Звіт

Автор: Момот Р. КІТ-119а

Дата: 07.05.2020

ЛАБОРАТОРНА РОБОТА № 12. АЛГОРИТМИ СОРТУВАННЯ РОЗПОДІЛОМ ТА ЗЛИТТЯМ

Мета: закріпити теоретичні знання та набути практичний досвід впорядкування набору статичних та динамічних структур даних.

Індивідуальне завдання

Написати програму, що реалізує три алгоритми сортування набору даних.

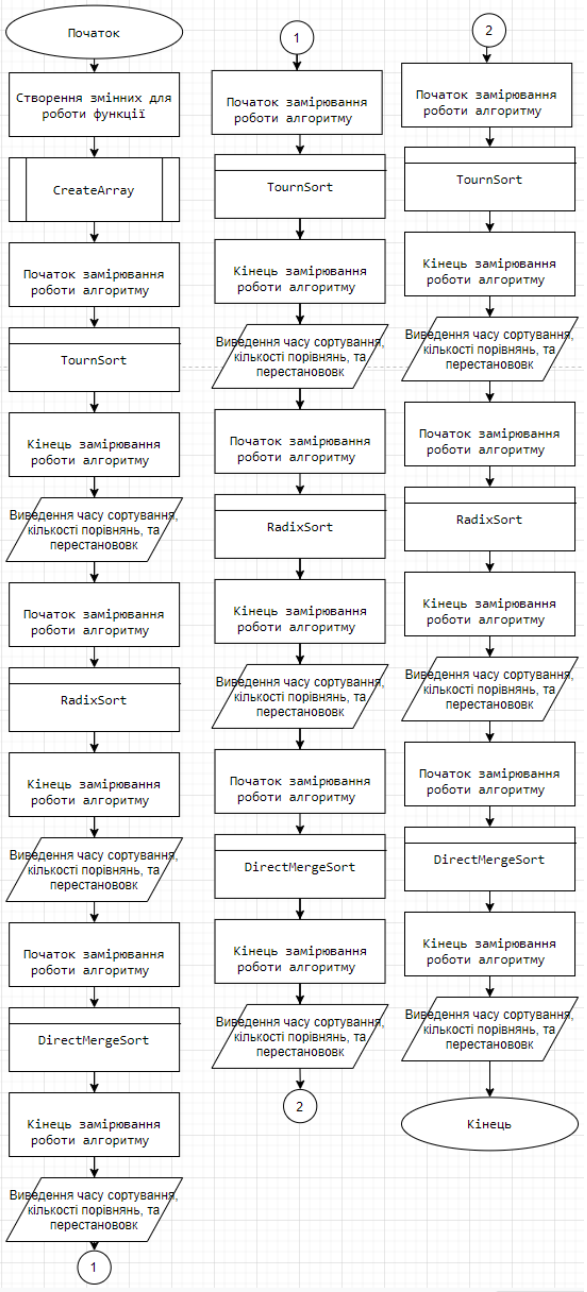
Визначити кількість порівнянь та обмінів для наборів даних, що містять різну кількість елементів (50, 1000, 5000, 10000, 50000).

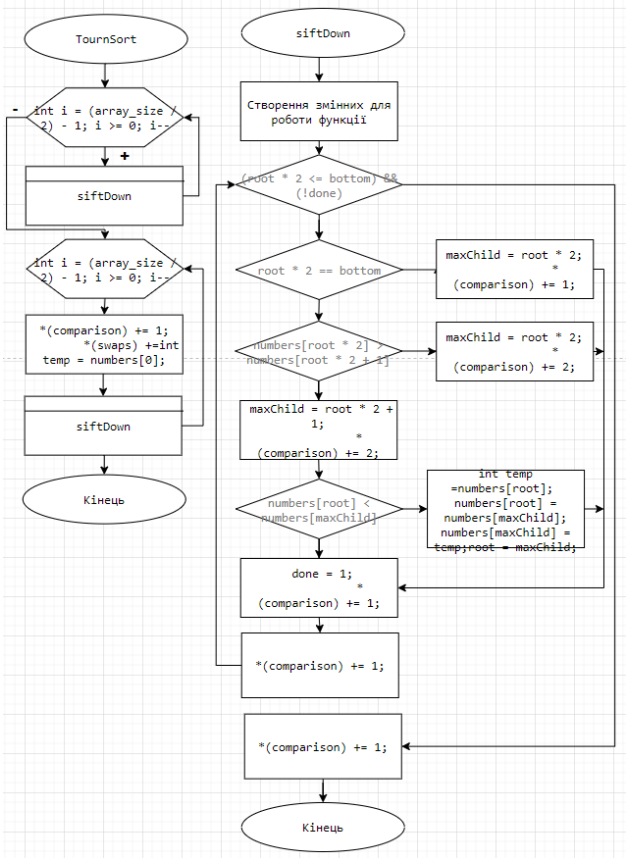
Оцінити час сортування. Дослідити вплив початкової впорядкованості набору даних (відсортований, відсортований у зворотньому порядку, випадковий).

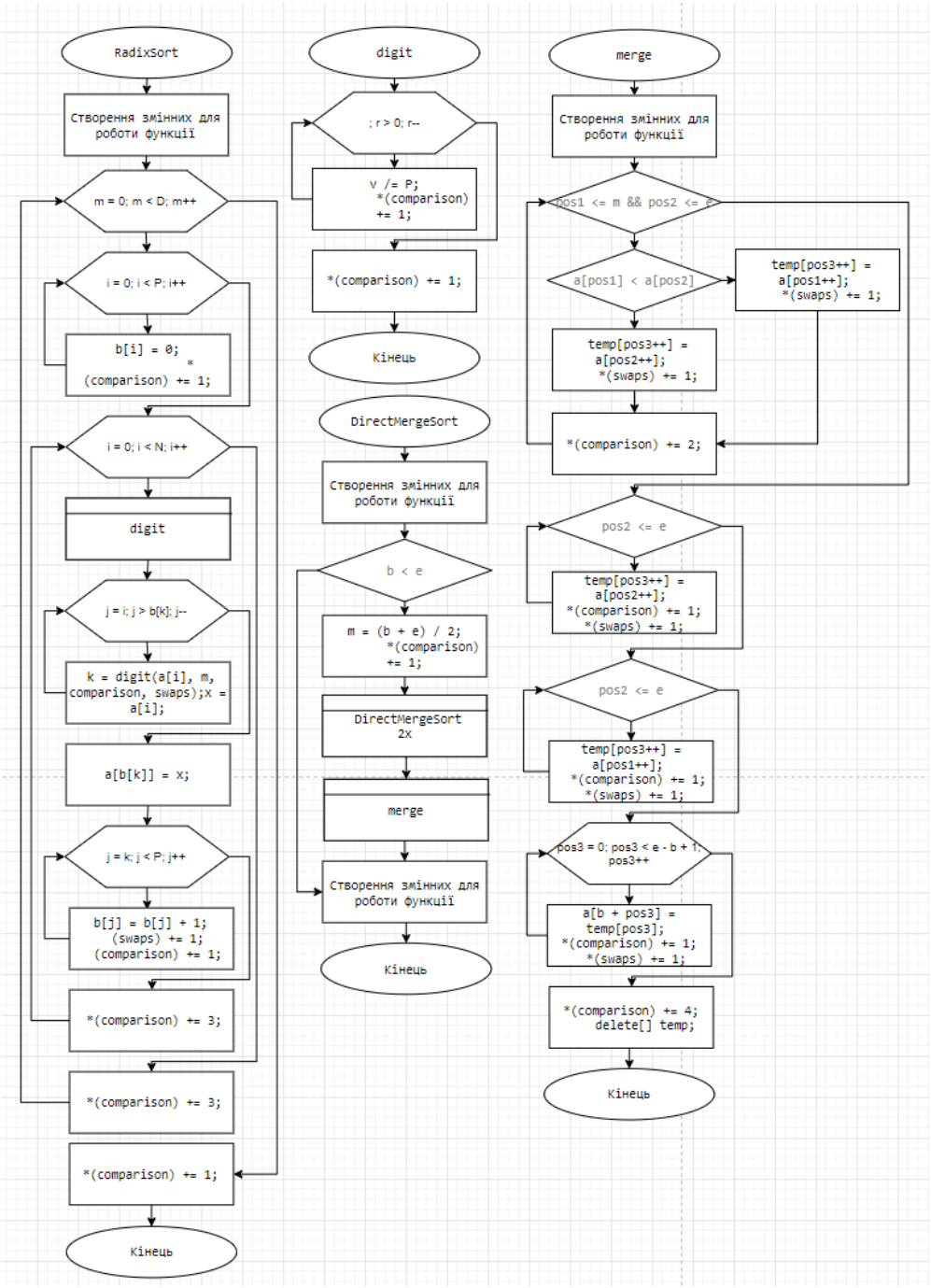
Алгоритми сортування:

- турнірне сортування;
- поразрядне цифрове сортування;
- сортування прямим злиттям.

Блок-схема алгоритму програми






RadixSort
Створення змінних для роботи функції
m = 0; m < D; m++
i = 0; i < P; i++
b[i] = 0;
(comparison) += 1;
i = 0; i < N; i++
digit
j = i; j > b[k]; j--
k = digit(a[i], m, comparison, swaps);x = a[i];
a[b[k]] = x;
j = k; j < P; j++
b[j] = b[j] + 1;
(swaps) += 1;
(comparison) += 1;
*(comparison) += 3;
*(comparison) += 3;
*(comparison) += 1;
Кінець
digit
; r > 0; r--
v /= P;
*(comparison) += 1;
*(comparison) += 1;
Кінець
DirectMergeSort
Створення змінних для роботи функції
b < e
m = (b + e) / 2;
*(comparison) += 1;
DirectMergeSort 2x
merge
Створення змінних для роботи функції
Кінець
merge
Створення змінних для роботи функції
pos1 <= m && pos2 <= e
a[pos1] < a[pos2]
temp[pos3++] = a[pos1++];
*(swaps) += 1;
temp[pos3++] = a[pos2++];
*(swaps) += 1;
*(comparison) += 2;
pos2 <= e
temp[pos3++] = a[pos2++];
*(comparison) += 1;
*(swaps) += 1;
pos2 <= e
temp[pos3++] = a[pos1++];
*(comparison) += 1;
*(swaps) += 1;
pos3 = 0; pos3 < e - b + 1; pos3++
a[b + pos3] = temp[pos3];
*(comparison) += 1;
*(swaps) += 1;
*(comparison) += 4;
delete[] temp;
Кінець

Текст програми

```
#define _CRTDBG_MAP_ALLOC
#include <crtdbg.h>
#define DEBUG_NEW new(_NORMAL_BLOCK, __FILE__, __LINE__)
#define new DEBUG_NEW

#include <iostream>
#include <locale>
#include <chrono>
using namespace std;

#define P 10
#define D 5             // Не более з-х разрядов в числах набора

void CreateArray(int*, int*, int*, int);
void OutputArray(int*, int);
void UsualSort(int*, int);

void TournSort(int*, int, int*, int*);
void ShiftDown(int*, int, int, int*, int*);

void RadixSort(int*, int, int*, int*);
int Digit(unsigned int, int, int*, int*);

void DirectMergeSort(int*, int, int, int*, int*);
void Merge(int*, int, int, int, int*, int*);

int main()
{
	setlocale(LC_ALL, "Rus");
	srand(time(0));
	const int SIZE = 20;		//размер массива
	int* array = new int[SIZE];//динамические массивы
	int* array2 = new int[SIZE];
	int* array3 = new int[SIZE];

	int comparisons = 0;		//количество сравнений
	int swaps = 0;				//количество перестановок
	int* pSwaps = &swaps;
	int* pComparison = &comparisons;

	CreateArray(array, array2, array3, SIZE);
	//OutputArray(array, SIZE);

	cout << "Количество элементов: " << SIZE << endl;
	cout << "=========================" << endl;
	cout << "Не отсортированные данные" << endl << endl;
	cout << "Турнирная сортировка" << endl;
	auto begin = chrono::steady_clock::now();
	TournSort(array, SIZE, pComparison, pSwaps);
	auto end = chrono::steady_clock::now();
	auto resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
	cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
	cout << "Количество сравнений: " << *pComparison << endl;
	cout << "Количество перестановок: " << *pSwaps << endl << endl;

	*pComparison = 0, * pSwaps = 0;

	cout << "Поразрядная цифровая сортировка" << endl;
	begin = chrono::steady_clock::now();
	RadixSort(array2, SIZE, pComparison, pSwaps);
	end = chrono::steady_clock::now();
	resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
```

```
cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
cout << "Количество сравнений: " << *pComparison << endl;
cout << "Количество перестановок: " << *pSwaps << endl << endl;

*pComparison = 0, * pSwaps = 0;

cout << "Поразрядная цифровая сортировка" << endl;
begin = chrono::steady_clock::now();
DirectMergeSort(array3, 0, SIZE - 1, pComparison, pSwaps);
end = chrono::steady_clock::now();
resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
cout << "Количество сравнений: " << *pComparison << endl;
cout << "Количество перестановок: " << *pSwaps << endl << endl;

*pComparison = 0, * pSwaps = 0;

cout << "==========================" << endl;
cout << "Отсортированные данные" << endl << endl;

cout << "Турнирная сортировка" << endl;
begin = chrono::steady_clock::now();
TournSort(array, SIZE, pComparison, pSwaps);
end = chrono::steady_clock::now();
resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
cout << "Количество сравнений: " << *pComparison << endl;
cout << "Количество перестановок: " << *pSwaps << endl << endl;

*pComparison = 0, * pSwaps = 0;

cout << "Поразрядная цифровая сортировка" << endl;
begin = chrono::steady_clock::now();
RadixSort(array2, SIZE, pComparison, pSwaps);
end = chrono::steady_clock::now();
resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
cout << "Количество сравнений: " << *pComparison << endl;
cout << "Количество перестановок: " << *pSwaps << endl << endl;

*pComparison = 0, * pSwaps = 0;

cout << "Поразрядная цифровая сортировка" << endl;
begin = chrono::steady_clock::now();
DirectMergeSort(array3, 0, SIZE - 1, pComparison, pSwaps);
end = chrono::steady_clock::now();
resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
cout << "Количество сравнений: " << *pComparison << endl;
cout << "Количество перестановок: " << *pSwaps << endl << endl;

UsualSort(array, SIZE);
UsualSort(array2, SIZE);
UsualSort(array3, SIZE);
*pComparison = 0, * pSwaps = 0;

cout << "==========================" << endl;
cout << "Отсортированные данные в обратном порядке" << endl << endl;

cout << "Турнирная сортировка" << endl;
begin = chrono::steady_clock::now();
TournSort(array, SIZE, pComparison, pSwaps);
end = chrono::steady_clock::now();
resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
cout << "Количество сравнений: " << *pComparison << endl;
```

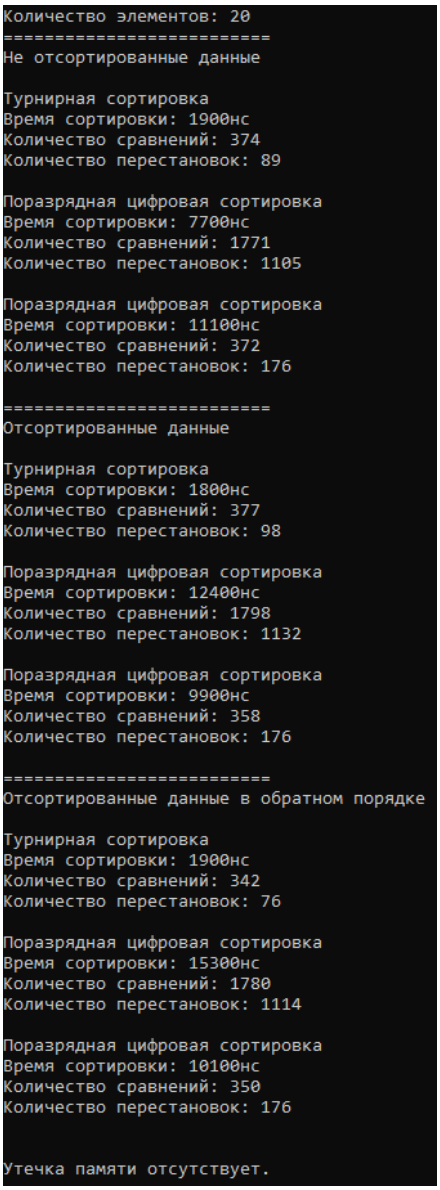
```
        cout << "Количество перестановок: " << *pSwaps << endl << endl;

        *pComparison = 0, * pSwaps = 0;

         cout << "Поразрядная цифровая сортировка" << endl;
        begin = chrono::steady_clock::now();
        RadixSort(array2, SIZE, pComparison, pSwaps);
        end = chrono::steady_clock::now();
        resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
        cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
        cout << "Количество сравнений: " << *pComparison << endl;
        cout << "Количество перестановок: " << *pSwaps << endl << endl;

        *pComparison = 0, * pSwaps = 0;

        cout << "Поразрядная цифровая сортировка" << endl;
        begin = chrono::steady_clock::now();
        DirectMergeSort(array3, 0, SIZE - 1, pComparison, pSwaps);
        end = chrono::steady_clock::now();
        resultClock = chrono::duration_cast<chrono::nanoseconds>(end - begin);
        cout << "Время сортировки: " << resultClock.count() << "нс" << endl;
        cout << "Количество сравнений: " << *pComparison << endl;
        cout << "Количество перестановок: " << *pSwaps << endl << endl;

        delete[] array;
        delete[] array2;
        delete[] array3;

        if (_CrtDumpMemoryLeaks())
                cout << "\nУтечка памяти обнаружена." << endl;
        else
                cout << "\nУтечка памяти отсутствует." << endl;
}

void CreateArray(int* array,int* array2, int* array3, int size)
{
    int value = 0;
    for (size_t i = 0; i < size; i++)
    {
        value = rand();
        *(array + i) = value;
        *(array2 + i) = value;
        *(array3 + i) = value;
    }

}
void OutputArray(int* array, int size)
{
        cout << endl;
        for (size_t i = 0; i < size; i++)
                cout << *(array + i) << "  ";
        cout << endl;
}
void UsualSort(int* array, int size)
{
    int temp;
    bool pr = 0;
    do
    {
        pr = 0;
        for (size_t i = 0; i < size; i++)
            if (array[i] < array[i + 1])
            {
                temp = array[i];
                array[i] = array[i + 1];
                array[i + 1] = temp;
```

```
                pr = 1;
            }
    } while (pr);
}

void TournSort(int* numbers, int array_size, int* comparison, int* swaps)
{
    for (int i = (array_size / 2) - 1; i >= 0; i--)     // Формируем нижний ряд пирамиды
    {
        *(comparison) += 1;
        ShiftDown(numbers, i, array_size - 1, comparison, swaps);
    }
    for (int i = array_size - 1; i >= 1; i--)   // Просеиваем через пирамиду остальные элементы
    {
        *(comparison) += 1;
        *(swaps) += 1;
        int temp = numbers[0];
        numbers[0] = numbers[i];
        numbers[i] = temp;
        ShiftDown(numbers, 0, i - 1, comparison, swaps);
    }
    *(comparison) += 2;
}
void ShiftDown(int* numbers, int root, int bottom, int* comparison, int* swaps)
{
    int maxChild;   // индекс максимального потомка
    int done = 0;   // флаг того, что куча сформирована
    while ((root * 2 <= bottom) && (!done))     // Пока не дошли до последнего ряда
    {
        if (root * 2 == bottom)           // если мы в последнем ряду, запоминаем левый потомок
        {
            maxChild = root * 2;
            *(comparison) += 1;
        }
        else if (numbers[root * 2] > numbers[root * 2 + 1]) // иначе запоминаем больший потомок
из двух
        {
            maxChild = root * 2;
            *(comparison) += 2;
        }
        else                              // если элемент вершины меньше максимального потомка
        {
            maxChild = root * 2 + 1;
            *(comparison) += 2;
        }
        if (numbers[root] < numbers[maxChild])
        {
            int temp = numbers[root];           // меняем их местами
            numbers[root] = numbers[maxChild];
            numbers[maxChild] = temp;
            root = maxChild;
            *(comparison) += 1;
            *(swaps) += 1;
        }
        else // иначе пирамида сформирована
        {
            done = 1;
            *(comparison) += 1;
        }
        *(comparison) += 1;
    }
    *(comparison) += 1;
}

void RadixSort(int* a, int N, int* comparison, int* swaps)  // Поразрядная сортировка
{
```

```
    int b[P];                                   // Индекс элемента, расположенного за последним
в i-й группе
    int i, j, k, m, x;
    for (m = 0; m < D; m++)                    // Перебираем все числа, начиная с младшего
разряда
    {
        for (i = 0; i < P; i++)               // Обнуляем индексы
        {
            b[i] = 0;
            *(comparison) += 1;
        }
        for (i = 0; i < N; i++)                // Проходим массив
        {
            k = Digit(a[i], m, comparison, swaps);                // Определяем m-ю цифру
            x = a[i];                                    // Сохраняем элемент
            for (j = i; j > b[k]; j--)       // И затираем его сдвигая массив вправо
            {
                a[j] = a[j - 1];
                *(swaps) += 1;
                *(comparison) += 1;
            }
            a[b[k]] = x;                          // Записываем его в конец k-й группы
            for (j = k; j < P; j++)                // Модифицируем все индексы не меньшие k
            {
                b[j] = b[j] + 1;
                *(swaps) += 1;
                *(comparison) += 1;
            }
            *(comparison) += 3;
        }
        *(comparison) += 3;
    }
    *(comparison) += 1;
}
int Digit(unsigned int v, int r, int* comparison, int* swaps)   // Возвращает n-ю цифру считая
с нуля в числе v
{
    for (; r > 0; r--)
    {
        v /= P;
        *(comparison) += 1;
    }
    *(comparison) += 1;
    return v % P;
}

void DirectMergeSort(int* a, int b, int e, int* comparison, int* swaps)  // Сама функция
сортировки слиянием
{
    long m;                                          // индекс, по которому делим массив
    if (b < e)                                          // если есть более 1 элемента
    {
        m = (b + e) / 2;
        *(comparison) += 1;
        DirectMergeSort(a, b, m, comparison, swaps);          // сортировать левую половину
        DirectMergeSort(a, m + 1, e, comparison, swaps);         // сортировать правую половину
        Merge(a, b, m, e, comparison, swaps);                  // слить результаты в общий
массив
    }
    *(comparison) += 1;
}
void Merge(int* a, int b, int m, int e, int* comparison, int* swaps)
{
    int pos1 = b;                    // текущая позиция чтения из первой последовательности
a[b]...a[m]
```

```
    int pos2 = m + 1;               // текущая позиция чтения из второй последовательности
a[m+1]...a[e]
    int pos3 = 0;                   // текущая позиция записи в temp
    int* temp = new int[e - b + 1];
    while (pos1 <= m && pos2 <= e)  // слияние, пока есть хотя бы один элемент в каждой
последовательности
    {
        if (a[pos1] < a[pos2])
        {
            temp[pos3++] = a[pos1++];
            *(swaps) += 1;
        }
        else
        {
            temp[pos3++] = a[pos2++];
            *(swaps) += 1;
        }
        *(comparison) += 2;
    }
    // одна последовательность закончилась - копировать остаток другой в конец буфера
    while (pos2 <= e)                        // пока вторая последовательность не пуста
    {
        temp[pos3++] = a[pos2++];
        *(comparison) += 1;
        *(swaps) += 1;
    }
    while (pos1 <= m)                          // пока первая последовательность не пуста
    {
        temp[pos3++] = a[pos1++];
        *(comparison) += 1;
        *(swaps) += 1;
    }
    for (pos3 = 0; pos3 < e - b + 1; pos3++) // скопировать буфер temp в a[b]...a[e]
    {
        a[b + pos3] = temp[pos3];
        *(comparison) += 1;
        *(swaps) += 1;
    }
    *(comparison) += 4;
    delete[] temp;
}
```

Результат роботи програми



Результати тестування алгоритма сортування даних

Алгоритм турнірного сортування					
Відсортований набір даних	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	377	42295	259515	559027	3262482
Кількість пересилань	98	10710	65958	141421	825541
Час сортування	1800	108800	596500	1753300	7328100
Відсортований в зворотньому порядку	**20**	**1000**	**5000**	**10000**	**50000**

Кількість порівнянь	342	38934	241535	523442	3078552
Кількість пересилань	76	9329	58342	126849	749364
Час сортування	1900	101200	597200	1327600	7386200
Випадковий набір даних	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	374	40544	250567	540698	3170346
Кількість пересилань	89	10049	62177	134161	787737
Час сортування	1900	125800	716400	1473000	9123000
Алгоритм порозрядного цифрового сортування					
Відсортований набір даних	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	1798	1144904	27813065	111270585	2767722599
Кількість пересилань	1132	1114838	27662999	110970519	2766222533
Час сортування	12400	2801300	76436600	268430100	6882409400
Відсортований в зворотньому порядку	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	1780	1166536	27734523	111276859	2769670088
Кількість пересилань	1114	1136470	27584457	110976793	2768170022
Час сортування	15300	2656300	76034000	251290500	6855075700
Випадковий набір даних	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	1771	1144833	27788759	111043490	2766138501
Кількість пересилань	1105	1114767	27638693	110743424	2764638435
Час сортування	7700	2695800	63891100	277814800	7183770500
Алгоритм сортування прямим злиттям					
Відсортований набір даних	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	358	31994	190755	406809	2329774
Кількість пересилань	176	19952	123616	267232	1568928
Час сортування	9900	350900	1825200	4455200	29542000
Відсортований в зворотньому порядку	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	350	31878	188414	401834	2301434

Кількість пересилань	176	19952	123616	267232	1568928
Час сортування	10100	355100	3010400	7184000	18086800
Випадковий набір даних	**20**	**1000**	**5000**	**10000**	**50000**
Кількість порівнянь	372	35660	213812	457644	2636989
Кількість пересилань	176	19952	123616	267232	1568928
Час сортування	11100	597600	1942600	7245400	21145600

*Час пошуку вказан у мілісекундах.

Висновок

У результаті лабораторної роботи програми було розроблено програму, яка базується на статичному масиві та включає в себе три алгоритми сортування, а саме: турнірне сортування, порозрядне цифрове сортування та сортування прямим злиттям. Як можна бачити із таблиці вище на випадковому наборі даних найшвидшим виявилося турнірне сортування.

Відповіді на питання

1. Поясніть, чому алгоритми сортувань на деревах характеризуються порядком log2(N)?

Алгоритми сортувань на деревах дуже ефективні через особливості їх будови, тому їх порядок log2(N).

2. В чому основна сутність алгоритму сортування Хоара?

Спочатку знаходиться «опорний елемент», далі кожен елемент порівнюється з опорним і масив сортується на 3 масива – менше опорного елемента, равні опорному, більше опорного. Далі більші та менші відрізки рекурсивно сортуються за тим же алгоритмом.

3. Які алгоритми сортувань вимагають додаткову область пам’яті і як вона використовується?

Турнірне сортування – елементи потрібно дублювати на разних рівнях масиву.

Порозрядне цифрове сортування – декілька елементів можуть попасти у одну групу.

Швидке сортування Хоара – додаткова пам’ять витрачається на стек, у який заноситься інформація про розмір інтервалу.

Сортування простими вставками – дані із одного масиву переносяться в інший масив.

Карманне сортування – покрокове переміщення елементів з одного масиву в інший.

4. Назвіть алгоритми сортувань, в яких немає порівнянь елементів набору даних, що впорядковується?

Порозрядне цифрове сортування, “карманне” сортування, сортування виродженим розподілом, блочне сортування, сортування підрахунком.

5. Які фактори впливають на вибір алгоритма сортування?

Порядок алгоритму, необхідний ресурс пам’яті, початкова впорядкованість вхідної множини,часові характеристики операцій, складність алгоритму, розмір даних, типи сортуємих даних

.

6. Напишіть програмний код «кишенькового» сортування, з однією областю пам’яті.

```
void bucketSort(float arr[], int n)
{
	vector<float> b[n];
	for (int i = 0; i < n; i++)
	{
		int bi = n * arr[i]; // индекс в ковши
		b[bi].push_back(arr[i]);
	}
	for (int i = 0; i < n; i++)
		sort(b.begin(), b.end());
	int index = 0;
	for (int i = 0; i < n; i++)
		for (int j = 0; j < b[i].size(); j++)
			arr[index++] = b[i][j];
}
```

7. За скільки кроків буде впорядковано за зростанням наступний набір чисел: 231, 24, 3, 35, 15, 932, 176, 83, 8, 54, 112?

Даний набір чисел буде впорядкован за 26 кроків.

8. Яке призначення має додаткова пам’ять, що необхідна алгоритму сортування Хоара?

У процесі роботи алгоритму розподіляються записи підмножин по іншим підмножинам, поки отримана підмножина не буде мати в собі лише один елемент. На кожному проході підмножин одночасно можна обробляти лише одну, а дані про другу підмножину повинні записуватися у стек.

9. Яке призначення додаткової пам’яті, що необхідна в алгоритмах пошука підрядка в рядку. Назвіть ці алгоритми?

У алгоритмах пошука підрядка в рядку додаткову пам’ять потребує дескриптор(масив), в який записуються значення зміщень для кожного символа зразка, якщо не порівняння буде на цьому символі.

10. Дайте опис алгоритма турнірного сортування.

У кожній ітерації циклу вибирається значення з вершини дерева - це найбільша з наявних значень ключа. Вузол-вершина при цьому звільняється, звільняються також і всі вузли, займані вибраним значенням на більш низьких рівнях турнірного дерева. За звільнилися вузли влаштовується (від низу до верху) змагання між їхніми нащадками.

11. Дайте опис алгоритма сортування частково впорядкованим деревом.

Для сортування цим методом повинні бути визначені дві операції:

- вставка в дерево нового елемента
- вибірка з дерева мінімального елемента.

Алгоритм вставки полягає в наступному. Новий елемент вставляється на перше вільне місце за кінцем дерева. Якщо ключ вставленого елемента менше, ніж ключ його предка, то предок і вставлений елемент міняються місцями. Ключ вставленого елемента тепер порівнюється з ключем його предка на новому місці і т.д. Порівняння закінчуються, коли ключ нового елемента виявиться більше ключа предка або коли новий елемент «випливе» в вершину піраміди.

Алгоритм вибірки. Мінімальний елемент знаходиться в вершині. Після вибірки за місце, що звільнилося влаштовується змагання між нащадками, і в вершину переміщається нащадок з найменшим значенням ключа. За час, що звільнився місце переміщених нащадка змагаються його нащадки і т.д., поки вільне місце не опуститься до листа дерева. Так відновлюється впорядкованість дерева, але може бути порушено умова його збалансованості, так як вільне місце знаходиться не в кінці дерева. Для відновлення збалансованості останній елемент дерева переноситься на місце, що звільнилося, а потім «спливає» за тим же алгоритмом, який застосовувався при вставці.

Листки тривіально впорядковані, тому можна почати з мінімальних піддерев,

що містять кілька вершин, і укрупнювати їх, кожен раз повністю, застосовуючи алгоритм спливання до тих пір, поки таким чином не буде досягнутий корінь дерева.

Саме так здійснюється формування вихідного майже упорядкованого дерева.

Після того як дерево впорядковано, найбільший (найменший) елемент виявляється в його корені. За алгоритмом знайдений елемент міняють місцями з найостаннішими листом в дереві (останній елемент розглянутого масиву), дерево зменшується на одну вершину і все готово для визначення нового найбільшого (найменшого) елемента множини.

12.Дайте опис алгоритма сортування прямим злиттям.

Алгоритм сортування прямим злиттям виконується наступним чином:

1) Послідовність a розбивається на дві половини, кожна з яких розбивається на дві половини; процес ділення закінчується, коли в наборах залишається по одному елементу.

2) Частини зливаються, при цьому поодинокі елементи утворюють впорядковані пари, впорядковані пари переходять в четвірки, потім у 8-ки і т.д. поки не буде впорядкована вся послідовність.

Результуючі набори, які виходять при злитті записуються в тимчасовий масив, який переписується в вихідний масив.

13.Дайте опис алгоритма сортування попарним злиттям.

Вхідна множина розглядається, як послідовність підмножин, кожне з яких складається з єдиного елемента і, отже, є вже впорядкованою. На першому проході кожні дві сусідні одноелементні множини зливаються в одну двоелементну впорядковану множину. На другому проході двоелементні множини зливаються в 4-елементні впорядковані множини і т.д. Зрештою виходить одна велика впорядкована множина.

14.Який метод лежить в основі побудови піраміди у вигляді масива без явної побудови дерева?

Почати побудову піраміди можна з a [k] ... a [n], k = [size / 2]. Ця частина масиву задовольняє властивості піраміди, так як в цьому діапазоні не існує таких значень індексів i і j, що б виконувалося i = 2i + 1 (або j = 2i + 2). Просто тому, що такі i, j знаходяться за кордоном масиву. Слід зауважити, що властивість піраміди зберігається лише в рамках вихідного, основного масиву a [0] ... a [n]. Далі частина масиву, що має властивість піраміди, розширюється. Для цього додається по одному елементу за крок. Наступний елемент на кожному кроці додавання - той, що стоїть перед уже готовою частиною. Щоб при додаванні елементу зберігалася

пірамідальність, виконується наступна дія з розширення піраміди a[i + 1] .. a[n] на елемент a[i] вліво:

• аналізуються нащадки зліва і справа - в масиві це елементи a[2i + 1] і a[2i + 2] і вибирається найбільший з них;

• якщо цей елемент більше a[i], його міняють місцями з a[i];

• після чого виконується перехід до кроку 2, маючи на увазі нове положення a[i] у масиві. І

Інакше - кінець процедури. Новий елемент ніби то «просівається» крізь піраміду.

15.Дайте опис алгоритма пірамідального сортування.

На першому етапі будується піраміда на основі масиву. Як видно з властивостей піраміди, в корені завжди знаходиться максимальний елемент. Звідси випливає алгоритм другого етапу:

1) Верхній елемент піраміди a [0] ... a [n] (перший в масиві) міняється місцями з останнім;

2) Після цього розглядається масив a [0] ... a [n-1]. Для перетворення його в піраміду досить просіяти лише новий перший елемент.

Крок 1 повторюється, поки що обробляється частина масиву не зменшиться до одного елемента.