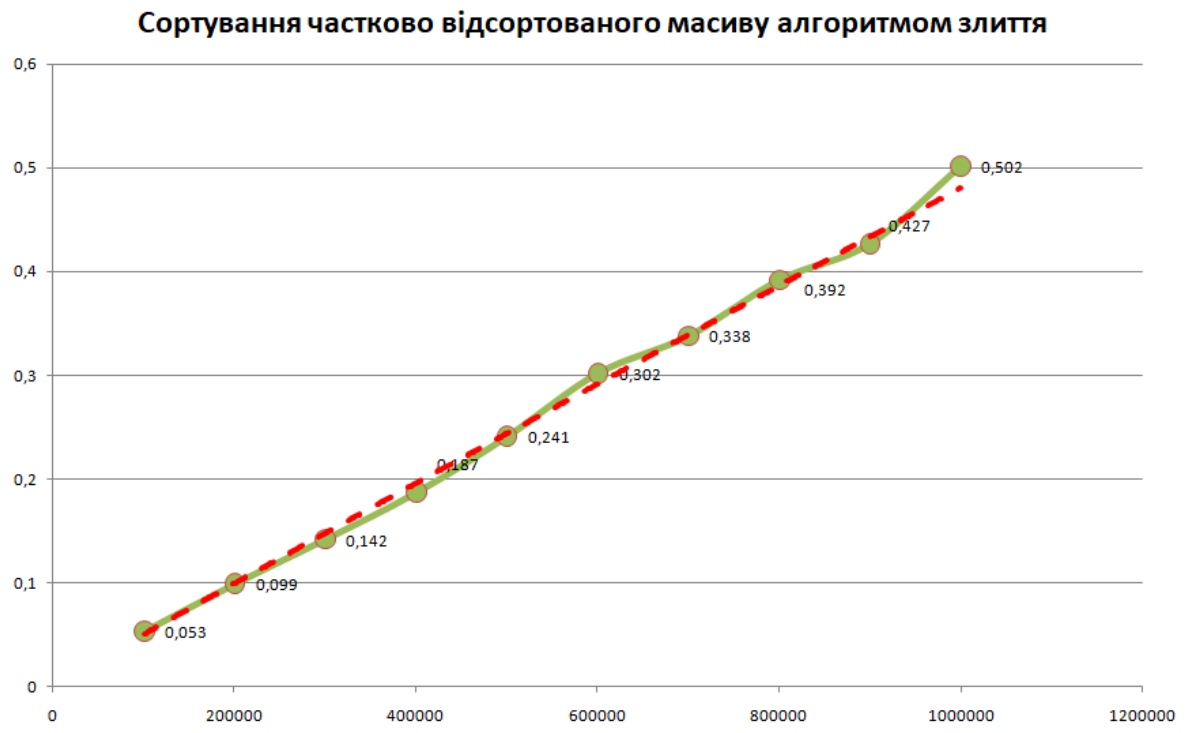
```
}

if (partlySorted == 1)
{
	cout << "\nУ гномів королівства ще N скарбів\nN = ";
	cin >> addLength0;
	ErorCheck(addLength0);
	addLength = int(addLength0);
	int* newArray = new int[length + addLength]();
	for (int i = 0; i < length; i++)
		newArray[i] = arr[i];
	addElementsToArray(newArray, length, addLength, end);
	cout << "\nНовий перелік скарбів: \n";
	printArray(newArray, length + addLength);
	//float start = clock();
	if (varik == 0)
	{
		cout << "\nСОРТУВАННЯ ЗЛИТТЯМ:";
		mergeSort(newArray, 0, length + addLength - 1);
	}
	else if (varik == 1)
	{
		cout << "\nПІРАМІДАЛЬНЕ СОРТУВАННЯ:";
		heapSort(newArray, length + addLength);
	}
	else
	{
		cout << "\nШВИДКЕ СОРТУВАННЯ:";
		quickSort(newArray, 0, length + addLength - 1);
	}
	//cout << (clock() - start) / CLOCKS_PER_SEC;
	cout << "\nПомічники короля надали упорядкований список скарбів: \n";
	printArray(newArray, length + addLength);
}
else
```

```
    {
        //cout << (clock() - start) / CLOCKS_PER_SEC;
        cout << "\nПомічники короля надали упорядкований список скарбів: \n";
        printArray(arr, length);
    }
    cout << "\n\n\n";
}
```

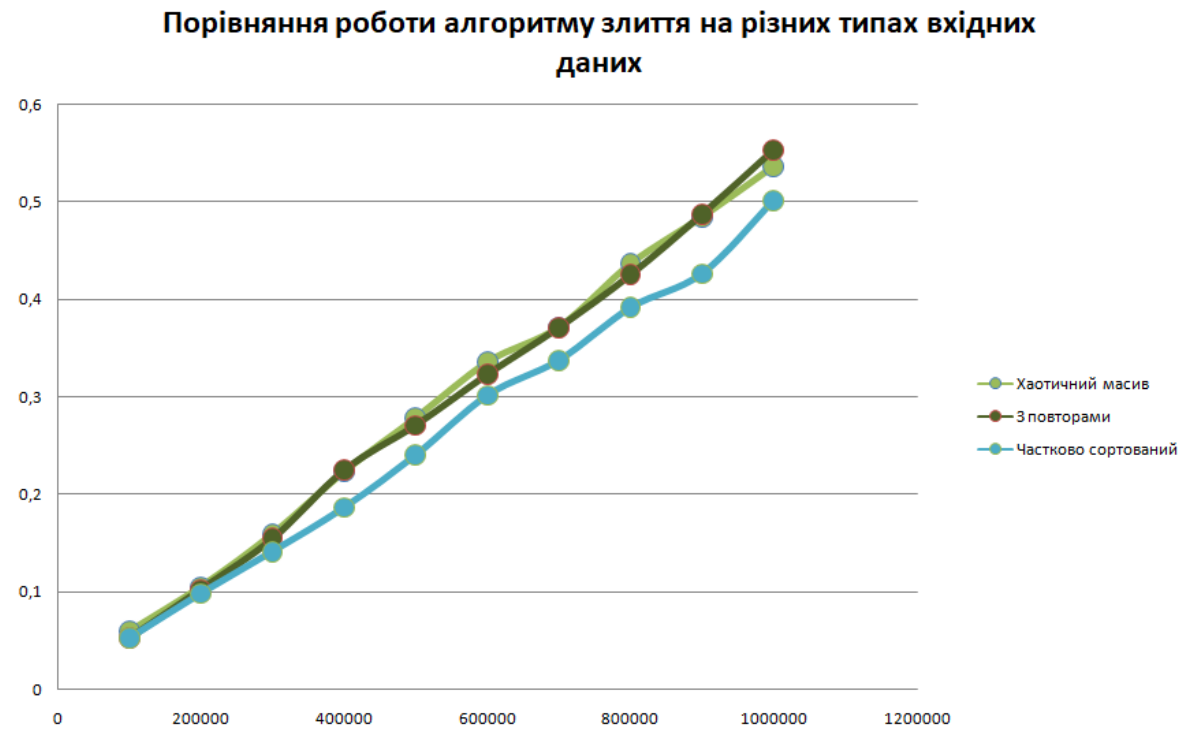
Перевірка обчислювальної складності програми:

Діагр. 1. Сортування хаотичного масиву алгоритмом злиття

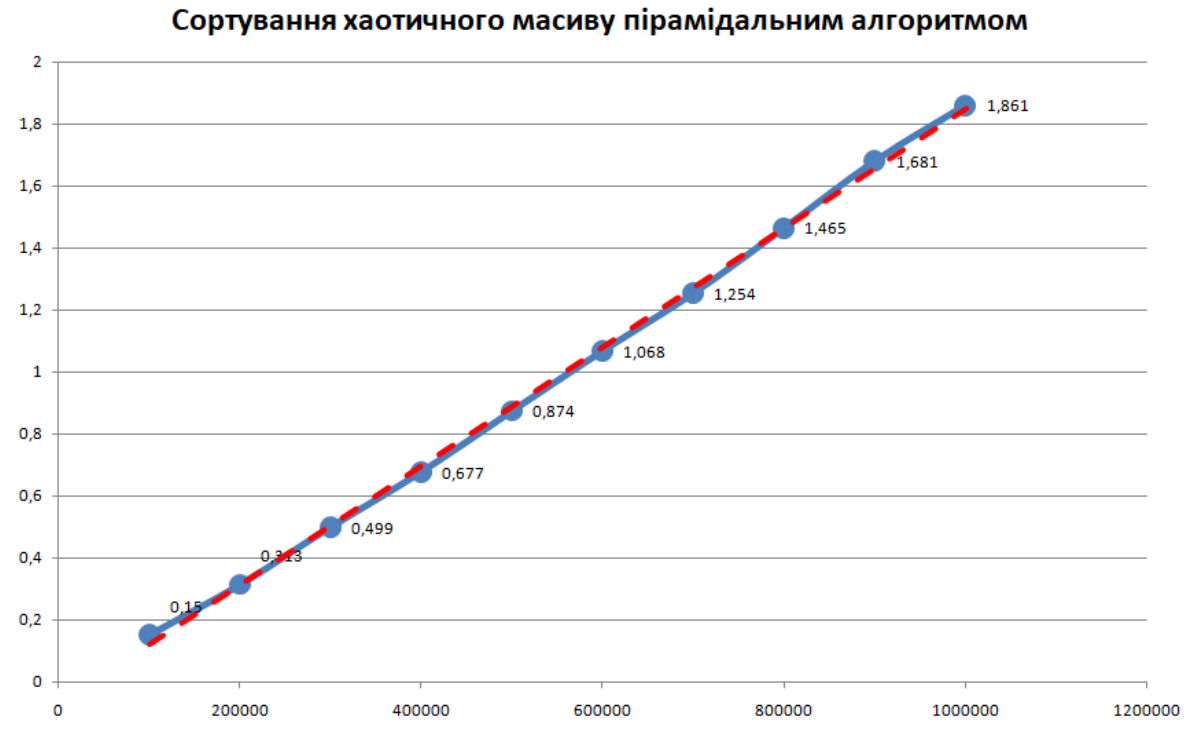


Діагр. 2. Сортування частково відсортованого масиву алгоритмом злиття

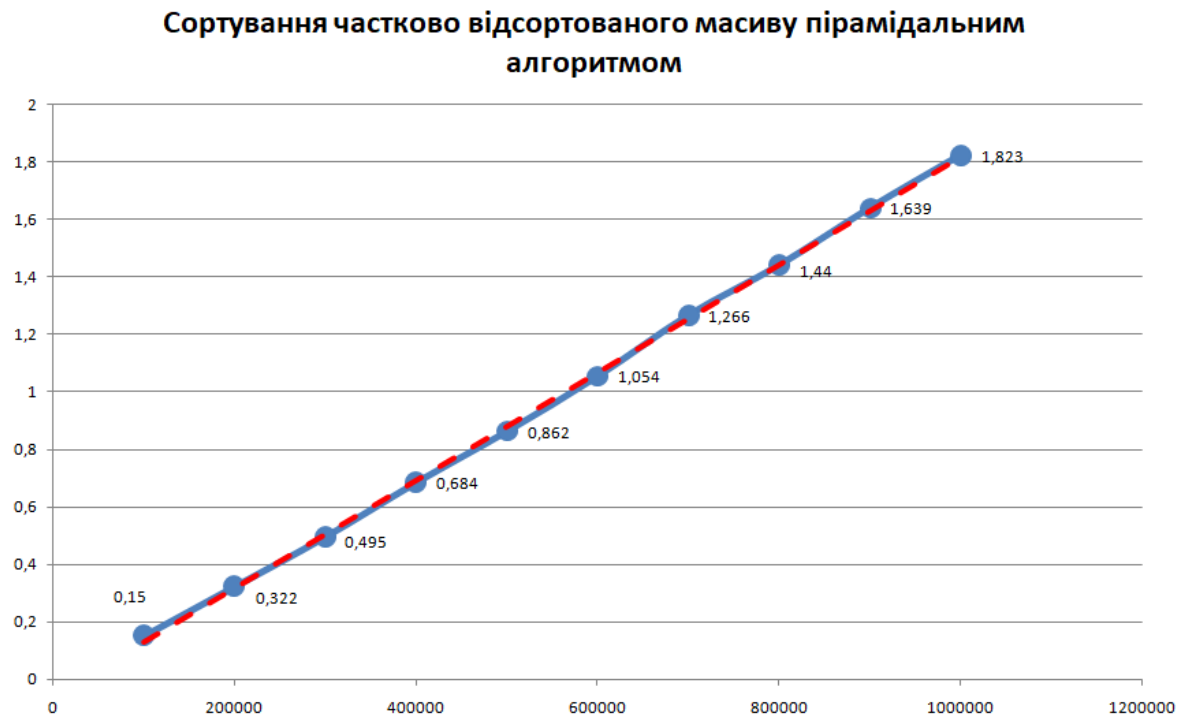
Діагр. 3. Сортування частково масиву з повторами алгоритмом злиття



Діагр. 4. Порівняльна діаграма роботи алгоритму злиття на різних типах вхідних даних



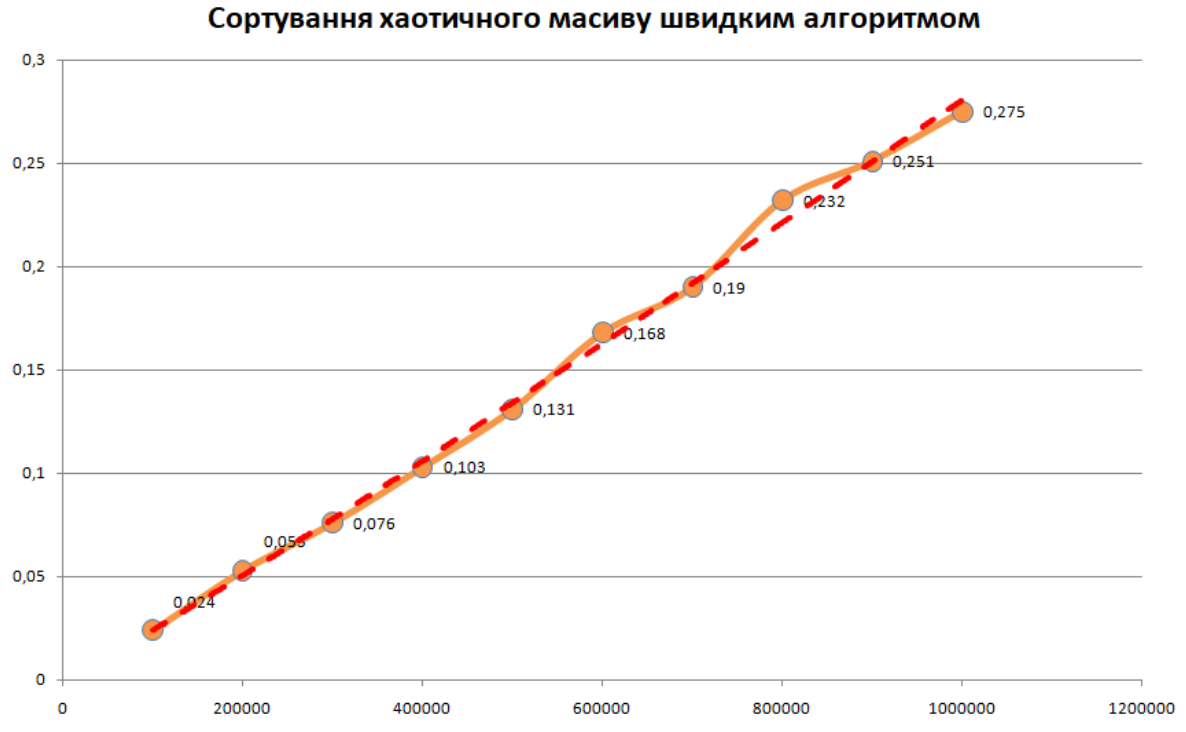
Діагр. 5. Сортування хаотичного масиву пірамідальним алгоритмом



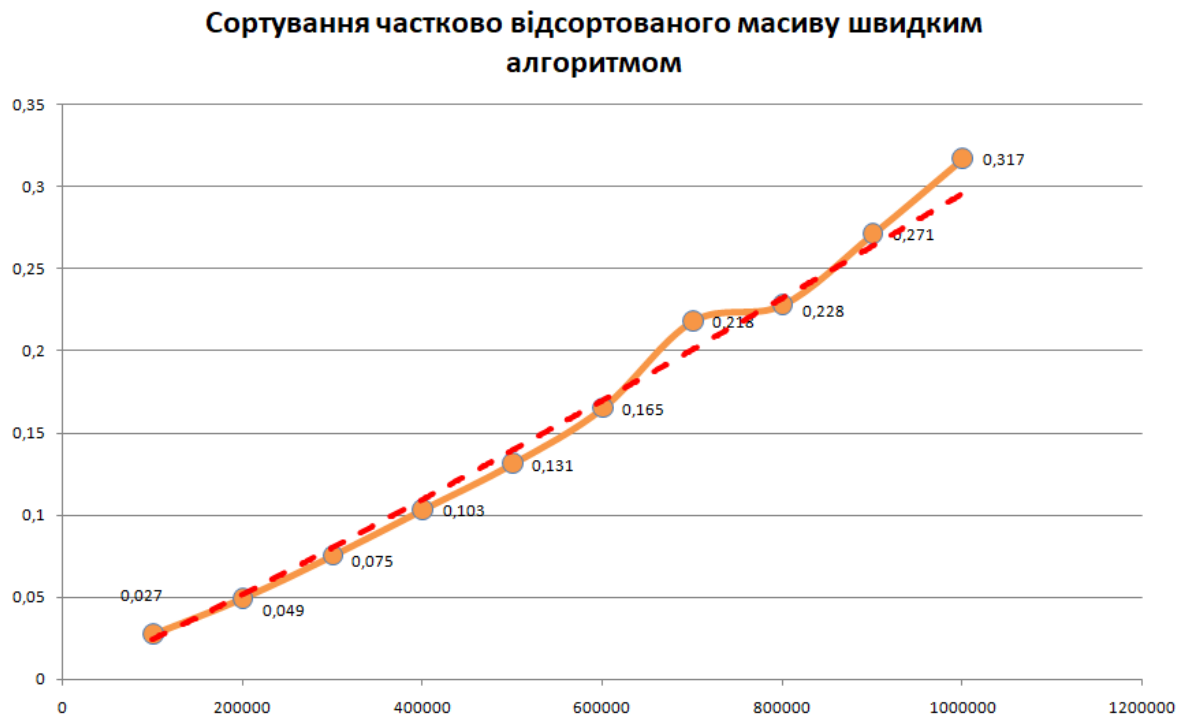
Діагр. 6. Сортування частково відсортованого масиву пірамідальним алгоритмом

Діагр. 7. Сортування масиву з повторами пірамідальним алгоритмом

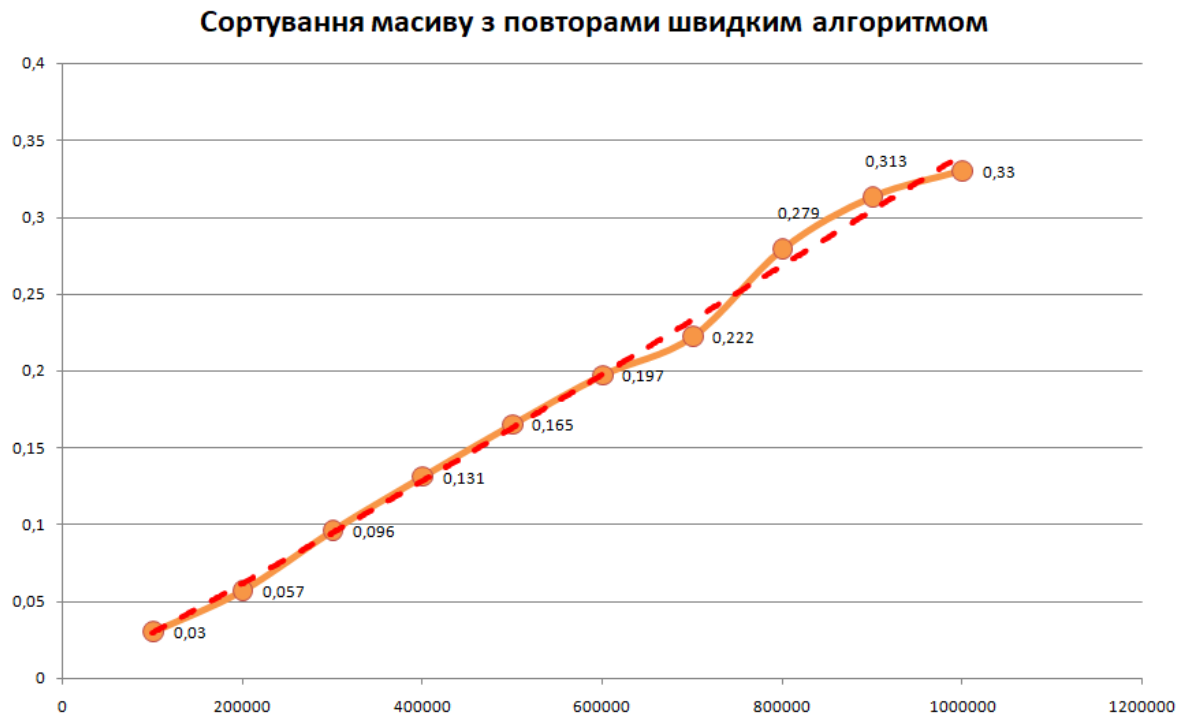
Діагр. 8. Порівняльна діаграма роботи пірамідального алгоритму на різних типах вхідних даних



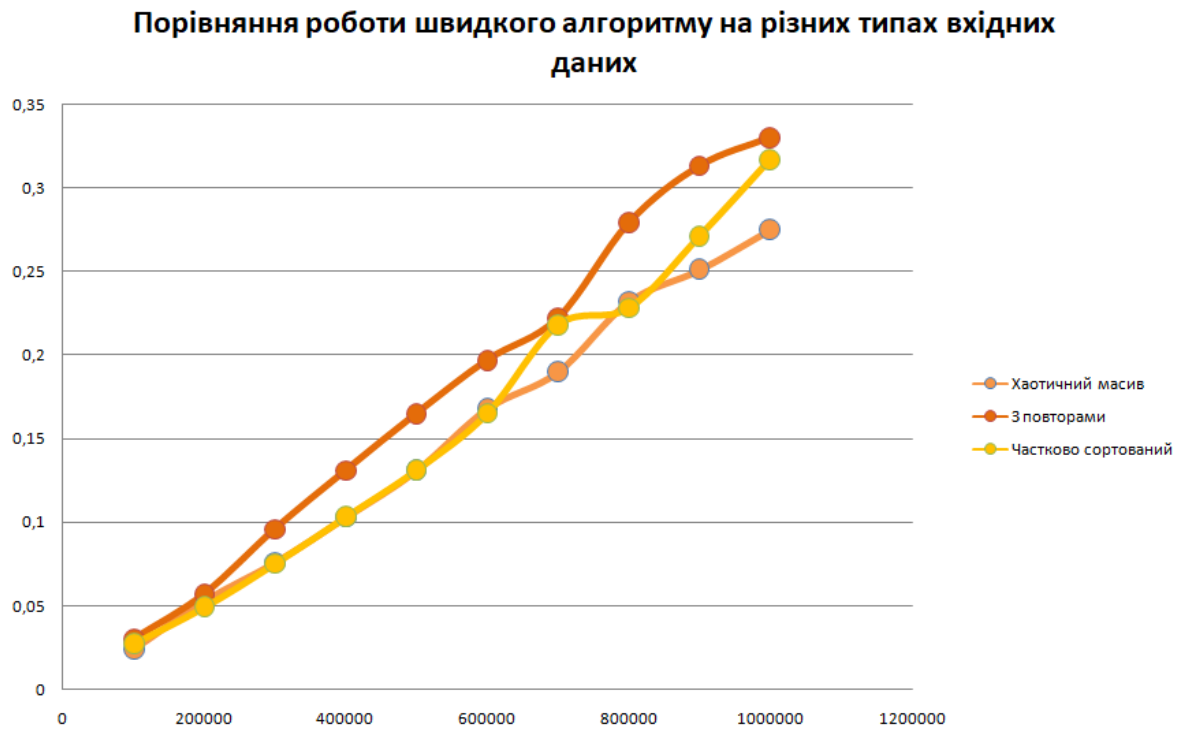
Діагр. 9. Сортування хаотичного масиву швидким алгоритмом



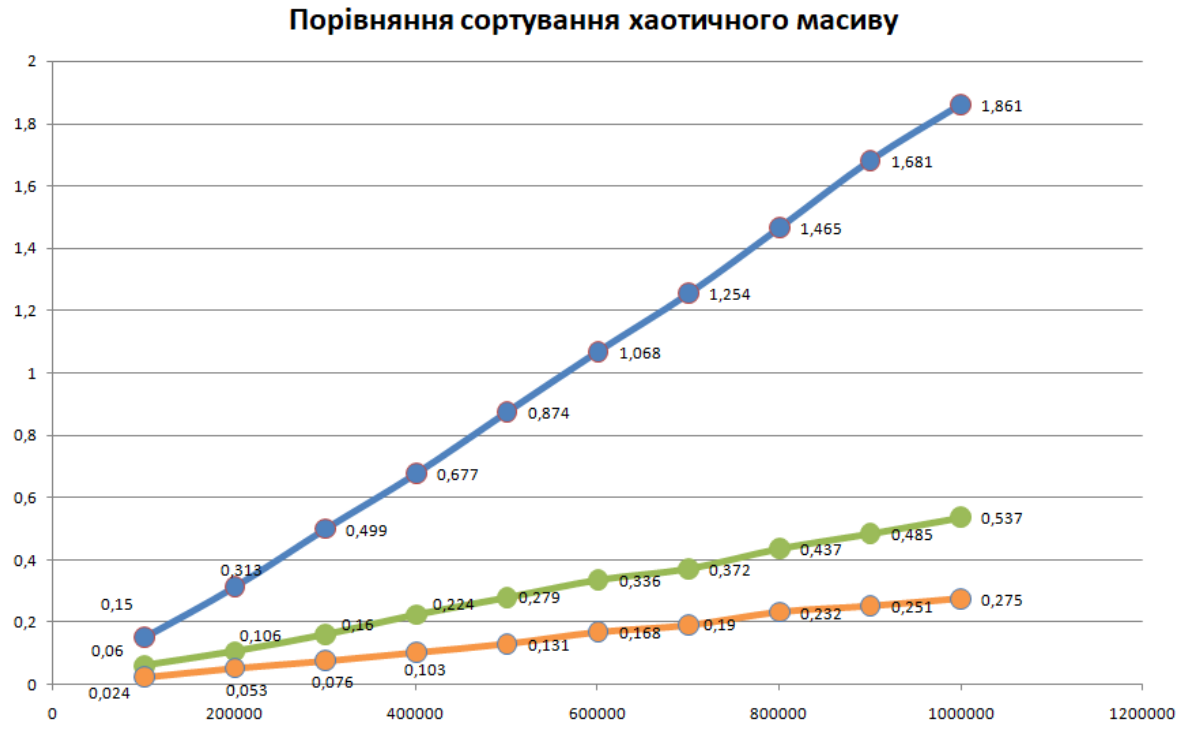
Діагр. 10. Сортування частково відсортованого масиву швидким алгоритмом



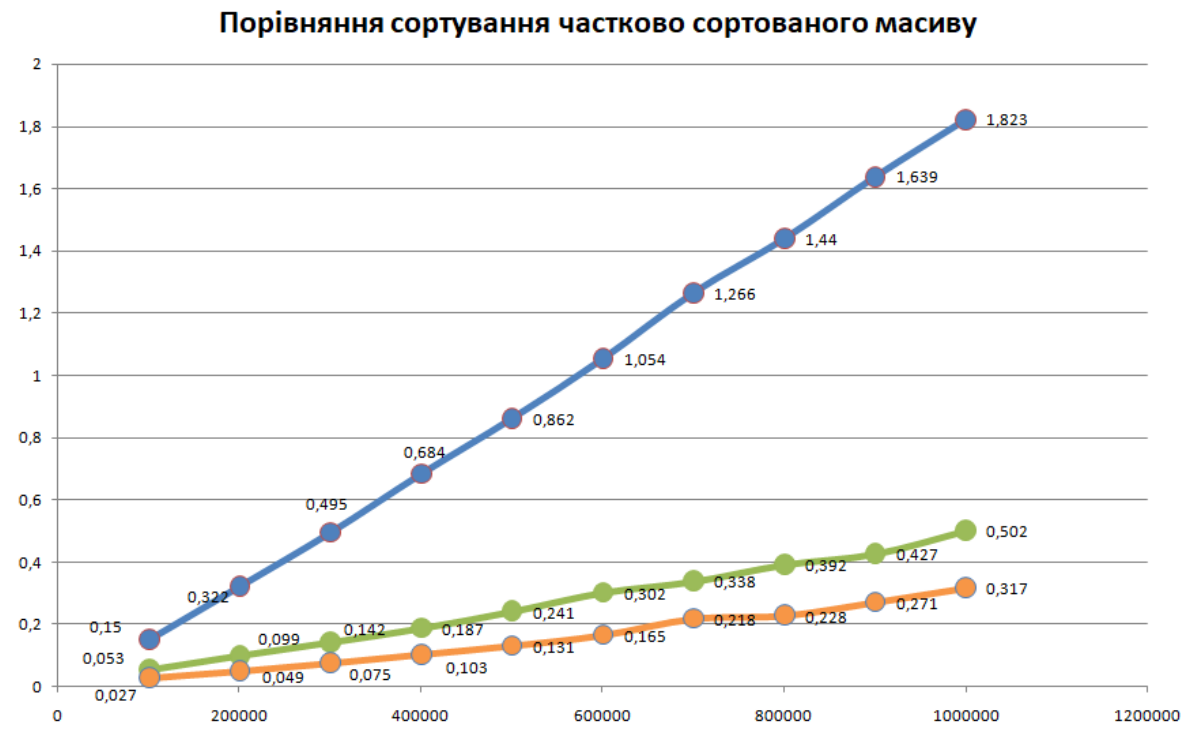
Діагр. 11. Сортування масиву з повторами швидким алгоритмом



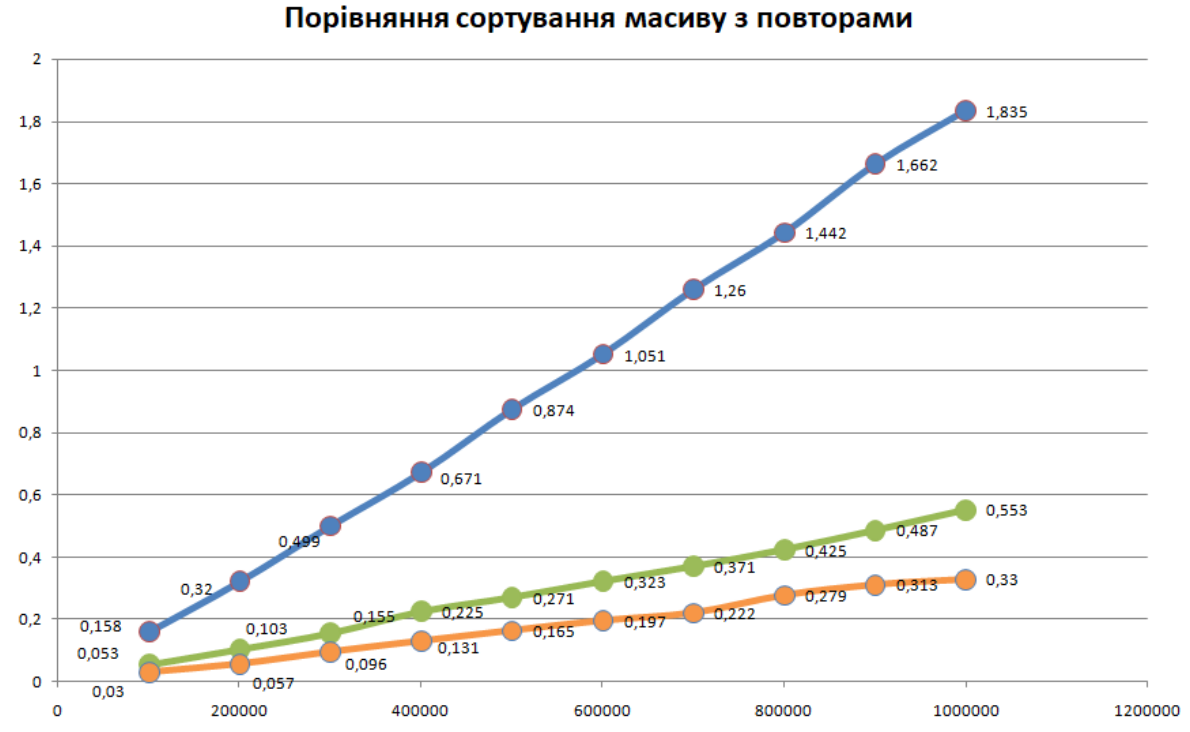
Діагр. 12. Порівняльна діаграма роботи швидкого алгоритму на різних типах вхідних даних



Діагр. 13. Порівняльна діаграма роботи різних алгоритмів на хаотичному масиві



Діагр. 14. Порівняльна діаграма роботи різних алгоритмів на частково сортованому масиві



Діагр. 15. Порівняльна діаграма роботи різних алгоритмів на масиві з повторами

Скріншоти роботи програми

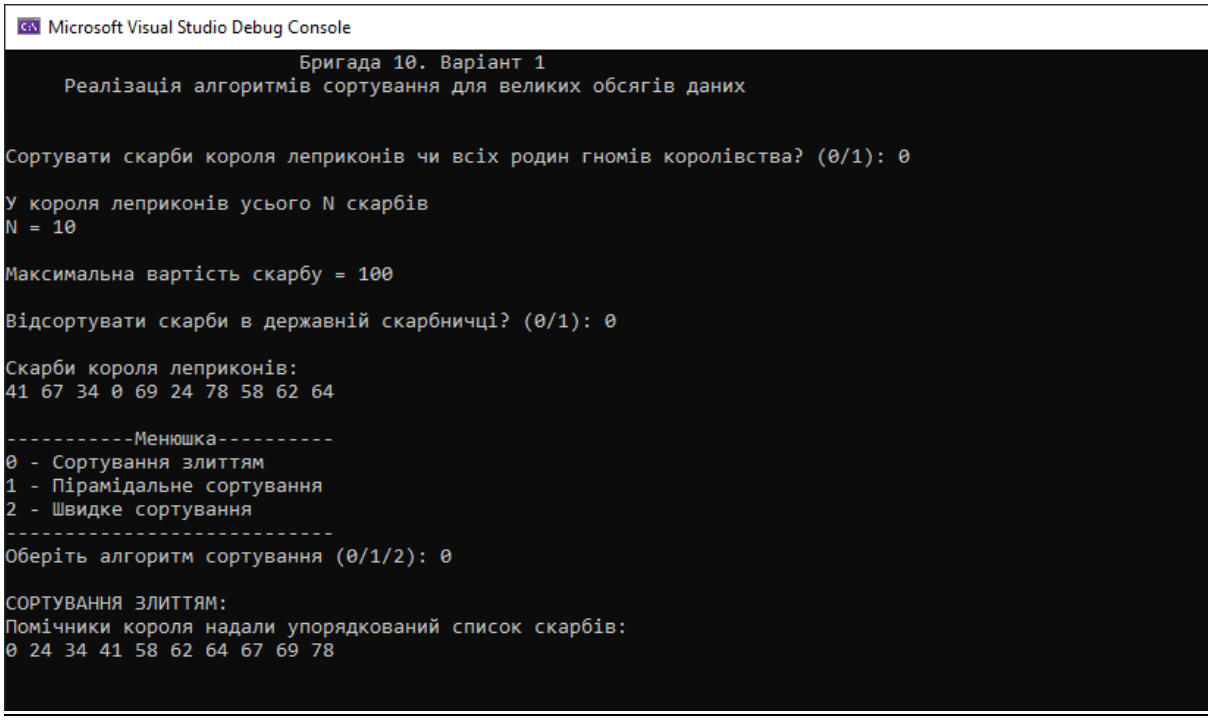


Рис. 4. Сортування злиттям з невідсортованим масивом

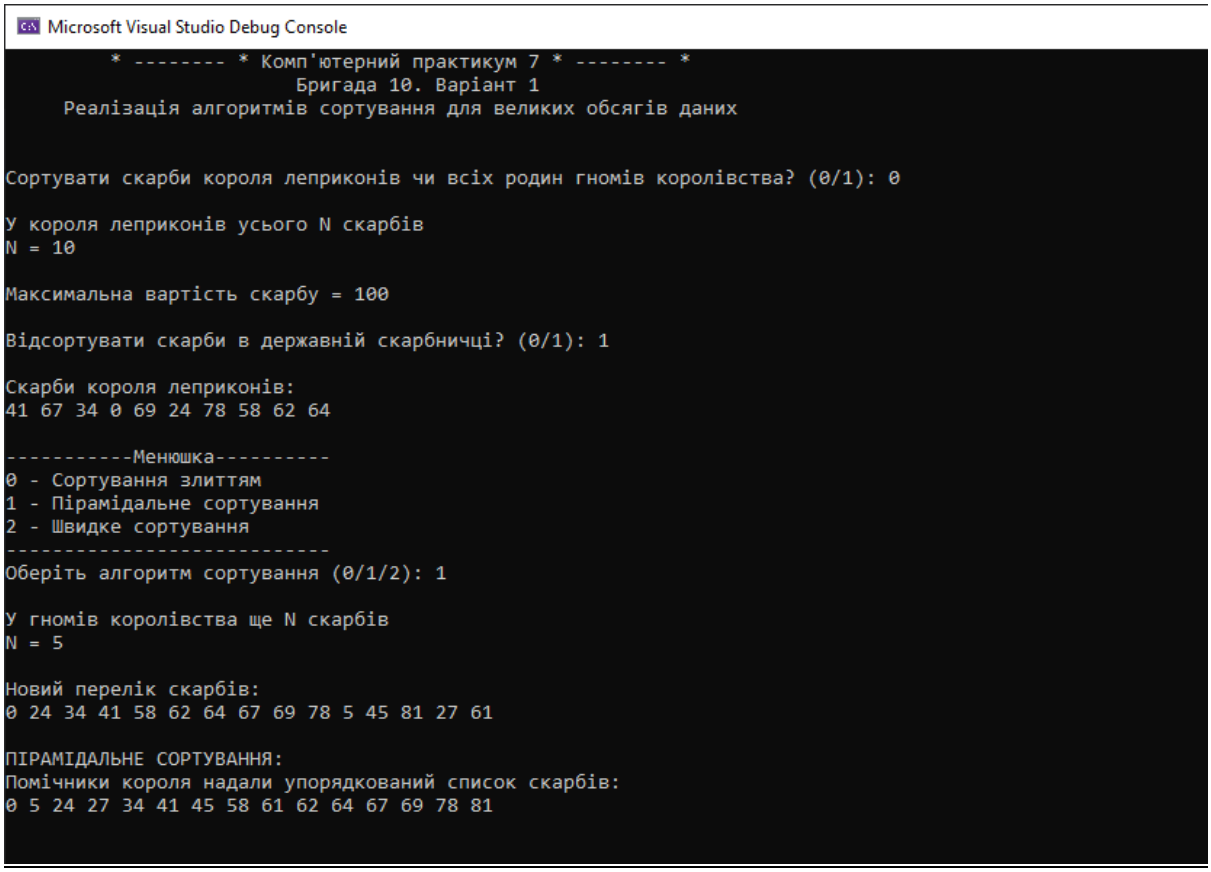


Рис. 5. Пірамідальне сортування з частково сортованним масивом

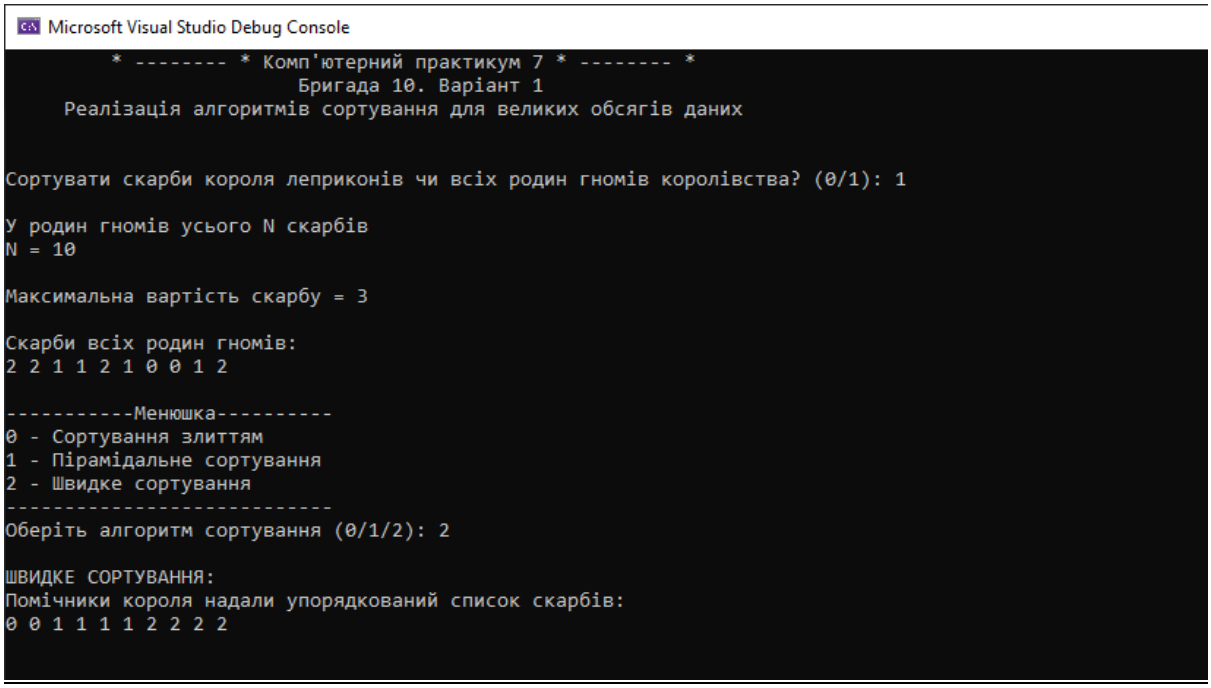


Рис. 6. Швидке сортування масиву з повторами

Висновок щодо доцільності використання алгоритму

Під час виконання лабораторної роботи, нами було зроблено припущення, щодо ефективності використання кожного з алгоритмів для сортування різного типу великих обсягів даних. Важливим питанням у нашому дослідженні була обчислювальна складність. У трьох випадках вона становить O(N*logN). Але розглядались і інші аспекти: складність реалізації, особливості поводження алгоритму на різних типах даних, його стабільність.

Фаворитом по швидкодії, беззаперечно, став алгоритм швидкого сортування, що гарно показав себе майже на кожному типу вхідних даних (з урахуванням просідання на масиві з повторами), що підтверджено діаграмами практичної складності.

Найгіршим випадком виявився алгоритм пірамідального сортування, що показав себе зі сторони складної реалізації і, у порівнянні з іншими, не вразив своєю швидкодією.

Перевірка правильності програми:

Вхідні дані length0, end0, addLength0	Результат	Призначення тесту
-1	Відомий	Перевірка, завершення некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
b	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
*	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
1.7	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
partlySorted0, var0 (value=asdf)	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
partlySorted0, var0 (value=*)	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)
partlySorted0, var0 (value=3)	Відомий	Перевірка реакції на некоректні вхідні дані (додавання повідомлень про помилки, завершення роботи програми)

Табл. 4. Таблиця тестування програми

Висновок

Під час виконання лабораторної роботи нами були досліджені різні алгоритми сортування великих обсягів даних. Зроблене нами припущення справдилось і метод швидкого сортування показав найефективніші результати роботи, а також є не найскладнішим у реалізації. Теоретична складність кожного з алгоритмів була підтверджена практично. Результатами роботи є дослідження та аналіз, щодо ефективності роботи, а також побудова наглядних порівняльних діаграм швидкодії роботи алгоритмів.

Відповіді на контрольні запитання

1. Перерахуйте та прокоментуйте переваги та недоліки швидкого сортування.

Що стосується складності швидкого сортування, то в найкращому випадку ми отримаємо складність Ω (n log n), а в найгіршому — O(n2). Окрім низької обчислювальної складності цьому алгоритму притаманні й інші переваги: на практиці це один з найбільш швидкодіючих алгоритмів внутрішнього сортування; алгоритм доволі простий як для розуміння, так і для реалізації; потребує лише O(n) пам’яті, для покращеної версії — O(1); дозволяє розпаралелювання для сортування підмасивів; працює на пов’язаних списках; є найефективнішим для сортування великої кількості даних.

Недоліками можна вважати:

- нестійкість;
- обчислювальна складність сильно деградує за умови невдалих вхідних даних.

2. Перерахуйте та прокоментуйте переваги та недоліки сортування злиттям. Які найпоширеніші модифікації цього методу сортування?

Алгоритм сортування злиття виконує точно так само і точно, незалежно від кількості елементів, що беруть участь у сортуванні. Це добре працює навіть з великим набором даних. Однак вона споживає більшу частину пам'яті. У сортуванні злиття масив повинен бути розділений на дві половини (тобто n / 2). На противагу, у швидкому роді, не існує примусу ділити список на рівні елементи. Найгірша складність швидкого сортування - O (n2), оскільки це потребує набагато більше порівнянь у найгіршому стані. На відміну від цього, сортування злиття має

однакові гірші випадки та середню складність випадку, тобто O (n log n). Сортування може добре працювати з будь-яким типом наборів даних, будь то великий чи малий.

3. Перерахуйте та прокоментуйте переваги та недоліки пірамідального сортування.

Переваги алгоритму:

- час роботи в найгіршому випадку — O(n\log n);
- вимагає O(1) додаткової пам'яті.

Недоліки алгоритму:

- нестійкий — для забезпечення стійкості потрібно розширювати ключ; на майже відсортованих даних працює так само довго, як і на хаотичних даних;
- складний в реалізації;
- на одному кроці вибірку доводиться робити хаотично по всій довжині масиву — тому алгоритм погано поєднується з кешуванням;
- методу потрібно «миттєвий» прямий доступ;
- не працює на зв'язаних списках та інших структурах пам'яті послідовного доступу.

4. Перерахуйте та прокоментуйте переваги та недоліки плавного сортування.

Як і пірамідальне сортування, в найгіршому випадку має швидкодію O(n log n). Перевагою плавного сортування є те, що його швидкодія наближається до O(n), якщо вхідні дані частково відсортовано, в той час як швидкодія пірамідального сортування є незмінною та не залежить від стану вхідних даних. Алгоритм плавного сортування вимагає пам'яті для

зберігання розмірів всіх куп в послідовності. Так як всі ці значення різні, як правило, для цієї мети застосовується бітова карта. Крім того, так як в послідовності не більш ніж O (log n) чисел, біти можуть бути закодовані O (1) машинними словами .

5. Зробіть порівняльну характеристику швидкого сортування та сортування злиттям.

	Швидке сортування	Сортування злиттям
Розбиття елементів у масиві	Розщеплення списку елементів не обов'язково ділиться навпіл	Масим завжди розділений навпіл (n/2)
Найгірший випадок складності	O (n^2)	O (N*logN)
Швидкість	Швидший за багатьох алгоритмів сортування	Постійна швидкість у всіх типах наборів даних
Спосіб сортування	Внутрішній	Зовнішній
Додаткова вимога місця для зберігання	Менше	Більше

6. Зробіть порівняльну характеристику пірамідального та плавного сортування.

Клас	Алгоритм сортування
Структура даних	масив
Найгірша швидкодія	$O(n \log n)$
Найкраща швидкодія	$\Omega(n), O(n \log n)$[1]
Середня швидкодія	$O(n \log n)$
Просторова складність у найгіршому випадку	$O(n)$ основний, $O(1)$ допоміжний
Оптимальний	Інколи
Стабільний	Нестійка

Клас	Алгоритм сортування
Структура даних	Масив
Найгірший випадок продуктивність	$O(n$ журнал $n)$
Кращий випадок продуктивність	$O(n)$
Середній продуктивність	$O(n$ журнал $n)$
Найгірший випадок космічна складність	$O(n)$ всього, $O(1)$ допоміжний