Міністерство освіти і науки України

Державний університет «Житомирська політехніка»

Факультет інформаційно-комп’ютерних технологій

Кафедра комп’ютерних наук

**Звіт**

з лабораторних робіт

# з дисципліни «Алгоритми та структури даних»

Виконав студент 1-го курсу, групи ІПЗ-20-3

спеціальності 121 «Інженерії програмного забезпечення»

М. О. Башманівський

Керівник Р. В. Петросян

Житомир – 2021

**ЗМІСТ**

**Лабораторна робота №1** 2

**Лабораторна робота №2** 10

[**Лабораторна робота №3**](#_bookmark2) 15

**Лабораторна робота №4** 30

**Лабораторна робота №5** 40

**Лабораторна робота №6** 49

**Лабораторна робота №7** 59

**Лабораторна робота № 1**

Робота з базовими типами даних

**Мета роботи:** отримати практичні навички по роботі з базовими типами даних (простими і складними типами даних)

**Хід роботи**

1. Записати і заповнити структуру даних зберігання поточного часу (включаючи секунди) і дату в найбільш компактному вигляді. Визначити обсяг пам'яті, яку займає змінна даного типу. Порівняти зі стандартною структурою tm (time.h). Вивести вміст структури в зручному вигляді для користувача на дисплей.

Створюю структуру данних з полями та задаю об’єм пам’яті. Створюю змінну створеного типу(структури) та ініціалізую поля. Порівнюю розмір створеної структури та та існуючої структури tm.

Лістинінг програми:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <math.h>

#include <stdio.h>

#include <Windows.h>

#include <stdlib.h>

#include <time.h>

struct date

{

unsigned short Rik : 5;

unsigned short Mis : 4;

unsigned short Tyzhd : 3;

unsigned short Den : 5;

unsigned short Godyn : 5;

unsigned short Hvyl : 6;

unsigned short Sec : 6;

};

int main()

{

date a;

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

a.Rik = 21;

a.Mis = 6;

a.Tyzhd = 4;

a.Den = 31;

a.Godyn = 21;

a.Hvyl = 45;

a.Sec = 56;

printf("%d.%d.%d %d:%d:%d |%d|", a.Den, a.Mis, a.Rik, a.Godyn, a.Hvyl, a.Sec, a.Tyzhd);

int k, g;

k = sizeof(date);

g = sizeof(tm);

printf("\nsizeof(date) = %d", k);

printf("\nsizeof(tm) = %d", g);

}

Результат виконання програми:

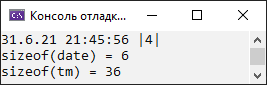


Рисунок №1 – Результат виконання програми

1. Реалізувати введення цілочисельного значення типу signed short. Визначити знак і значення, використовуючи: 1) структури даних та об'єднання; 2) побітові логічні операції.

Створюю об’єднання для збереження числа типу unsigned short. При записі числа в структуру, воно розбивається на 16 бітів, останній із яких вказує на знак.

Лістинінг програми:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <math.h>

#include <stdio.h>

#include <Windows.h>

#include <stdlib.h>

#include <time.h>

union num {

struct n {

unsigned short a0 : 1;

unsigned short a1 : 1;

unsigned short a2 : 1;

unsigned short a3 : 1;

unsigned short a4 : 1;

unsigned short a5 : 1;

unsigned short a6 : 1;

unsigned short a7 : 1;

unsigned short a8 : 1;

unsigned short a9 : 1;

unsigned short a10 : 1;

unsigned short a11 : 1;

unsigned short a12 : 1;

unsigned short a13 : 1;

unsigned short a14 : 1;

unsigned short a15 : 1;

}numer;

signed short count;

}num1;

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

puts("Введіть число: ");

scanf\_s("%hd", &num1);

if (num1.count == 0)

{

printf("Значення заданого числа: ");

printf("%d\n", num1.count);

exit(0);

}

if (num1.numer.a15 == 0) puts("Знак числа +");

else if(num1.numer.a15 == 1) puts("Знак числа -");

printf("Значення заданого числа: ");

printf("%d\n", num1.count);

}

Результат виконання програми:

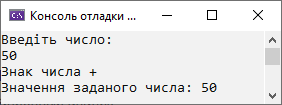


Рисунок №1.1 – Результат виконання програми

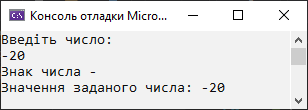


Рисунок №1.2 – Результат виконання програми

1. Виконати операції: а) 5 + 127; б) 2-3; в) -120-34; г) (unsigned char) (- 5); д) 56 & 38; е) 56 | 38. Всі значення (константи) повинні зберігатися в змінних типу signed char. Виконати перевірку результату в ручну. Пояснити результат, використовуючи двійкову систему числення.

Лістинінг програми:

#include <time.h>

#include <stdio.h>

#include <Windows.h>

#include <cstdlib>

#include <math.h>

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

signed char res\_asm;

int Nom;

printf("Оберіть номер завдання від а(1) до е(6): "); scanf\_s("%d", &Nom);

printf("\n");

switch (Nom)

{

case 1:

{

printf("%d) 5 + 127\n", Nom);

\_\_asm

{

mov al, 5;

mov bl, 127;

add al, bl;

mov cl, al;

mov res\_asm, cl;

}

printf("%d", res\_asm);

printf("\nПереповнення біту!");

break;

}

case 2:

{

printf("%d) 2-3\n", Nom);

\_\_asm

{

mov al, 2;

mov bl, 3;

sub al, bl;

mov cl, al;

mov res\_asm, cl;

}

printf("%d", res\_asm);

break;

}

case 3:

{

printf("%d) -120-34\n", Nom);

\_\_asm

{

mov al, -120;

mov bl, 34;

sub al, bl;

mov cl, al;

mov res\_asm, cl;

}

printf("%d", res\_asm);

printf("\nПереповнення біту!");

break;

}

case 4:

{

printf("%d) (unsigned char(0-255)) (- 5)\n", Nom);

int x;

int y = (-5);

printf("Введіть x:");

scanf\_s("%d", &x);

if (x > y)

{

printf("%d < %d", y, x);

}

else

printf("%d > %d", y, x);

break;

}

case 5:

{

printf("%d) 56 & 38\n", Nom);

int x = 56, y = 38;

if (x > y)

{

printf("%d > %d", x, y);

}

else

printf("%d < %d", x, y);

break;

}

case 6:

{

printf("%d) 56 | 38\n", Nom);

int x = 56, y = 38;

if (x > y)

{

printf("%d", x);

}

else

printf("%d", y);

break;

}

}

}

1. 5 + 127 = 132 (Максимальні значення для signed char -127 – 127. Тому відбувається переповнення і сума досягнувши максимуму в 127 починає заповнюватись з початку із -127.)
2. 2 – 3 = -1
3. -120 – 34 = -154 Відбувається переповнення і різниця досягнувши максимуму в -127 починає іти з максимуму із 127.
4. (unsigned char)(-5) (Число -5 перетворюєтся в незнаковий тип, проте змінна в яку записується результат знакова, тому результат знову перетворюється в знаковий)
5. 56 & 38 = 32 (56 = 1110002,) (38 = 1001102.)
6. 56 | 38 = 62 (Побітове або. 56 = 1110002, 38 = 1001102. Якщо хоча б один біт 1, в результаті 1, Результат 1111102 = 62)

Результат виконання програми:

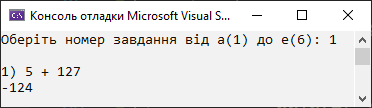


Рисунок №1.3 – Результат виконання програми

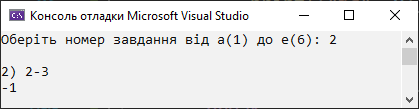


Рисунок №1.4 – Результат виконання програми

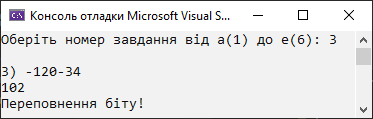


Рисунок №1.5 – Результат виконання програми

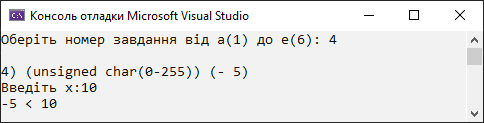


Рисунок №1.6 – Результат виконання програми

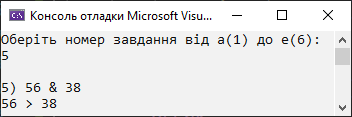


Рисунок №1.7 – Результат виконання програми

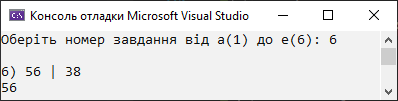


Рисунок №1.8 – Результат виконання програми

1. Записати і заповнити структуру даних (об'єднання) для зберігання дійсного числа типу float в найбільш компактному вигляді. Реалізувати відображення на дисплей: 1) значення побитово; 2) значення побайтово; 3) знака, мантиси і ступінь значення. Виконати перевірку результату в ручну. Визначити обсяг пам'яті, яку займає змінна користувацького типу.

Лістинінг програми:

#include<stdio.h>

#include<windows.h>

union data {

float num;

struct {

unsigned char b0 : 1;

unsigned char b1 : 1;

unsigned char b2 : 1;

unsigned char b3 : 1;

unsigned char b4 : 1;

unsigned char b5 : 1;

unsigned char b6 : 1;

unsigned char b7 : 1;

unsigned char b8 : 1;

unsigned char b9 : 1;

unsigned char b10 : 1;

unsigned char b11 : 1;

unsigned char b12 : 1;

unsigned char b13 : 1;

unsigned char b14 : 1;

unsigned char b15 : 1;

unsigned char b16 : 1;

unsigned char b17 : 1;

unsigned char b18 : 1;

unsigned char b19 : 1;

unsigned char b20 : 1;

unsigned char b21 : 1;

unsigned char b22 : 1;

unsigned char b23 : 1;

unsigned char b24 : 1;

unsigned char b25 : 1;

unsigned char b26 : 1;

unsigned char b27 : 1;

unsigned char b28 : 1;

unsigned char b29 : 1;

unsigned char b30 : 1;

unsigned char b31 : 1;

} bits;

struct {

unsigned char byte0;

unsigned char byte1;

unsigned char byte2;

unsigned char byte3;

} bytes;

};

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

data number;

printf("Введіть число: "); scanf\_s("%f", &number.num);

printf("\nВведене число типу float в десятковій системі: number = %f\n", number.num);

printf("\nВведене число в двійковій системі: number = %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d\n", number.bits.b31, number.bits.b30, number.bits.b29, number.bits.b28, number.bits.b27, number.bits.b26, number.bits.b25, number.bits.b24, number.bits.b23, number.bits.b22, number.bits.b21, number.bits.b20, number.bits.b19, number.bits.b18, number.bits.b17, number.bits.b16, number.bits.b15, number.bits.b14, number.bits.b13, number.bits.b12, number.bits.b11, number.bits.b10, number.bits.b9, number.bits.b8, number.bits.b7, number.bits.b6, number.bits.b5, number.bits.b4, number.bits.b3, number.bits.b2, number.bits.b1, number.bits.b0);

printf("\nВведене число в шістнадцятковій системі: number = 0x%x%x%x%x\n", number.bytes.byte3, number.bytes.byte2, number.bytes.byte1, number.bytes.byte0);

if (number.bits.b31 == 0)

printf("\nЗнак числа: +.\n");

else

printf("\nЗнак числа: -.\n");

printf("\nМантиса: %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d%d %d%d%d\n", number.bits.b22, number.bits.b21, number.bits.b20, number.bits.b19, number.bits.b18, number.bits.b17, number.bits.b16, number.bits.b15, number.bits.b14, number.bits.b13, number.bits.b12, number.bits.b11, number.bits.b10, number.bits.b9, number.bits.b8, number.bits.b7, number.bits.b6, number.bits.b5, number.bits.b4, number.bits.b3, number.bits.b2, number.bits.b1, number.bits.b0);

printf("\nСтупінь числа: %d%d%d%d %d%d%d%d\n", number.bits.b30, number.bits.b29, number.bits.b28, number.bits.b27, number.bits.b26, number.bits.b25, number.bits.b24, number.bits.b23);

printf("\nОбсяг пам'яті, яку займає змінна %d біта\n", sizeof(number));

return 0;

}

Результат виконання програми:

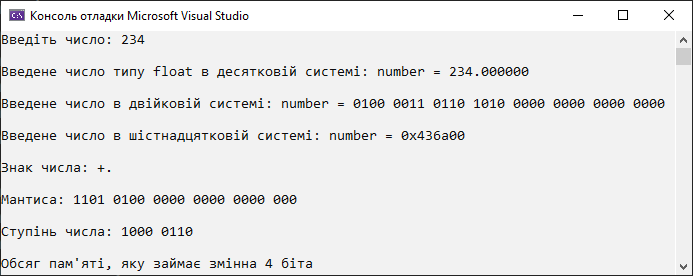


Рисунок №1.9 – Результат виконання програми

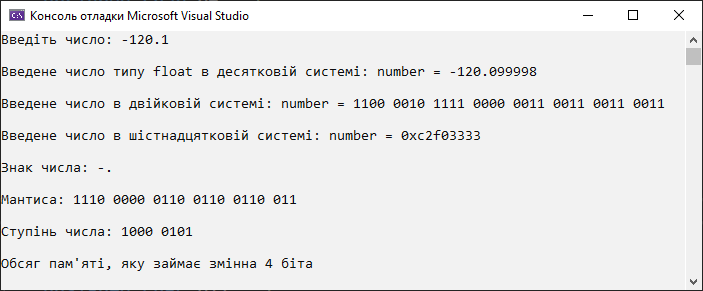


Рисунок №1.10 – Результат виконання програми

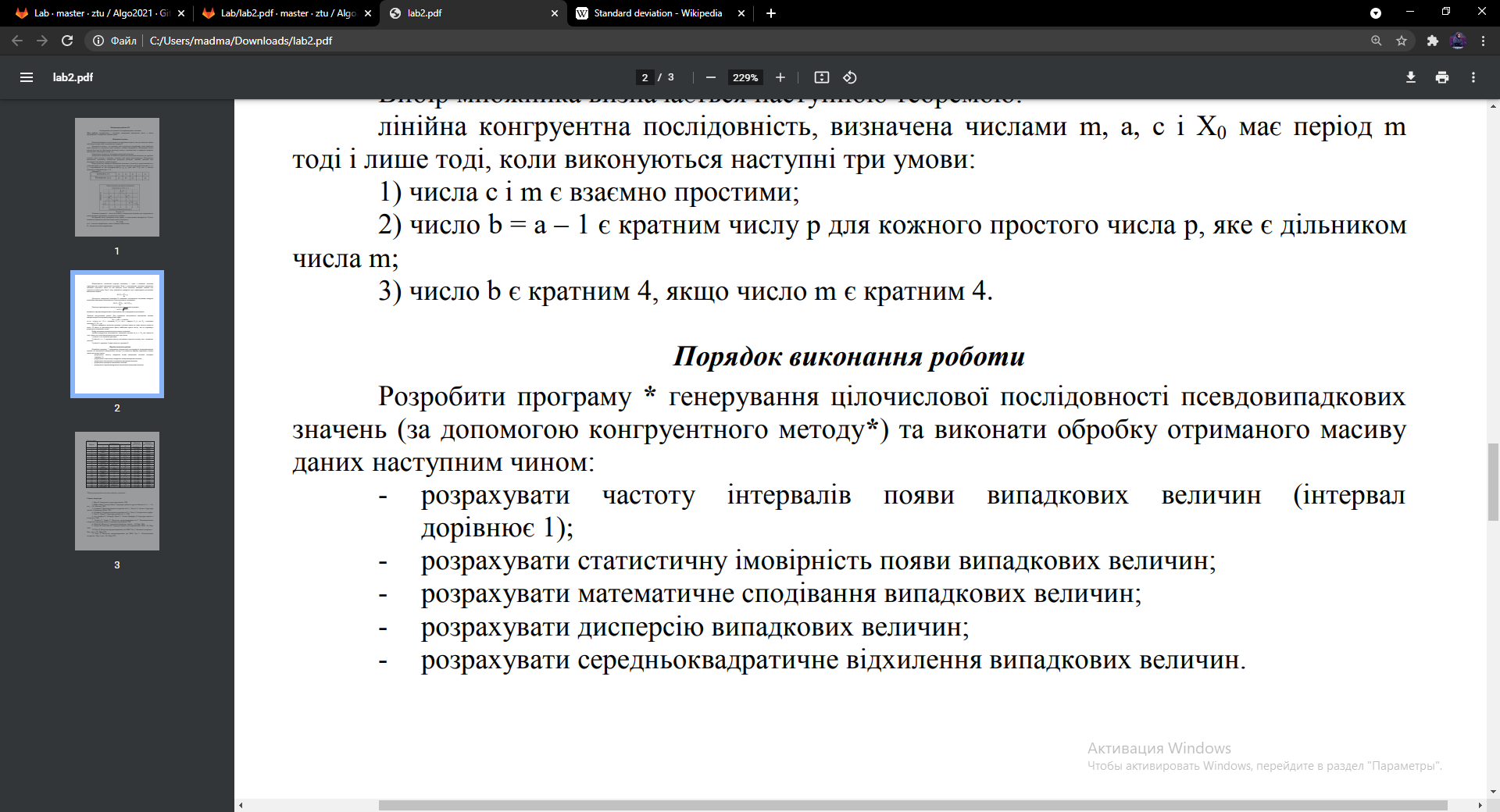
**Лабораторна робота № 2**

«Генерування послідовності псевдовипадкових значень»

**Мета роботи:** ознайомитись з методами генерування випадкових чисел, а також формуванням та обробкою масивів даних.

**2.1 Хід роботи**

Завдання:



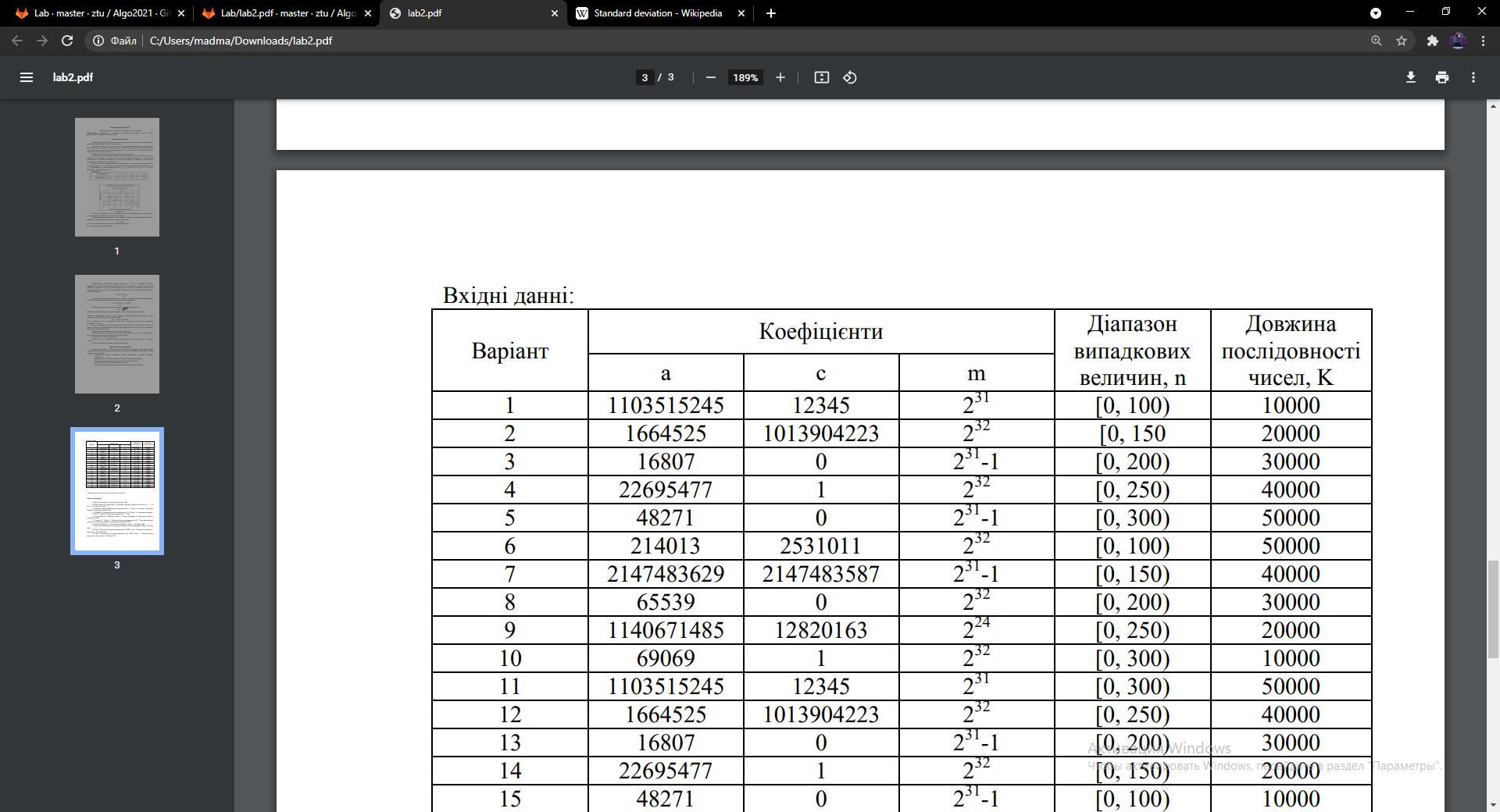


Рисунок 2 - Варіант завдання(3)

Лістинг програми:

#include <iostream>

#include "Windows.h"

#include "stdio.h"

#include "math.h"

#define N 30000

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

unsigned int x0, a = 16807, c = 0, k = 30000;

int arr[N], arr1[250];

float x = 1;

int m = (pow(2, 31) - 1);

printf("Введіть початкове значеня x:");

scanf\_s("%d", &x0);

for (int i = 0; i < k; i++)

{

x0 = (x0 \* a + c) % m;

x = ((float)x0 / m \* 200);

arr[i] = x;

printf(" %d", arr[i]);

}

printf("\n");

printf("\nЧастота інтервалів появи випадкових величин:\n");

int f = 0, g = 0;

float arr2[200];

for (int h = 0; h < 200; h++)

{

for (int i = 0; i < k; i++)

{

if (arr[i] == h)

f++;

}

printf(" (%d) | %d\n", h, f);

arr1[h] = f;

f = 0;

}

printf("\n");

printf("Статистична ймовірність появи випадкових величин:\n");

for (int h = 0; h < 200; h++)

{

for (int i = 0; i < k; i++)

{

if (arr[i] == h)

g++;

}

printf(" %d | %lf\n", h, (float)g / 30000);

arr2[h] = (float)g / 30000;

g = 0;

}

double Expvalue = 0;

for (int i = 0; i < 200; i++)

{

Expvalue += arr[i] \* arr2[i];

}

printf("\n");

printf("Математичне сподівання появи випадкових величин: %lf\n", Expvalue);

double variance = 0;

for (int i = 0; i < k; i++)

{

variance += pow(arr[i] - Expvalue, 2) \* arr[i];

}

printf("Дисперсія випадкових велечин: %lf\n", variance);

double deviation = 0;

deviation = sqrt(variance);

printf("Середньоквадратичне відхілення випадкової велечини: %lf\n", deviation);

}

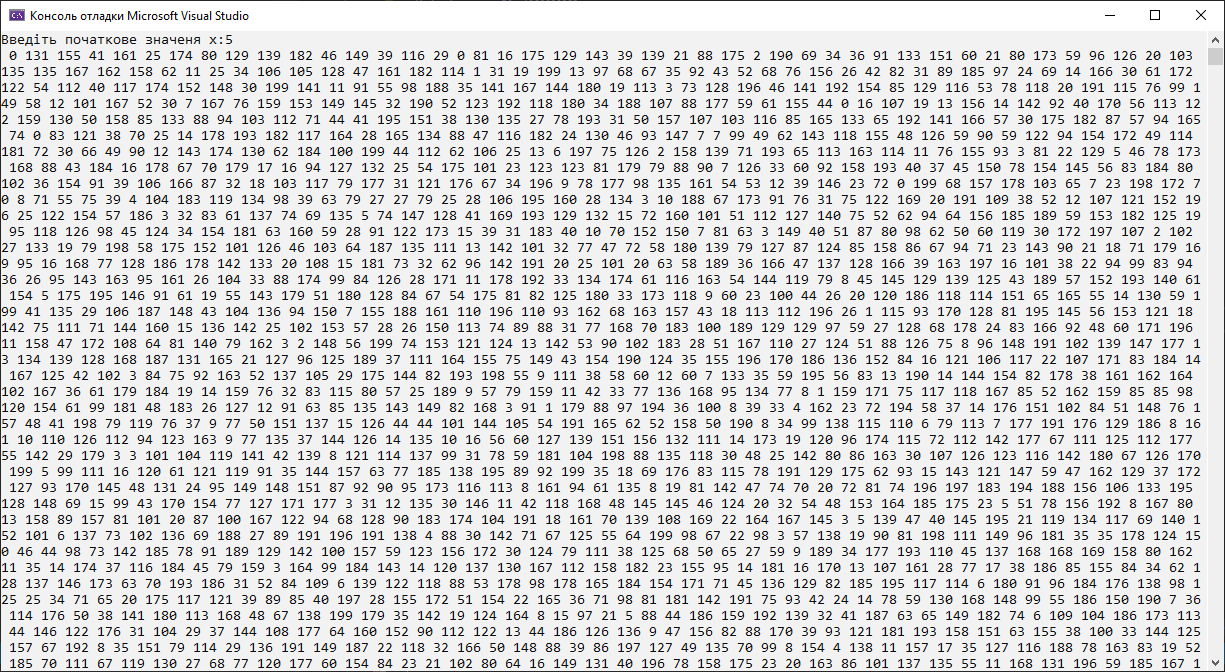


Рисунок 2.1 - Результат генерування цілочислової послідовності псевдовипадкових значень

Частота інтервалів появи випадкових величин (інтервал дорівнює 1):

int f = 0, g = 0;

float arr2[200];

for (int h = 0; h < 200; h++)

{

for (int i = 0; i < k; i++)

{

if (arr[i] == h)

f++;

}

printf(" (%d) | %d\n", h, f);

arr1[h] = f;

f = 0;

}

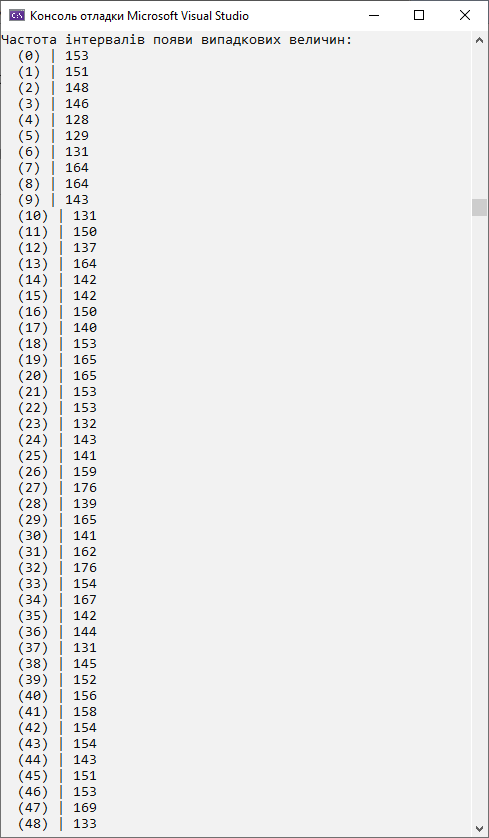


Рисунок 2.2 - Частота інтервалів появи випадкових величин

Статистична імовірність появи випадкових величин:

for (int h = 0; h < 200; h++)

{

for (int i = 0; i < k; i++)

{

if (arr[i] == h)

g++;

}

printf(" %d | %lf\n", h, (float)g / 30000);

arr2[h] = (float)g / 30000;

g = 0;

}

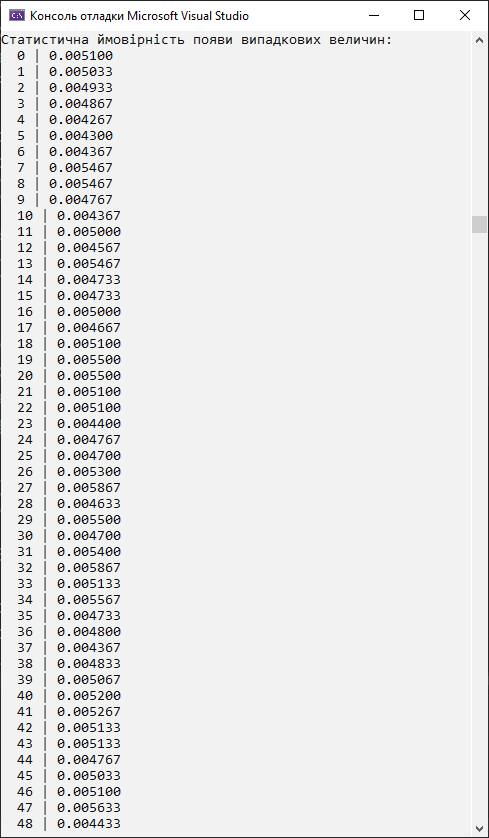


Рисунок 2.3 - Статистична імовірність появи випадкових величин

Математичне сподівання випадкових величин:

double Expvalue = 0;

for (int i = 0; i < 200; i++)

{

Expvalue += arr[i] \* arr2[i];

}

Дисперсія випадкових величин:

double variance = 0;

for (int i = 0; i < k; i++)

{

variance += pow(arr[i] - Expvalue, 2) \* arr[i];

}

Середньоквадратичне відхилення випадкових величин:

double deviation = 0;

deviation = sqrt(variance);

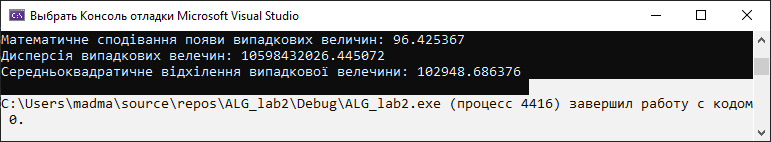


Рисунок 2.4 - Математичне сподівання випадкових величин , дисперсія випадкових величин , середньоквадратичне відхилення випадкових величин

**Лабораторна робота № 3**

Оцінка часової складності алгоритмів

**Мета роботи:** набуття навичок дослідження часової складності алгоритмів і визначення її асимптотичних оцінок.

**3.1 Хід роботи**

Завдання 1: Написати програму для табулювання наступних функцій: f(n)=n; f(n)=log(n); f(n)=n⋅log(n); f(n)=n2 ; f(n)=2n ; f(n)=n!. Табулювання виконати на відрізку [0, 50] з кроком 1. Побудувати графіки функцій (за допомогою Excel) в одній декартовій системі координат. Значення осі ординат обмежити величиною 500.

Лістинг програми:

#include<stdio.h>

#include<windows.h>

#include <iostream>

#include <chrono>

#include <math.h>

#define GETTIME std::chrono::steady\_clock:: now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

unsigned \_\_int64 factorial(int count)

{

unsigned \_\_int64 vlfct = 1;

for (int i = 1; i <= count; i++)

vlfct = vlfct \* i;

return vlfct;

}

void func\_1()

{

int y = 0;

for (int x = 0; x <= 50; x++)

{

y = x;

printf("\ny = %d , x = %d", y, x);

}

}

void func\_2()

{

float y = 0;

for (int x = 0; x <= 50; x++)

{

y = log(x);

printf("\ny = %f , x = %d", y, x);

}

}

void func\_3()

{

float y = 0;

for (int x = 0; x <= 50; x++)

{

y = x \* log(x);

printf("\ny = %f , x = %d", y, x);

}

}

void func\_4()

{

int y = 0;

for (int x = 0; x <= 50; x++)

{

y = pow(x,2);

printf("\ny = %d , x = %d", y, x);

}

}

void func\_5()

{

long long int y = 0;

for (int x = 0; x <= 50; x++)

{

y = pow(2, x);

printf("\ny = %lld , x = %d", y, x);

}

}

void func\_6()

{

unsigned \_\_int64 y = 0;

for (int x = 0; x <= 50; x++)

{

y = factorial(x);

printf("\ny = %llu , x = %d", y, x);

}

}

int main()

{

auto begin = GETTIME();

func\_6();

auto end = GETTIME();

auto elapsed\_ns = CALCTIME(end - begin);

printf("\nThe time: %lld ns\n", elapsed\_ns.count());

}

Результат виконання програми:

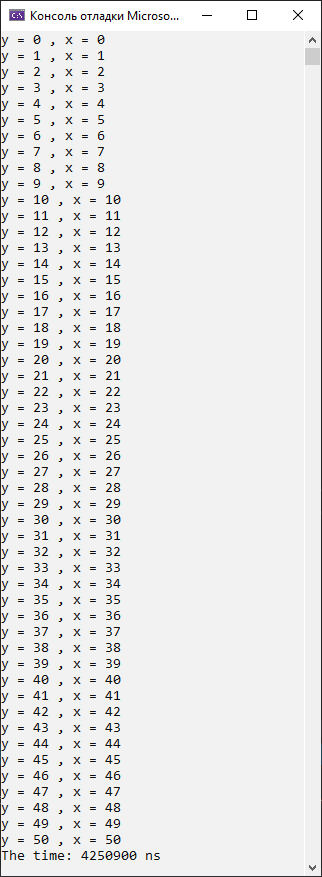


Рисунок 4.1 - Табулювання функції f(n)=n

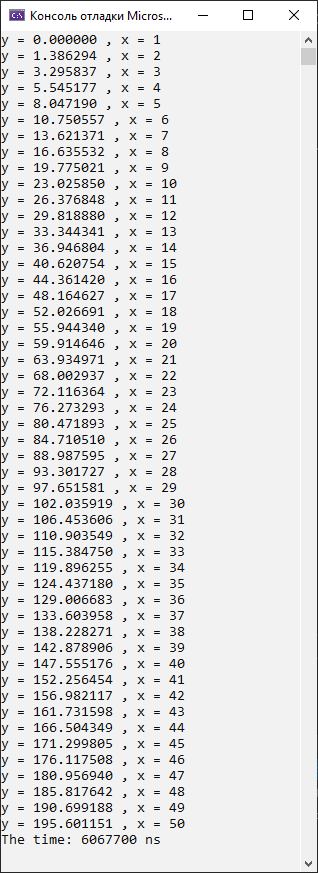


Рисунок 4.2 – Табулювання функції f(n)=log(n)

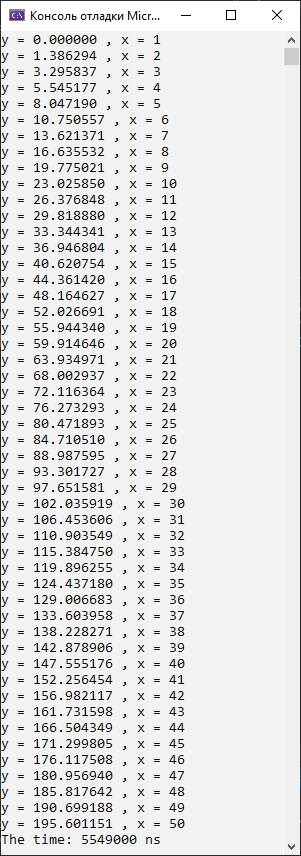


Рисунок 4.3 – Табулювання функції f(n)=n⋅log(n)

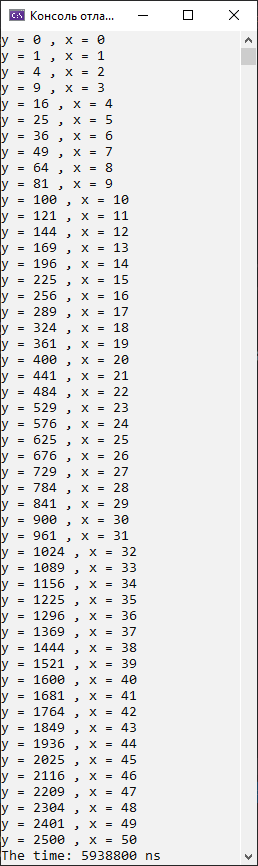


Рисунок 4.4 – Табулювання функції f(n)=n^2

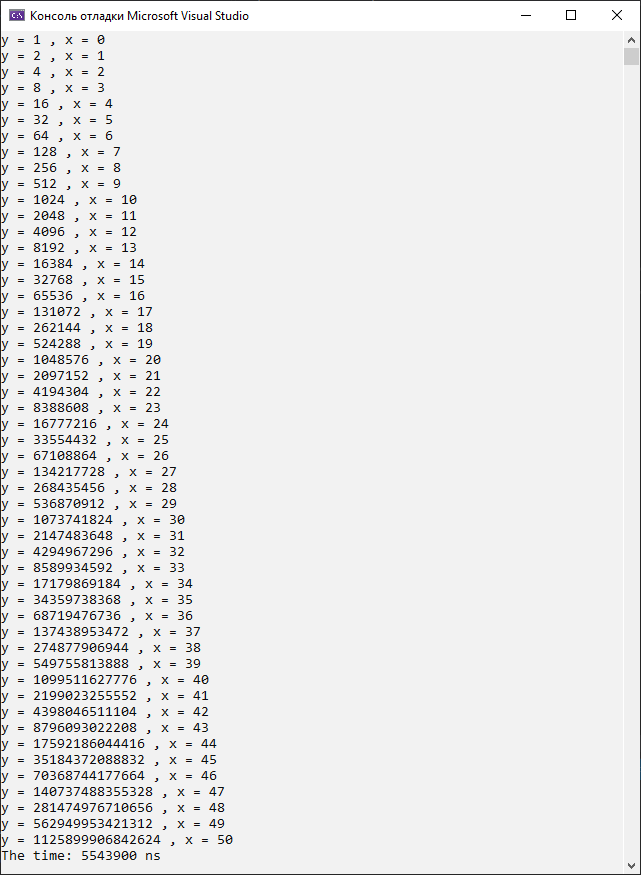


Рисунок 4.5 – Табулювання функції f(n)=2^n

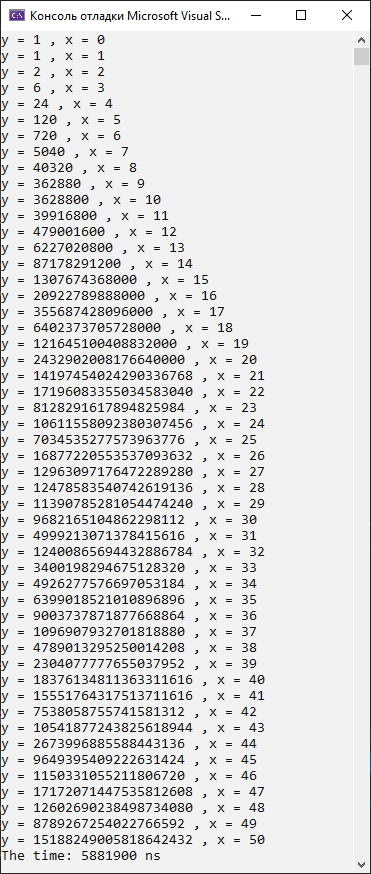


Рисунок 4.6 – Табулювання функції f(n)=n!

При табулюванні функції f(n)=n! відбувається переповнення після х=22 так як unsigned long long має діапазон значень від 0 до 18 446 744 073 709 551 615.

Графіки функцій

Рисунок 4.7 – Графік функції f(n)=n

Рисунок 4.8 – Графік функції f(n)=log(n)

Рисунок 4.9 – Графік функції f(n)=n \* log(n)

Рисунок 4.10 – Графік функції f(n)=n^2

Рисунок 4.11 – Графік функції f(n)=2^n

Рисунок 4.12 – Графік функції f(n)=n!

Завдання 2: Напишіть програму згідно індивідуального завдання (таблиця 3.1 та таблиця 3.2). Виміряти час виконання функцій та побудувати графіки за допомогою Excel. Провести аналіз і оцінку часової складності алгоритмів. Порівняти практично отримані результати з оцінкою часової складності алгоритмів.

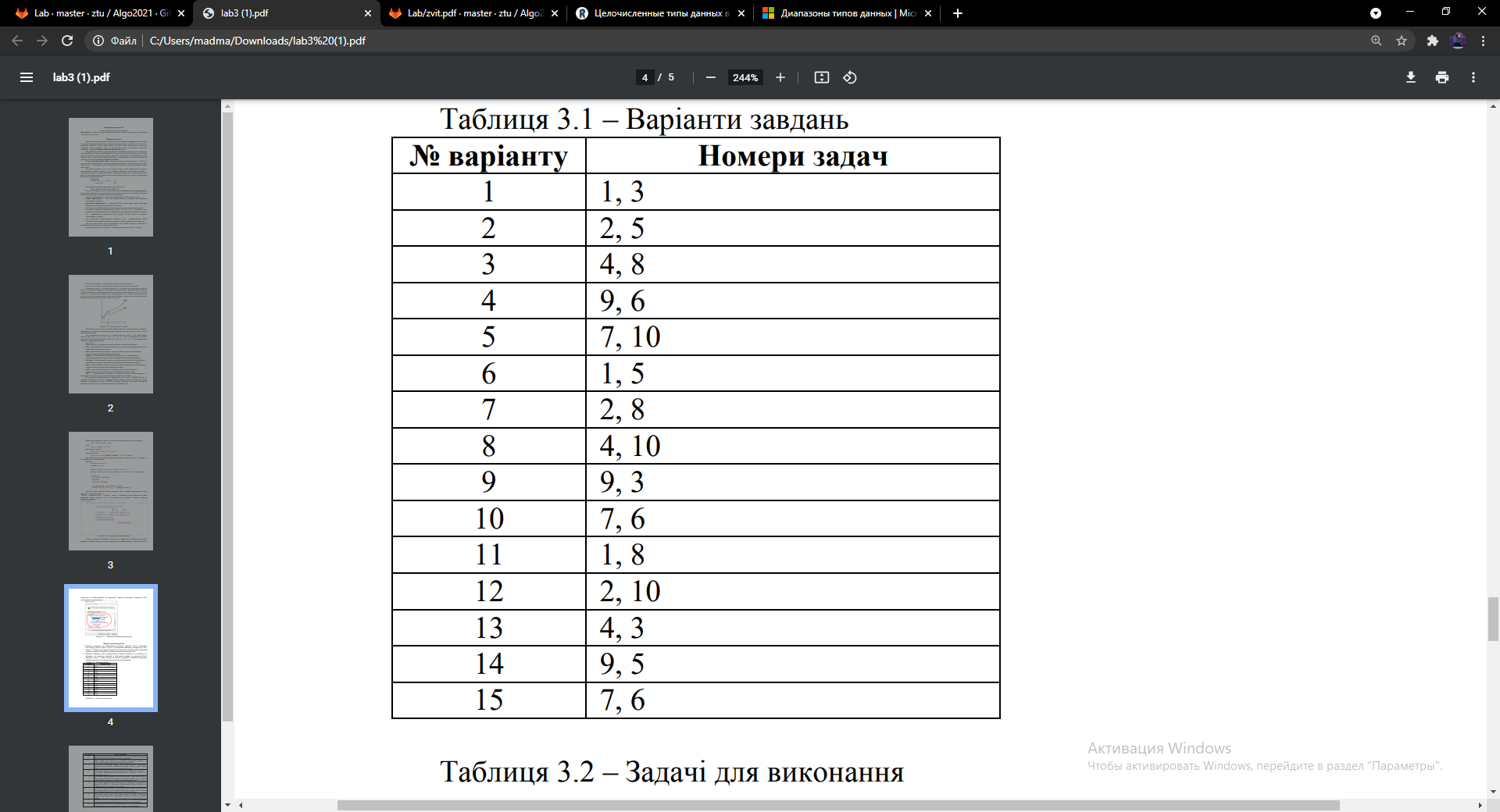


Рисунок 4.13 – Номер варіанту

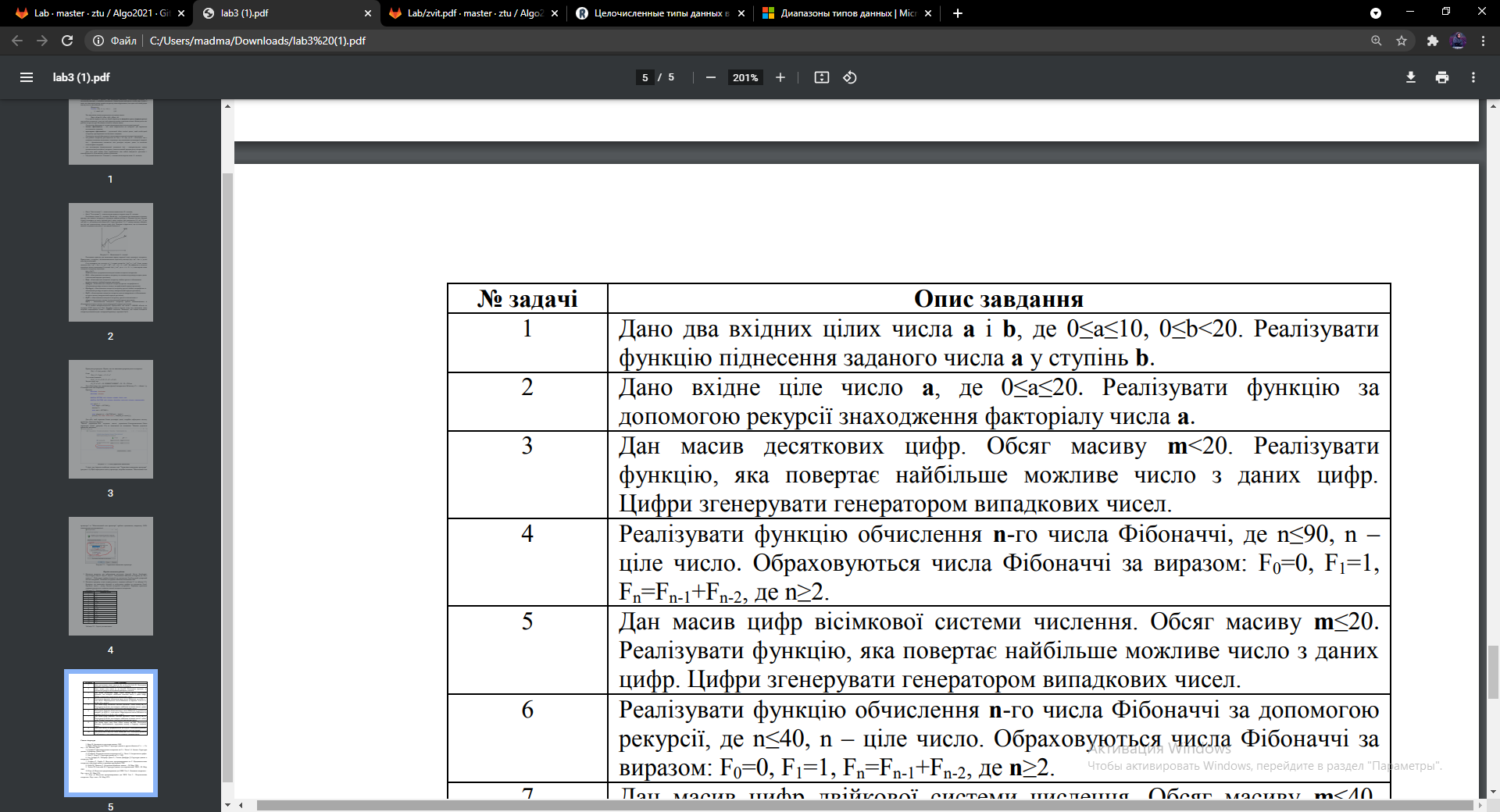


Рисунок 4.14 - Варіант завдання 1

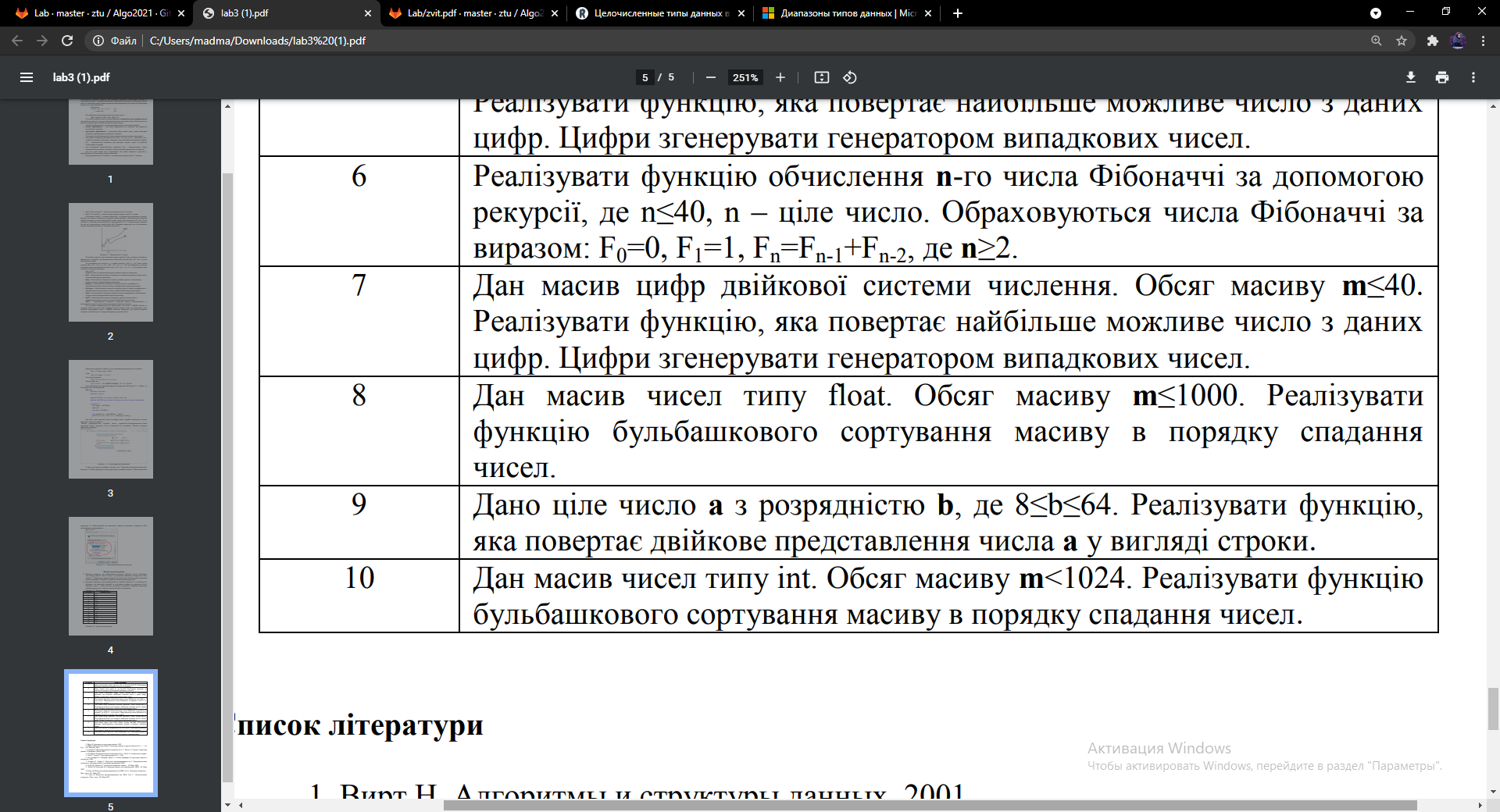


Рисунок 4.15 - Варіант завдання 2

Лістинг програми 1:

#include<stdio.h>

#include<windows.h>

#include <iostream>

#include <chrono>

#include <math.h>

#define GETTIME std::chrono::steady\_clock:: now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

int main()

{

auto begin = GETTIME();

printf("Послідовність Фібоначчі : \n0\n1");

unsigned long long a1 = 0, a2 = 1, temp;

for (int n = 2; n <= 90; n++)

{

temp = a2;

a2 = a1 + a2;

a1 = temp;

printf(" \n%llu", a2);

}

auto end = GETTIME();

auto elapsed\_ns = CALCTIME(end - begin);

printf("\nThe time: %lld ns\n", elapsed\_ns.count());

}

Результат виконання програми:

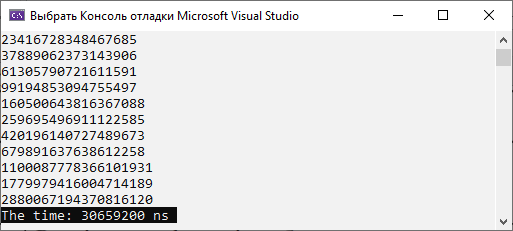


Рисунок 4.15 – Результат виконання програми 1

Аналіз і оцінка часової складності алгоритмів:

t(10) = 535900 ns

T(n) = Сt⋅ t(n), де t(n) = O(n)

T(n) ≤ Сt⋅ Сxg(n) = Сt⋅ Сx⋅ n

t2/t1 ≤ Сt⋅ Сx⋅ n2/Сt⋅ Сx⋅ n1 = n2/n1.

t2 ≤ t1⋅ n2/n1 = 535900 \* 80/10 = 4287200 ns

За стільки часу має виконуватись алгоритм – послідовність Фібоначчі з 80 елементів

Послідовність Фібоначчі має оцінку складності алгоритму – O(n) так як зі збільшенням вхідних даних час роботи збільшується лінійно.

Практично отримане значення(середнє арефметичне 5 дослідів) - 9887200 ns

Рисунок 4.16 - Графік часу і вхідних даних

Лістинг програми 2:

#include<stdio.h>

#include<windows.h>

#include <iostream>

#include <chrono>

#include <math.h>

#define GETTIME std::chrono::steady\_clock:: now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

#define MAX\_SIZE 1000

void bubbleSort(double\* array, int size)

{

for (int i = 0; i < size - 1; i++)

{

for (int j = (size - 1); j > i; j--)

{

if (array[j - 1] < array[j])

{

double temp = array[j - 1];

array[j - 1] = array[j];

array[j] = temp;

}

}

}

}

int main()

{

auto begin = GETTIME();

srand(time(0));

double array[MAX\_SIZE];

for (int i = 0; i < MAX\_SIZE; i++)

{

array[i] = rand() + (rand() % RAND\_MAX) / 1000.0;

}

bubbleSort(array, MAX\_SIZE);

for (int i = 0; i < MAX\_SIZE; i++)

{

printf("%f ", array[i]);

}

auto end = GETTIME();

auto elapsed\_ns = CALCTIME(end - begin);

printf("\nThe time: %lld ns\n", elapsed\_ns.count());

}

Аналіз і оцінка часової складності алгоритмів:

t(1000) = 39569100 ns

T(n) = Сt⋅ t(n), де t(n) = O(n^2)

T(n) ≤ Сt⋅ Сxg(n) = Сt⋅ Сx⋅ n^2

t2/t1 ≤ Сt⋅ Сx⋅ n2^2/Сt⋅ Сx⋅ n1^2 = n2^2/n1^2.

t2 ≤ t1⋅ n2^2/n1^2 = 39569100 \* 5000^2/1000^2 = 989227500 ns

За такий час має відсортуватися масив з 5000 дійсних чисел

Сортування масиву має оцінку складності алгоритму – O(n^2) , так як бульбашкове сортування має вкладений цикл , кількість операцій буде залежати від розмірності масива.

Практично отримане значення(середнє арефметичне 5 дослідів) - 281980100 ns

Рисунок 4.17 - Графік часу і вхідних даних

Результат виконання програми:



Рисунок 4.18 – Результат виконання програми 2

**Лабораторна робота № 4**

«Зв’язний список, стек, черга. Зворотній польський запис»

**Мета роботи:** ознайомитися з основами роботи з двозв’язним списком, однозв’язним списком, стеком та чергою. Розробити основні функції для обчислення арифметичного виразу, записаного з використанням зворотного польського запису.

**4.1 Хід роботи**

Завдання 1: Розробити всі основні функції роботи з двозв’язним списком (доповнити функції, які відсутні у прикладі, що розглядався на лекції для тих, хто претендує на оцінку "відмінно".).

1. Створюю заголовочний файл clist.h для збереження функцій та полів потрібних для реалізації списку.

Лістинг файлу .h:

#ifndef \_CLIST\_H\_

#define \_CLIST\_H\_

#include <stdlib.h>

#include <stdbool.h>

typedef int elemtype; // Тип элемента списка

struct elem {

elemtype\* value; // Значение переменной

struct elem\* next; // Ссылка на следующий элемент списка

struct elem\* prev; // Ссылка на предыдущий элемент списка

};

struct myList {

struct elem\* head; // Первый элемент списка

struct elem\* tail; // Последний элемент списка

int size; // Количество элементов в списке

};

typedef struct elem cNode;

typedef struct myList cList;

cList\* createList(void); // створення списку

void deleteList(cList\* list); // видалення списку

bool isEmptyList(cList\* list); // перевірка на пустоту

int pushFront(cList\* list, elemtype\* data); //елемент спочатку

int popFront(cList\* list, elemtype\* data); //видалення першого елемента

int pushBack(cList\* list, elemtype\* data); // елемент в кінці

int popBack(cList\* list, elemtype\* data); // видалення елементу в кінці

cNode\* getNode(cList\* list, int index); // отримання елементу

void printList(cList\* list, void (\*fun)(elemtype\*)); // вивід списку

int insertNode(cList\* list, size\_t index, elemtype\* data);// додавання елементу по індексу

int deleteNode(cList\* list, size\_t index); // видалення по індексу

#endif //\_CLIST\_H\_

1. Реалізую функції списку у файлі clist.cpp

Динамічно виділяю пам’ять для списку.

// Создание пустого списка

cList\* createList(void) {

cList\* list = (cList\*)malloc(sizeof(cList));

if (list) {

list->size = 0;

list->head = list->tail = NULL;

}

return list;

}

Звільняю пам’ять в циклі для кожного елемента.

// Удаление списка

void deleteList(cList\* list) {

cNode\* head = list->head;

cNode\* next = NULL;

while (head) {

next = head->next;

free(head);

head = next;

}

free(list);

list = NULL;

}

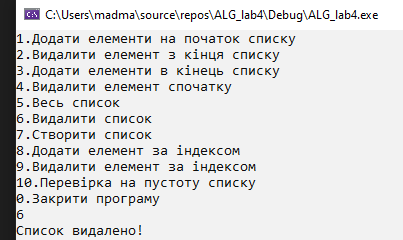


Рисунок 3.1 – Видалення списку

Повертає true або false , якщо наявні елементи в вершині або в кінці списку

// Проверка списка на пустоту

bool isEmptyList(cList\* list) {

return ((list->head == NULL) || (list->tail == NULL));

}

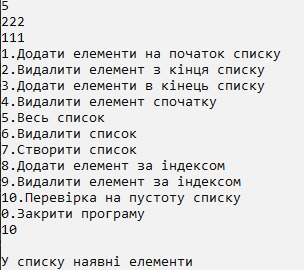


Рисунок 3.2 – Перевірка на наявність вузлів

Виділяю динамічно пам’ять для нового вузла , елемент стає в вершині списку

// Добавление нового узла в начало списка

int pushFront(cList\* list, elemtype\* data) {

cNode\* node = (cNode\*)malloc(sizeof(cNode));

if (!node) {

return(-1);

}

node->value = data;

node->next = list->head;

node->prev = NULL;

if (!isEmptyList(list)) {

list->head->prev = node;

}

else {

list->tail = node;

}

list->head = node;

list->size++;

return(0);

}

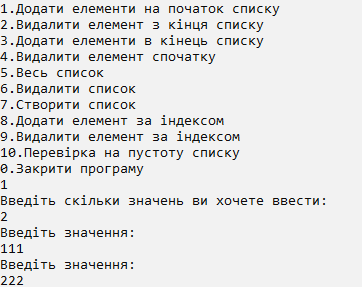


Рисунок 3.3 – Новий вузол спочатку

Видалення верхнього елемента

// Извлечение узла из начала списка

int popFront(cList\* list, elemtype\* data) {

cNode\* node;

if (isEmptyList(list)) {

return(-2);

}

node = list->head;

list->head = list->head->next;

if (!isEmptyList(list)) {

list->head->prev = NULL;

}

else {

list->tail = NULL;

}

data = node->value;

list->size--;

free(node);

return(0);

}

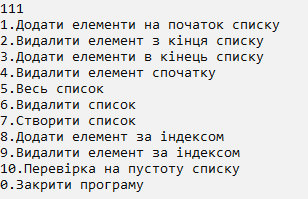


Рисунок 3.4 – Видалення елементу , верхнього = 222

Виділяю пам’ять та через показник вставляю елемент в кінець списку.

// Добавление нового узла в конец списка

int pushBack(cList\* list, elemtype\* data) {

cNode\* node = (cNode\*)malloc(sizeof(cNode));

if (!node) {

return(-3);

}

node->value = data;

node->next = NULL;

node->prev = list->tail;

if (!isEmptyList(list)) {

list->tail->next = node;

}

else {

list->head = node;

}

list->tail = node;

list->size++;

return(0);

}

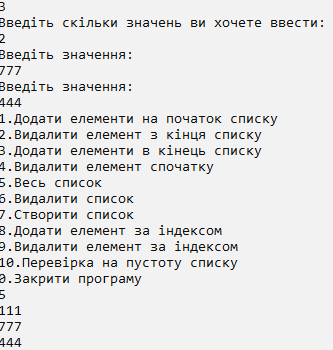


Рисунок 3.5 – Додав 2 елементи в кінець

Видаляю останній елемент.

// Извлечение узла из конца списка

int popBack(cList\* list, elemtype\* data) {

cNode\* node = NULL;

if (isEmptyList(list)) {

return(-4);

}

node = list->tail;

list->tail = list->tail->prev;

if (!isEmptyList(list)) {

list->tail->next = NULL;

}

else {

list->head = NULL;

}

data = node->value;

list->size--;

free(node);

return(0);

}

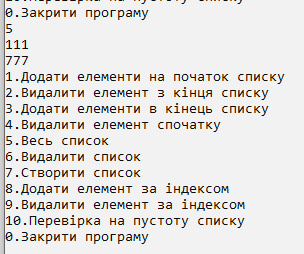


Рисунок 3.6 – Видалив останній елемент = 444

Шукаю потрібний індекс в циклі ділячи список

// Чтение произвольного узла списка

cNode\* getNode(cList\* list, int index)

{

cNode\* node = NULL;

int i;

if (index >= list->size) {

return (NULL);

}

if (index < list->size / 2) {

i = 0;

node = list->head;

while (node && i < index) {

node = node->next;

i++;

}

}

else {

i = list->size - 1;

node = list->tail;

while (node && i > index) {

node = node->prev;

i--;

}

}

return node;

}

Через показник звертаюсь до значення кожного елемента.

// Вывод списка в консоль

void printList(cList\* list, void (\*func)(elemtype\*)) {

cNode\* node = list->head;

if (isEmptyList(list)) {

return;

}

while (node) {

func(node->value);

node = node->next;

}

}

//Добавить узел

int insertNode(cList\* list, size\_t index, elemtype\* data)

{

if (index == 0) {

pushFront(list, data);

return(0);

}

cNode\* node = getNode(list, index - 1); // елемент після якого потрібно вставити вузол

cNode\* insertNode = (cNode\*)malloc(sizeof(cNode)); // виділення пам'яті під вузол

if (!node) {

return(-5);

}

if (!insertNode) {

return(-6);

}

insertNode->value = data; // присвоюю значення через показник

insertNode->prev = node;

insertNode->next = node->next; //вставляю новий елемент

if (node->next) {

node->next->prev = insertNode;

}

node->next = insertNode;

if (!node->prev) {

list->head = node;

}

if (!node->next) {

list->tail = node;

}

list->size++; //збільшую розмірність

return(0);

}

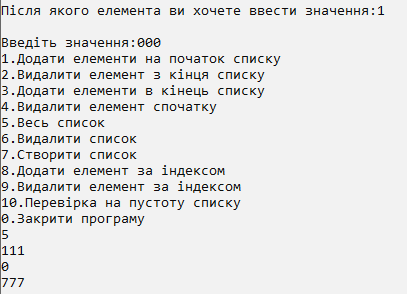


Рисунок 3.7 – Додав елемент = 0 , після першого елемента

// Видалення вузла за індексом з списку

int deleteNode(cList\* list, size\_t index) {

cNode\* node = getNode(list, index); // знаходжу елемент який потрібно видалити

if (!node) {

return(-7);

}

if (node->prev) {

node->prev->next = node->next; // змінюю порядок елементів

}

if (node->next) {

node->next->prev = node->prev;

}

if (!node->prev) {

list->head = node->next;

}

if (!node->next) {

list->tail = node->prev;

}

free(node); // звільняю пам'ять цього вузла

list->size--;

return(0);

}

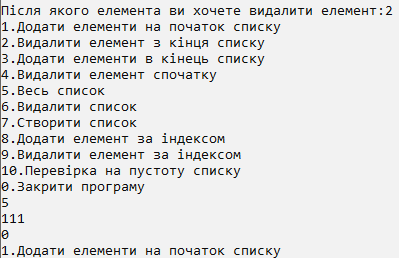


Рисунок 3.8 – Видалив елемент = 777

1. Реалізую меню для демонстрації функцій

#include <stdio.h>

#include <stdlib.h>

#include "clist.h"

#include "windows.h"

#define h 30

void printNode(elemtype\* value) {

printf("%d\n", \*((int\*)value));

}

int pushtoStack(int znach)

{

int f;

f = znach;

return f;

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

elemtype tmp;

int g = 1;

elemtype tmp1 = 0;

elemtype tmp3;

int tmp2;

int menu;

int mass[h];

int mass1[h];

int znach;

int znach1;

int j=0;

int l = 0;

cList\* mylist = createList();

do {

printf("1.Додати елементи на початок списку\n2.Видалити елемент з кінця списку\n3.Додати елементи в кінець списку\n4.Видалити елемент спочатку\n5.Весь список\n6.Видалити список\n7.Створити список\n8.Додати елемент за індексом\n9.Видалити елемент за індексом\n10.Перевірка на пустоту списку\n0.Закрити програму\n");

scanf\_s("%d", &menu);

switch (menu)

{

case 1:printf("Введіть скільки значень ви хочете ввести:\n");

scanf\_s("%d", &g);

for (int i = 0; i < g; i++)

{

printf("Введіть значення:\n");

scanf\_s("%d", &znach);

mass[i] = znach;

pushFront(mylist, &mass[i]);

}

break;

case 2:popBack(mylist, &tmp);

break;

case 3:printf("Введіть скільки значень ви хочете ввести:\n");

scanf\_s("%d", &g);

for (int i = 0; i < g; i++)

{

printf("Введіть значення:\n");

scanf\_s("%d", &znach1);

mass1[i] = znach1;

pushBack(mylist, &mass1[i]);

}

break;

case 4:popFront(mylist, &tmp);

break;

case 5:

printList(mylist, printNode);

break;

case 6:deleteList(mylist); printf("Список видалено!\n");

break;

case 7: {cList\* mylist = createList(); }

break;

case 8:printf("\nПісля якого елемента ви хочете ввести значення:"); scanf\_s("%d", &j);

printf("\nВведіть значення:"); scanf\_s("%d", &l); insertNode(mylist, j, &l);

break;

case 9:printf("\nПісля якого елемента ви хочете видалити елемент:");scanf\_s("%d", &j);

deleteNode(mylist, j);

break;

case 10: if (isEmptyList(mylist) == true)

{

printf("\nСписок пустий!!!\n");

}

else printf("\nУ списку наявні елементи\n");

break;

}

} while (menu != 0);

return 0;

}

Завдання 3: Розробити програму обчислення арифметичного виразу (використати зворотну 6 польську запис). Операнди у виразі розділяти пробілами. Операції: додавання (+), віднімання (-), множення (\*), ділення (/), зведення в ступінь (^), корінь квадратний (sqrt). Допускається використати готові класи роботи з динамічними структурами даних.

Реалізую функцію для результату виконаних дій:

static int Counts(string str)

{

string num = "";

int n;

int[] arr = new int[20];

int arri = 0;

foreach (char c in str)

{

if (int.TryParse(c.ToString(), out n))

{

num += c;

continue;

}

if (num != "")

{

arr[arri] = Convert.ToInt32(num);

num = "";

arri++;

}

if (!int.TryParse(c.ToString(), out n) && c != ' ')

{

switch (c)

{

case '+':

arr[arri - 2] = arr[arri - 1] + arr[arri - 2];

break;

case '-':

arr[arri - 2] = arr[arri - 2] - arr[arri - 1];

break;

case '\*':

arr[arri - 2] = arr[arri - 1] \* arr[arri - 2];

break;

case '/':

arr[arri - 2] = arr[arri - 1] / arr[arri - 2];

break;

case '^':

arr[arri - 2] = (int)Math.Pow(arr[arri - 2], arr[arri - 1]);

break;

case 's':

arr[arri - 1] = (int)Math.Sqrt(arr[arri - 1]);

arri++;

break;

}

arri--;

}

}

return arr[arri - 1];

}

Функція для запису в оберненому польському записі:

static string ConvertToPoly(string str)

{

string result = "";

string massSIM = "";

string num = "";

int n;

foreach (char c in str)

{

if (int.TryParse(c.ToString(), out n))

{

num += n.ToString();

}

if (!int.TryParse(c.ToString(), out n))

{

result += num;

if (c != ' ')

{

result += " ";

}

num = "";

if (c == 's')

{

massSIM += c;

}

else if (c == '+' || c == '-' || c == '\*' || c == '/')

{

if (massSIM.Length != 0)

{

while (massSIM[massSIM.Length - 1] == '\*' || massSIM[massSIM.Length - 1] == '/' || massSIM[massSIM.Length - 1] == '^' || massSIM[massSIM.Length - 1] == 's')

{

result += massSIM[massSIM.Length - 1];

result += " ";

massSIM = massSIM.Remove(massSIM.Length - 1);

if (massSIM.Length == 0)

{

break;

}

}

}

massSIM += c;

}

else if (c == '^')

{

massSIM += c;

}

}

}

if (num != "")

{

result += num;

result += " ";

}

while (massSIM.Length != 0)

{

result += massSIM[massSIM.Length - 1];

result += " ";

massSIM = massSIM.Remove(massSIM.Length - 1);

}

return result;

}

Виклик реалізованих функцій:

static void Main(string[] args)

{

Console.WriteLine($"{Counts(ConvertToPoly("23 + 2 + 1 + 5 + 6 - 2"))}");

Console.WriteLine($"{ConvertToPoly("23 + 2 + 1 + 5 + 6 - 2")}");

}

Результат роботи програми:

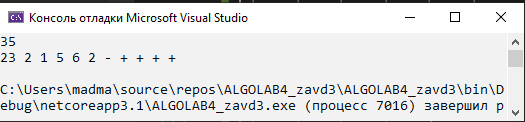


Рисунок 3.9 – Обернений польський запис

**Лабораторна робота № 5**

Прості методи сортування

**Мета роботи:** реалізація простих алгоритмів сортування та дослідження їх характеристик (швидкодія, необхідний обсяг пам'яті, застосування тощо).

**5.1 Хід роботи**

Завдання 1: Реалізувати алгоритми сортування:

a) сортування вибором (структура даних – двусвязний список);

б) сортування вставками (структура даних – масив);

в) сортування вставками (структура даних – двусвязний список);

**1(A):**

Сортування вибором(структура даних – двусвязний список);

**Алгоритм працює таким чином:**

1. Знаходить у списку найменше значення
2. Міняє його місцями із першим значеннями у списку
3. Повторює два попередніх кроки, доки список не завершиться (починаючи з наступної позиції)

Фактично, таким чином ми поділили список на дві частини: перша (ліва) — повністю відсортована, а друга (права) — ні.

Виконаний алгоритм:

#include <iostream>

#include <iomanip>

#include <locale.h>

#include "windows.h"

#include <cstdlib>

#include <iostream>

#include <chrono>

#define GETTIME std::chrono::steady\_clock::now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

using namespace std;

struct List {

int info;

List\* pred, \* next;

};

// Вставка елемента в двузв'язний список після останнього елемента

List\* InsertElementInList(List\* last, List\* p)

{

if (last && p)

{

p->pred = last;

p->next = last->next;

last->next = p;

p->next->pred = p;

return p; // Повертається адреса елемента

}

else

return NULL;

}

//Створення списку та запис елементів

void CreateList(List\*& head, List\*& tail)

{

head = new List;

tail = new List;

head->next = tail;

tail->pred = head;

int k;

List\* last = head;

for (int count = 0; count < 40000; ++count)

{

k = rand();

List\* p = new List;

p->info = k;

last = InsertElementInList(last, p);

}

return;

}

void PrintList(List\* head, List\* tail) // вивід списку

{

List\* p = head->next;

while (p != tail)

{

cout << "\t" << p->info;

p = p->next;

}

cout << endl;

return;

}

void SortListVYB(List\*& head, List\*& tail)

{

List\* last = tail;

tail->next = head;

head->pred = tail;

while (head->next != tail) // поки залишились елементи шукаємо мінімальний та видаляємо

{

List\* min = head->next, \* p = head->next;

while (p != tail)

{

if (p->info < min->info)

min = p;

p = p->next;

}

min->next->pred = min->pred; //видалення min елемента із списку

min->pred->next = min->next;

last = InsertElementInList(last, min); // додаємо найдений елемент

}

swap(head, tail);

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

auto begin = GETTIME();

auto end = GETTIME();

srand(time(NULL));

List\* head, \* tail = NULL;

CreateList(head, tail);

cout << "\nСписок\n" << endl;

PrintList(head, tail);

cout << "\nВідсортований список\n" << endl;

SortListVYB(head, tail);

PrintList(head, tail);

auto elapsed\_ns = CALCTIME(end - begin);

printf("The time: %lld ns\n", elapsed\_ns.count());

}

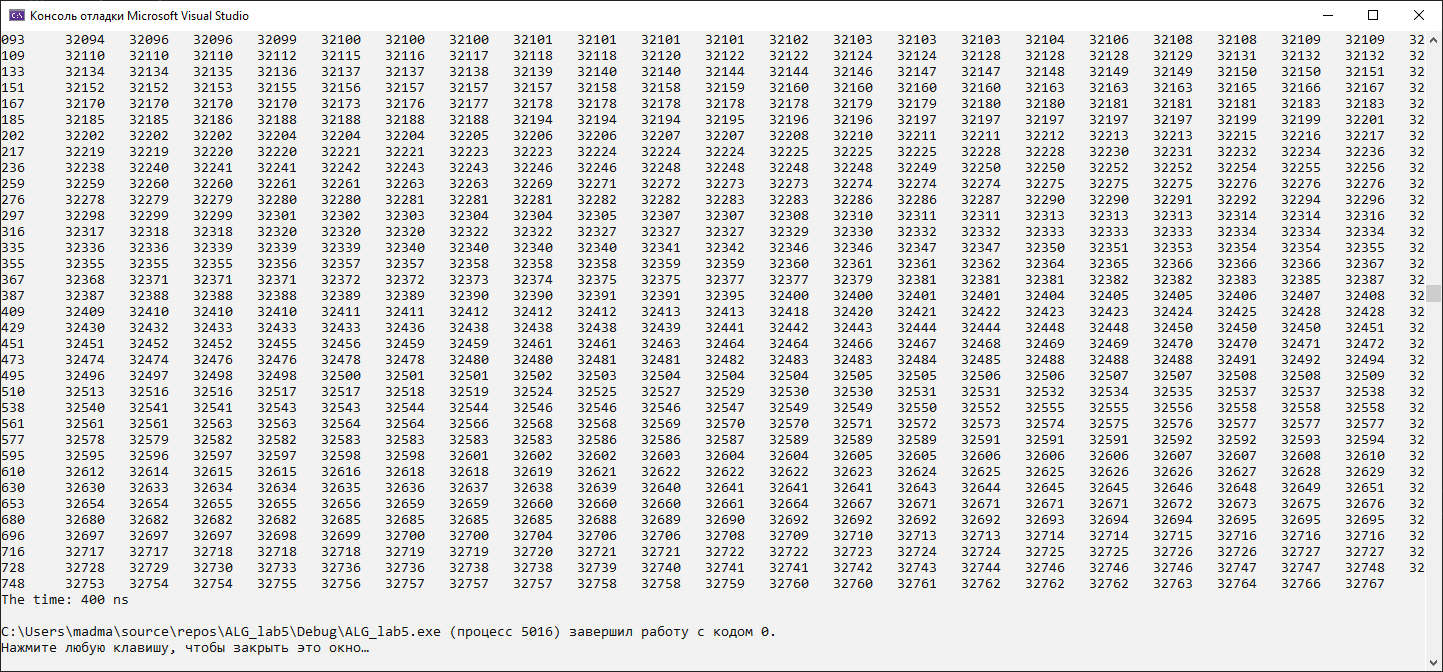


Рисунок 5.1 – Результат сортування вибором(списку)

**1(B):** сортування вставками (структура даних – масив);

**Алгоритм працює таким чином:**

На кожному кроці алгоритму ми вибираємо один з елементів вхідних даних і вставляємо його на потрібну позицію у вже відсортованому списку доти, доки набір вхідних даних не буде вичерпано. Метод вибору чергового елементу з початкового масиву довільний; може використовуватися практично будь-який алгоритм вибору. Зазвичай (і з метою отримання стійкого алгоритму сортування), елементи вставляються за порядком їх появи у вхідному масиві.

Виконаний алгоритм:

#include <stdio.h>

#include <iostream>

#include "windows.h"

#include "stdlib.h"

#include "time.h"

#include <ctime>

#include <cstdlib>

#include "math.h"

#include <chrono>

#define MAX\_SIZE 10000

#define GETTIME std::chrono::steady\_clock::now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

void InsertSort(int\* array, int size)

{

int c;

for (int i = 1; i < size; i++) //Проходимо по масиву знаходячи мінімальний елемент , вставляємо його у відсортовану частину

{

c = array[i];

for (int j = i - 1; j >= 0 && array[j] > c; j--)

{

array[j + 1] = array[j];

array[j] = c;

}

}

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

auto begin = GETTIME();

auto end = GETTIME();

srand(time(NULL));

int array[MAX\_SIZE];

for (int i = 0; i < MAX\_SIZE; i++)

{

array[i] = rand();

}

InsertSort(array, MAX\_SIZE);

for (int i = 0; i < MAX\_SIZE; i++)

{

printf("%d ", array[i]);

}

auto elapsed\_ns = CALCTIME(end - begin);

printf("The time: %lld ns\n", elapsed\_ns.count());

return 0;

}

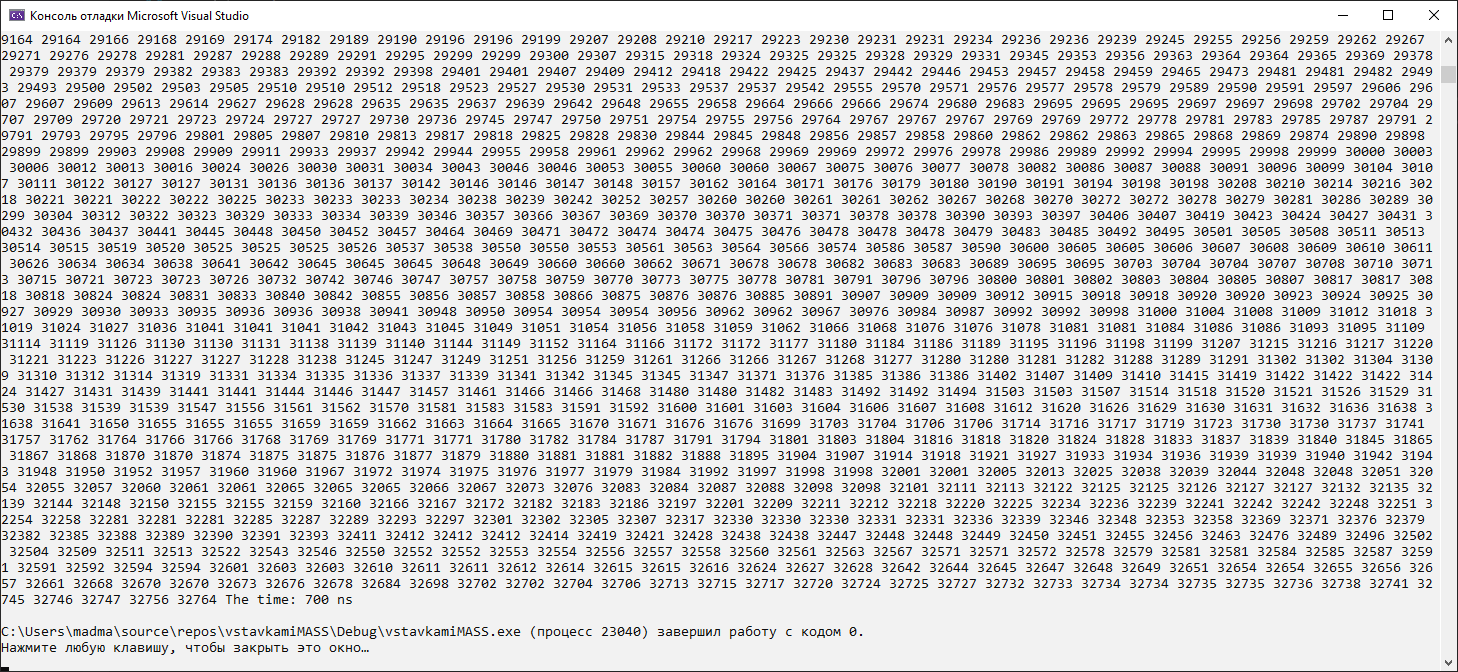


Рисунок 5.2 – Сортування вставками(масив)

**1(С):** Сортування вставками (структура даних – двусвязний список);

**Алгоритм працює таким чином:**

1)Створити порожній відсортоване (або результат) двусвязний список.

2) Пройдіть заданий двусвязний список, виконайте наступні дії для кожного вузла.

a) Вставити поточний вузол відсортованим способом в відсортоване (або результат) двусвязний список.

3) Замінити заголовок даного пов'язаного списку на заголовок відсортованого (або результату) списку.

#include <iostream>

#include <iomanip>

#include <locale.h>

#include "windows.h"

#include <cstdlib>

#include <iostream>

#include <chrono>

#define GETTIME std::chrono::steady\_clock::now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

using namespace std;

struct List

{

List() : Next(nullptr), Prev(nullptr) {}

int info;

List\* Next, \* Prev;

};

// Функция сортирует список методом вставки

void InsertionSort(List\* head)

{

List\* curr = nullptr, \* prev = nullptr;

for (curr = head->Next; curr->Next; curr = curr->Next)

{

int tmp = curr->info;

for (prev = curr->Prev; prev && prev->info > tmp; prev = prev->Prev)

{

prev->Next->info = prev->info;

}

prev->Next->info = tmp;

}

}

// Функция вставляет элемент p после элемента last

// возвращает адрес на вставленный в список элемент

List\* InsertElementIntoList(List\* last, List\* p)

{

if (last != NULL || p != NULL)

{

p->Prev = last;

p->Next = last->Next;

last->Next = p;

p->Next->Prev = p;

return p;

}

else

return NULL;

}

// Функция создает двусвязный список из n элементов

// через head и tail возвращает указатели головного и хвостового сторожей

void CreateRandomList(List\*& head, List\*& tail, int n)

{

head = new List;

tail = new List;

head->Next = tail;

tail->Prev = head;

srand(time(NULL));

List\* last; // Последний вставленный элемент в список

last = head;

for (int i = 0; i < n; i++)

{

List\* p = new List;

p->info = rand();

last = InsertElementIntoList(last, p);

}

}

// Функция выводит список на экран

void PrintList(List\* head, List\* tail)

{

List\* p = head->Next;

while (p != tail)

{

cout << "\t" << p->info;

p = p->Next;

}

cout << endl;

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

auto begin = GETTIME();

auto end = GETTIME();

srand(time(NULL));

List\* head, \* tail;

CreateRandomList(head, tail, 10000);

cout << "Исходный список: \n";

PrintList(head, tail);

InsertionSort(head);

cout << "Отсортированный список: \n";

PrintList(head, tail);

auto elapsed\_ns = CALCTIME(end - begin);

printf("The time: %lld ns\n", elapsed\_ns.count());

return 0;

}

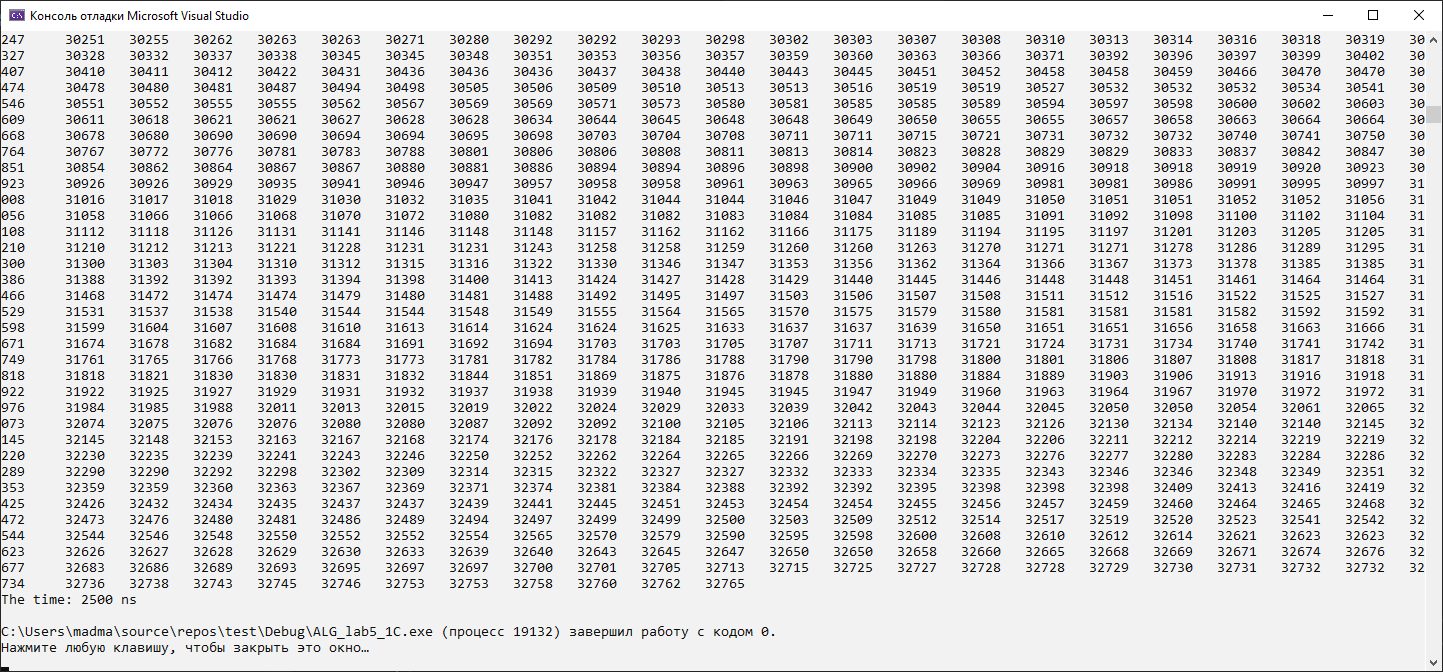


Рисунок 5.3 – Сортування вставками(список)

Графіки результатів вимірів часу:

Рисунок 5.4 – графік залежності часу від к-сті елементів

|  |  |
| --- | --- |
| Час | К-сть |
| 19545500 | 100 |
| 92021300 | 500 |
| 2,17E+08 | 1000 |
| 4E+08 | 2000 |
| 1,05E+09 | 5000 |
| 2,21E+09 | 10000 |
| Вставками(стр.) | |

|  |  |
| --- | --- |
| Час | К-сть |
| 1555800 | 100 |
| 10441600 | 500 |
| 18350000 | 1000 |
| 49426500 | 2000 |
| 1,6E+08 | 5000 |
| 3,28E+08 | 10000 |
| Вставками(мас) | |

|  |  |
| --- | --- |
| Час | К-сть |
| 20159900 | 100 |
| 97234100 | 500 |
| 2,29E+08 | 1000 |
| 4,64E+08 | 2000 |
| 1,08E+09 | 5000 |
| 2,31E+09 | 10000 |
| Вибором | |

Висновок: сортування ревлізоване з допомогою вставок має кращі показники складності , однак не практична у використанні з елементами до яких потрібен доступ.

**Лабораторна робота № 6**

Швидкі методи сортування

**Мета роботи:** реалізація швидких алгоритмів сортування та дослідження їх характеристик (швидкодія, необхідний обсяг пам'яті, застосування тощо).

**6.1 Хід роботи**

Завдання 1: Реалізувати алгоритми сортування у відповідності за таблицею 6.1:

a) пірамідальне сортування (структура даних – масив);

б) сортування Шелла (структура даних – масив);

в) сортування підрахунком (структура даних – масив).

****

**1(A):**

Пірамідальне сортування (структура даних – масив);

Складність: O(n logn)

Додаткова пам'ять: O(1)

**Суть сортування:**

Етап 1: Куча

Організація даних в спеціальну деревоподібну структуру – кучу.

Беремо перший елемент масиву і вважаємо що це корінь дерева – вузол 1 рівня. Наступні 2 елементи це – вузли 2 рівня , правий і лівий нащадки кореневого елемента. Наступні 4 елементи – це вузли 3 рівня , праві і ліві нащадки 2 та 3 елемента масиву і так далі поки не буде побудоване дерево(куча).

Для побудови використовують формули:

2 × **i** + 1 – лівий потомок

2 × **i** + 2 – правий потомок

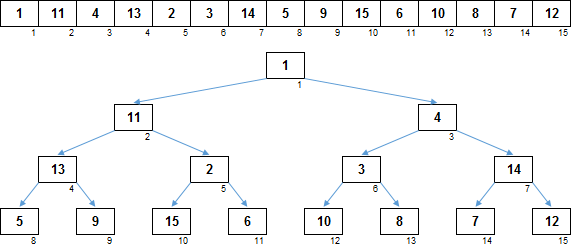


Рисунок 6.1 – Приклад побудованої кучі

Етап 2: Просейка

Просейка потрібна для того щоб відсортувати кучу.

Якщо елемент менший або більший чим його батьки(залежить від того max чи min куча) , то цей елемент потрібно перемістити наверх на один рівень.

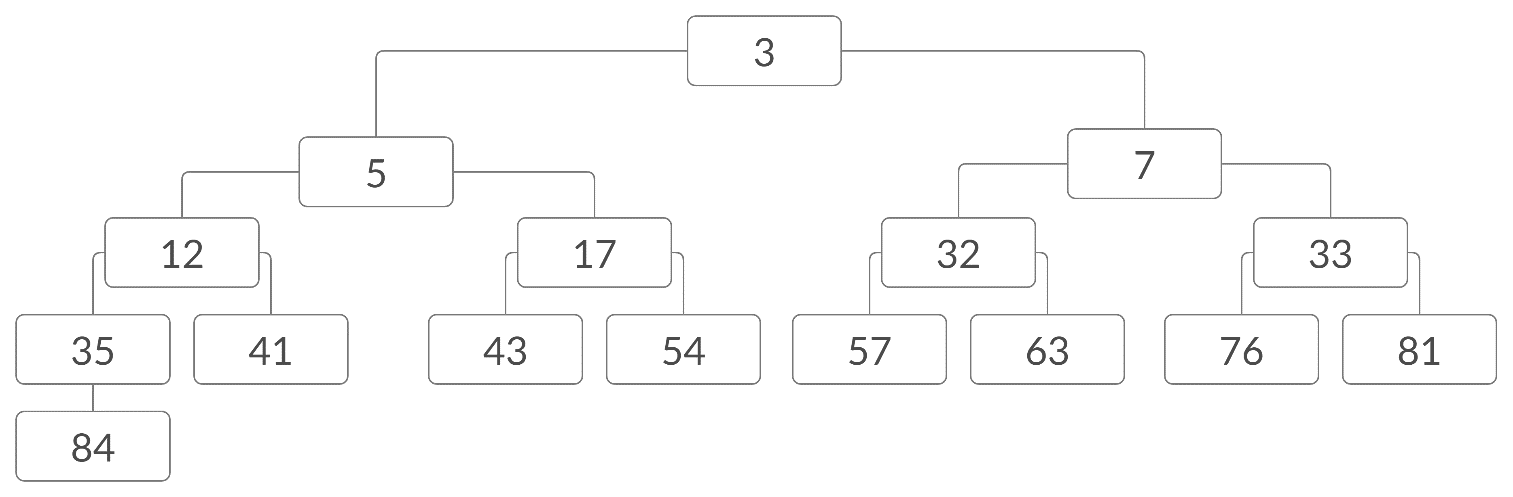
****

Рисунок 6.2 – Приклад відсортованої кучі(min)

Етап 3: Heap – sort.

Так як дані в масиві після першого етапу вдають із себе Сортувальне дерево, максимальний елемент знаходиться на першому місці в масиві. Перший елемент (він же максимум) міняємо з останнім елементом невідсортоване частини масиву місцями. Після цього обміну максимум виявився своєму остаточному місці, тобто максимальний елемент відсортований. Несортованими частина масиву перестала бути сортують деревом, але це виправляється одноразової просейкой - в результаті чого на першому місці масиву виявляється попередній за величиною максимальний елемент. Дії цього етапу знову повторюються для залишилася невпорядкованою області, до тих пір поки максимуми по черзі НЕ будуть переміщені на свої остаточні позиції.

**Реалізований алгоритм:**

#include <iostream>

#include <stdio.h>

#include "windows.h"

#include "stdlib.h"

#include "time.h"

#include <ctime>

#include <cstdlib>

#include "math.h"

#define MAX\_SIZE 1000

#include <chrono>

#include <random>

#define GETTIME std::chrono::steady\_clock::now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

#define left\_child(node) ( (node) \* 2 + 1 ) // Лівий потомок

#define right\_child(node) ( (node) \* 2 + 2 ) // Правий потомок

void swap(int\* array, int i, int j) // Функція для зміни порядку елементів , використовується в сортуванні

{

int tmp = array[i];

array[i] = array[j];

array[j] = tmp;

}

void heap\_it(int\* array, int length, int root) //Сортування кучі

{

int leftChild = left\_child(root);

int rightChild = right\_child(root);

int biggest = root;

if (leftChild < length && array[root] < array[leftChild]) // Переміщення елементів вверх по кучі

biggest = leftChild;

if (rightChild < length && array[biggest] < array[rightChild]) // Переміщення елементів вверх по кучі

biggest = rightChild;

if (biggest != root) // якщо найбільший елемент не у верху кучі , переміщуємо у верх , знову виконуємо heap\_it тільки вже з найбільшим елементом у верху.

{

swap(array, biggest, root);

heap\_it(array, length, biggest);

}

}

void make\_heap(int\* array, int length) //Створення кучі

{

int i = length / 2;

for (; i >= 0; --i)

heap\_it(array, length, i);

}

void heap\_sort(int\* array, int count) //Загальний алгоритм сортування

{

int last;

make\_heap(array, count);

for (last = count - 1; last > 0; --last)

{

swap(array, 0, last);

heap\_it(array, last, 0);

}

}

#define COUNT (1000)

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

auto begin = GETTIME();

auto end = GETTIME();

srand(time(0));

int i;

int array[COUNT];

for (int f = 0; f < COUNT; f++)

{

array[f] = 0 + rand() % (100 + 1); // [0 ; 100]

}

printf("unSorted:\n");

for (i = 0; i < COUNT; ++i)

printf("%d ", array[i]);

heap\_sort(array, COUNT); //Сортування

printf("\nSorted:\n");

for (i = 0; i < COUNT; ++i)

printf("%d ", array[i]);

printf("\n");

return 0;

auto elapsed\_ns = CALCTIME(end - begin);

printf("The time: %lld ns\n", elapsed\_ns.count());

}

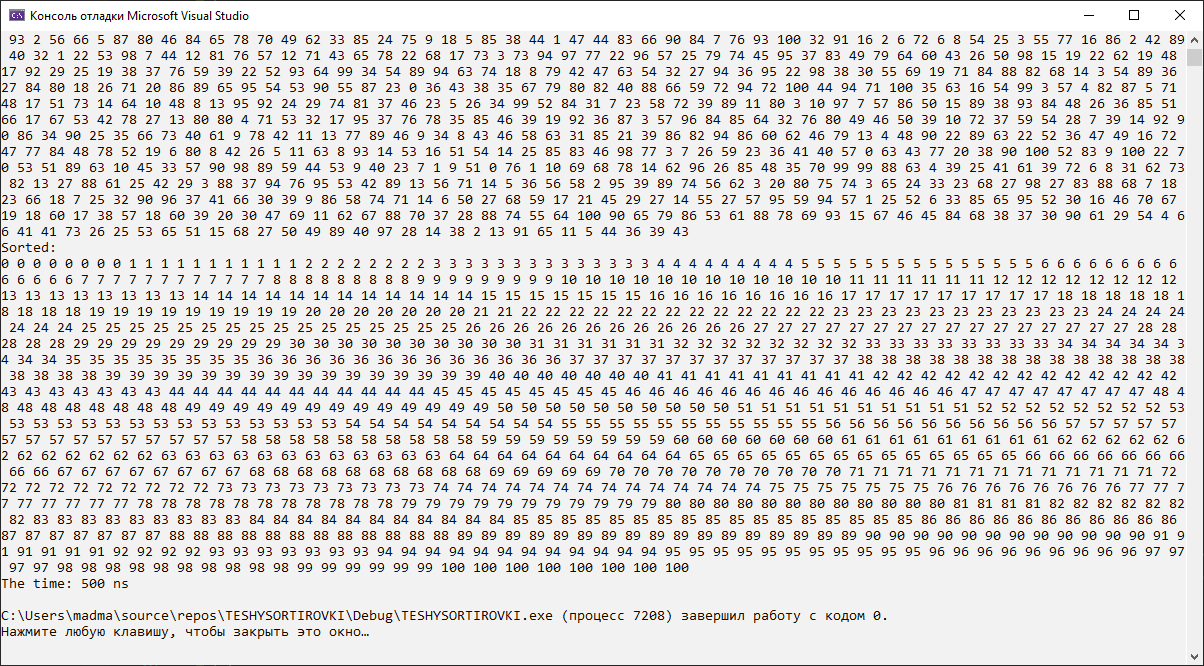


Рисунок 6.3 – Відсортований масив.

**1(B):**

сортування Шелла (структура даних – масив);

Складність:  O(n log² n)

Додаткова пам'ять: O(1)

**Суть сортування:**

Модифікований варіант сортування вставками.

Перше значення - це половина довжини сортованого масиву, друге - половина від попереднього і так далі, кожен раз округляючи значення до цілого числа.

З точки зору коду ми просто додаємо до коду сортування вставками ще один цикл із значенням зсуву, що зменшується кожну ітерацію, і починаємо сортування тепер від елемента з індексом, рівним цьому значенню.

Метод має властивість – приріст. В моєму варіанті розглядається формула Седжвіка.



Рисунок 6.4 – Формула Седжвіка

Приріст - відстань між сортованими елементами, в залежності від проходу.

При використанні таких приростів середня кількість операцій: O (n7 / 6), в гіршому випадку - порядку O (n4 / 3).

**Реалізований алгоритм:**

#include <iostream>

#include <stdio.h>

#include "windows.h"

#include "stdlib.h"

#include "time.h"

#include <ctime>

#include <cstdlib>

#include "math.h"

#define MAX\_SIZE 1000

#include <chrono>

#include <random>

#define GETTIME std::chrono::steady\_clock::now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

#define left\_child(node) ( (node) \* 2 + 1 ) // Лівий потомок

#define right\_child(node) ( (node) \* 2 + 2 ) // Правий потомок

void swap(int\* array, int i, int j) // Функція для зміни порядку елементів , використовується в сортуванні

{

int tmp = array[i];

array[i] = array[j];

array[j] = tmp;

}

void heap\_it(int\* array, int length, int root) //Сортування кучі

{

int leftChild = left\_child(root);

int rightChild = right\_child(root);

int biggest = root;

if (leftChild < length && array[root] < array[leftChild]) // Переміщення елементів вверх по кучі

biggest = leftChild;

if (rightChild < length && array[biggest] < array[rightChild]) // Переміщення елементів вверх по кучі

biggest = rightChild;

if (biggest != root) // якщо найбільший елемент не у верху кучі , переміщуємо у верх , знову виконуємо heap\_it тільки вже з найбільшим елементом у верху.

{

swap(array, biggest, root);

heap\_it(array, length, biggest);

}

}

void make\_heap(int\* array, int length) //Створення кучі

{

int i = length / 2;

for (; i >= 0; --i)

heap\_it(array, length, i);

}

void heap\_sort(int\* array, int count) //Загальний алгоритм сортування

{

int last;

make\_heap(array, count);

for (last = count - 1; last > 0; --last)

{

swap(array, 0, last);

heap\_it(array, last, 0);

}

}

#define COUNT (1000)

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

auto begin = GETTIME();

srand(time(0));

int i;

int array[COUNT];

for (int f = 0; f < COUNT; f++)

{

array[f] = 0 + rand() % (100 + 1); // [0 ; 100]

}

printf("unSorted:\n");

for (i = 0; i < COUNT; ++i)

printf("%d ", array[i]);

heap\_sort(array, COUNT); //Сортування

printf("\nSorted:\n");

for (i = 0; i < COUNT; ++i)

printf("%d ", array[i]);

printf("\n");

auto end = GETTIME();

auto elapsed\_ns = CALCTIME(end - begin);

printf("The time: %lld ns\n", elapsed\_ns.count());

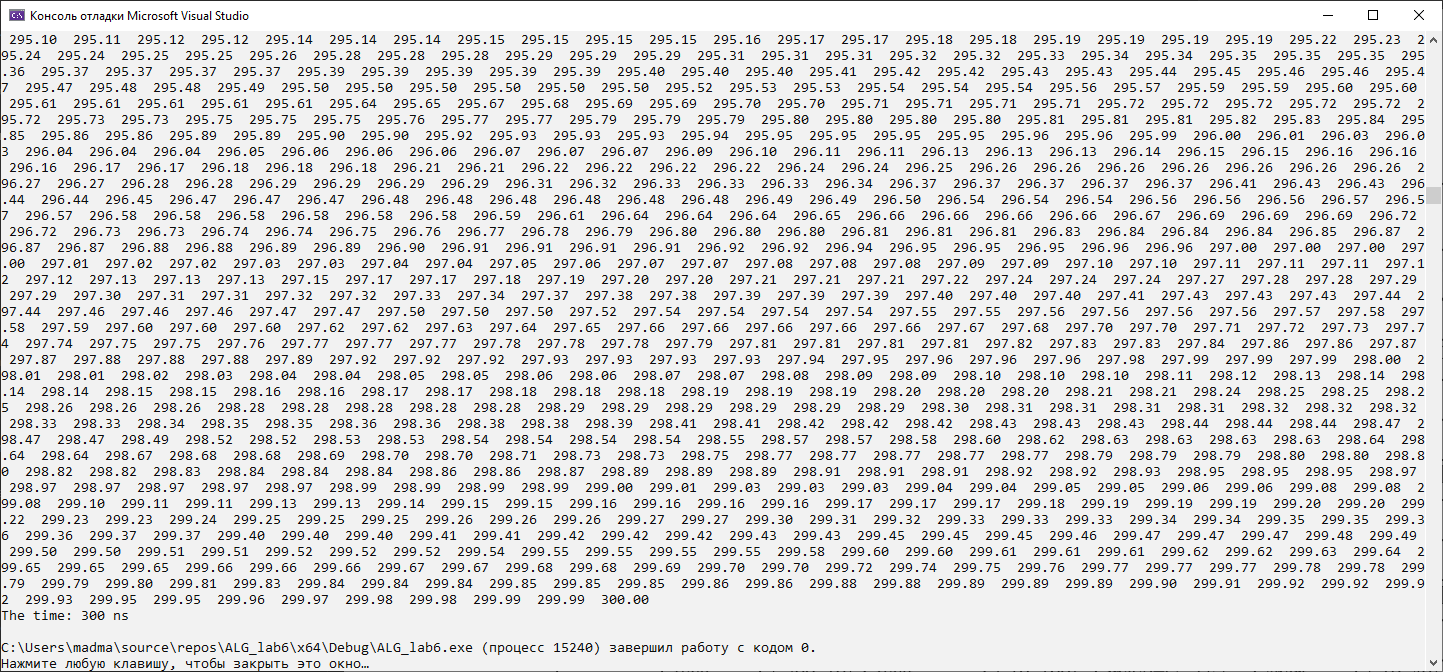
}

Рисунок 6.5 – Результат сортування

**1(C):**

Сортування підрахунком (структура даних – масив);

Складність: O(n+k) K – ширина діапвзону.

Додаткова пам'ять: В алгоритмі використовуються два додаткових масиви. Тому алгоритм потребує O(N+K) додаткової пам'яті.

**Суть сортування:** Ідея алгоритму полягає в наступному: спочатку підрахувати скільки разів кожен елемент (ключ) зустрічається в вихідному масиві. Спираючись на ці дані можна одразу вирахувати на якому місці має стояти кожен елемент, а потім за один прохід поставити всі елементи на свої місця.

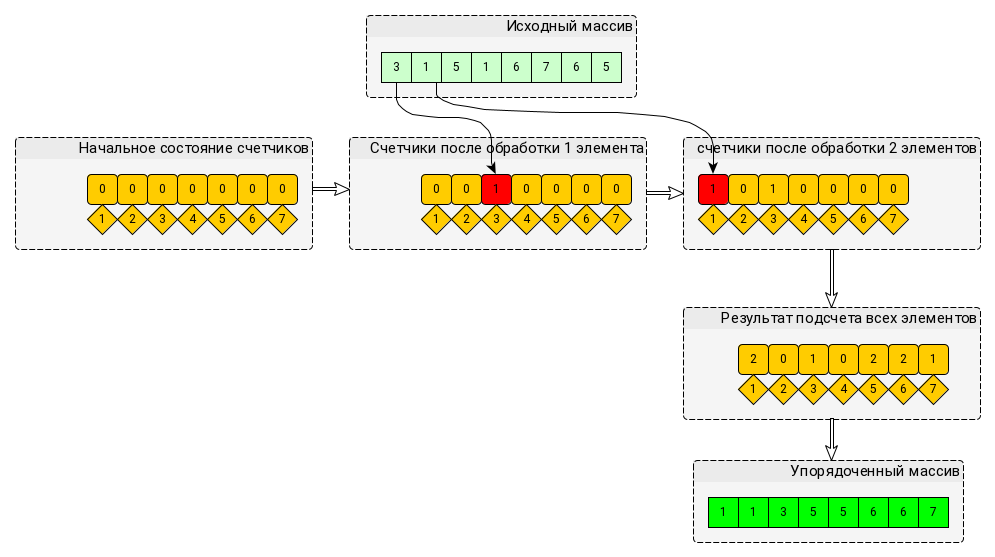


Рисунок 6.6 – алгоритм підрахунку

**Реалізований алгоритм:**

#include <iostream>

#include <time.h>

#include <chrono>

#include <random>

#include "windows.h"

#define GETTIME std::chrono::steady\_clock::now

#define CALCTIME std::chrono::duration\_cast<std::chrono::nanoseconds>

#define N 5000

using namespace std;

void CountingSort(short mass[], int n)

{

int max = INT\_MIN, min = INT\_MAX;

for (int i = 0; i < n; i++) {

if (mass[i] > max)

max = mass[i];

if (mass[i] < min)

min = mass[i];

}

int\* c = new int[max + 1 - min];

for (int i = 0; i < max + 1 - min; i++) {

c[i] = 0;

}

for (int i = 0; i < n; i++) {

c[mass[i] - min] = c[mass[i] - min] + 1;

}

int i = 0;

for (int j = min; j < max + 1; j++) {

while (c[j - min] != 0) {

mass[i] = j;

c[j - min]--;

i++;

}

}

}

int main()

{

srand(time(NULL));

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

auto begin = GETTIME();

auto end = GETTIME();

short mass[N];

for (int i = 0; i < N; i++)

{

mass[i] = -200 + rand() % (10 - (-200) + 1);

printf(" %d ", mass[i]);

}

printf("\n\n\n\n\n");

CountingSort(mass, N);

for (int i = 0; i < N; i++)

{

printf(" %d ", mass[i]);

}

auto elapsed\_ns = CALCTIME(end - begin);

printf("\nThe time: %lld ns\n", elapsed\_ns.count());

}

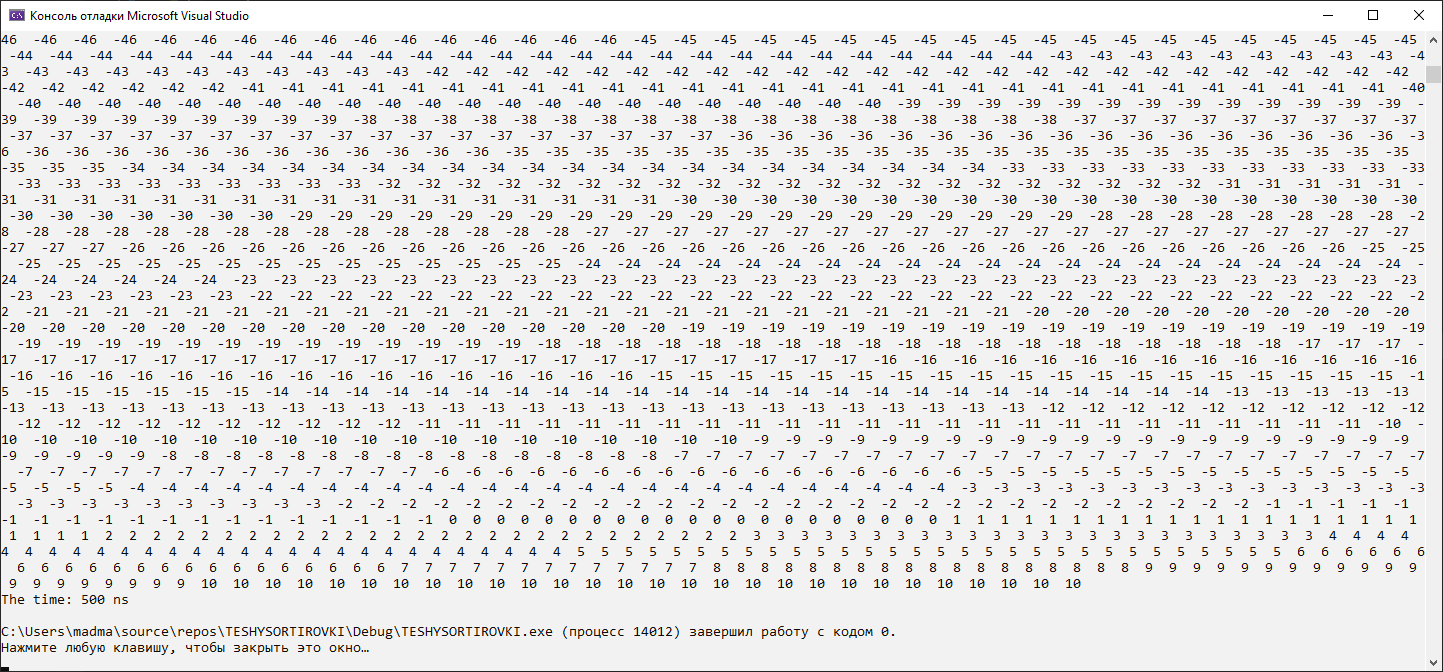


Рисунок 6.7 – Результат сортування

Рисунок 6.8 – Графік сортування

|  |  |
| --- | --- |
| Час | К-сть |
| 3184200 | 100 |
| 15753000 | 500 |
| 33022500 | 1000 |
| 213361500 | 5000 |
| 450564000 | 10000 |
| 920061100 | 20000 |
| 2264474600 | 50000 |
| Пірамідальне | |

|  |  |
| --- | --- |
| Час | К-сть |
| 3018800 | 100 |
| 15435700 | 500 |
| 35924700 | 1000 |
| 218565400 | 5000 |
| 462058600 | 10000 |
| 919604000 | 20000 |
| 2480017000 | 50000 |
| Шела | |

|  |  |
| --- | --- |
| Час | К-сть |
| 2972000 | 100 |
| 19438900 | 500 |
| 47720700 | 1000 |
| 2,99E+08 | 5000 |
| 6,1E+08 | 10000 |
| 1,19E+09 | 20000 |
| 2,94E+09 | 50000 |
| Підрахунком | |

Отже сортування підрахунком ефективне , при сортуванні великої к-сті елементів.

Метод Шела з формулою приросту Седжвіка ефективніший ніж звичайний метод.

**Лабораторна робота № 7-8**

Швидкі методи сортування

**Мета роботи:** реалізація швидких алгоритмів сортування та дослідження їх характеристик (швидкодія, необхідний обсяг пам'яті, застосування тощо).

**7.1 Хід роботи**

Завдання 1:

Маршруты движения автобусов с Киева;

1. Киев –(135) Житомир –(80) Новоград-Волынский –(100) Ровно –(68)

Луцк

2. Киев –(135) Житомир –(38) Бердичев –(73) Винница –(110)

Хмельницкий –(104) Тернополь

3. Киев –(135) Житомир –(115) Шепетовка

4. Киев –(78) Белая церковь –(115) Умань

5. Киев –(78) Белая церковь –(146) Черкассы –(105) Кременчуг

6. Киев –(78) Белая церковь –(181) Полтава – (130) Харьков

7. Киев –(128) Прилуки –(175) Сумы

8. Киев –(128) Прилуки –(109) Миргород

**Графи. Робота з графами.**

Графи - це абстрактний спосіб представлення типів відносин, наприклад доріг, що з'єднують міста, і інших видів мереж. Графи складаються з ребер і вершин. Вершина - це точка на графі, а ребро - це те, що з'єднує дві точки на графі.

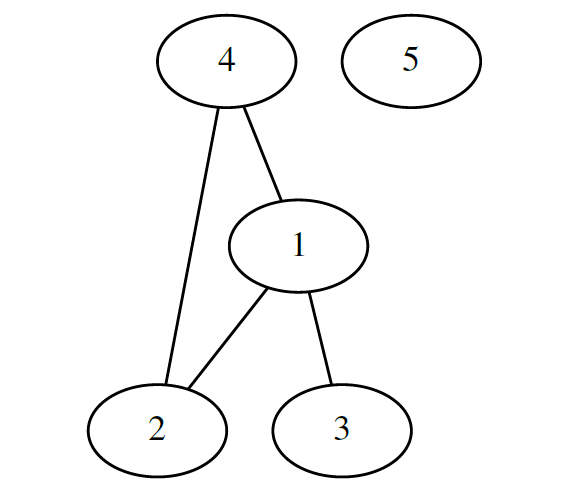


Рисунок 7.1 – Приклад графу

**Матриця суміжностей** - являє собою граф у вигляді двовимірної матриці з розмірами V x V, де V - кількість вершин графа. Матриці суміжності найкраще застосовувати, коли V² приблизно дорівнює E (числу ребер), тобто коли граф щільний.

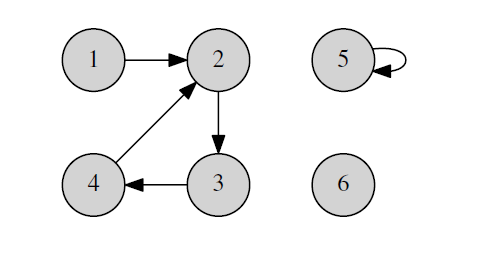


Рисунок 7.2 – Приклад графу який можна представити матрицею

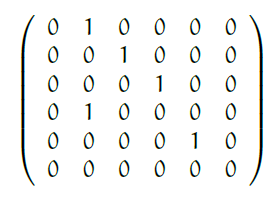


Рисунок 7.3 – Матриця суміжностей

Пошук в глибину(DFS):

Пошук в глибину - це один з базових алгоритмів на графах. Він застосовується для пошуку відстані від однієї вершини до інших вершин в графі. Це алгоритм обходу.

Пошук в глибину позначає кожну вершину в графі однієї з двох назв відвіданих або НЕ відвіданих. Алгоритм позначає кожну вершину як відвідану, якщо вдається уникнути циклів. Він працює таким чином:

1. Розміщуємо будь-яку з вершин графа в стек.
2. Беремо елемент зі стека і додаємо його в список відвіданих.
3. Створюємо список сусідів цієї вершини. Додаємо в стек ті, що не знаходяться в списку відвіданих.
4. Повторюємо 2 і 3 пункти, поки стік не спорожніє.

Пошук в ширину(BFS):

Пошук в ширину теж поміщає кожну вершину в графі в одну з двох категорій: відвіданих або невідвіданих. І мета у обох алгоритмів одна і та ж: позначати кожну вершину в графі як відвіданих, якщо вдається уникнути циклів. Ось як працює алгоритм пошуку в ширину:

1. Розміщуємо будь-яку вершину в графі в кінець черги.
2. Беремо елемент на початку черги і додаємо його в список відвіданих.
3. Створюємо список сусідів цієї вершини. Додаємо в кінець черги невідвіданих.
4. Повторюємо 2 і 3 пункти, поки черга не спорожніє.

Реалізовану матрицю надав в Exel документі репозиторію.

Реалізований алгоритм:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Task1

{

class Program

{

static void Main(string[] args) {

Console.OutputEncoding = Encoding.Unicode;

Console.InputEncoding = Encoding.Unicode;

for (int i = 0; i < n; i++)

{

Marked[i] = false;

}

Console.WriteLine("DFS:"); //вывод функции с алгоритмом DFS

DFS(0);

for (int i = 0; i < n; i++)

{

Marked[i] = false;

}

Console.WriteLine("\nBFS:"); //вывод функции с алгоритмом BFS

BFS(0);

}

public static string[] Routes = //масив с названиями городов

{

"Київ",

"Житомир",

"Біла церква",

"Прилуки",

"Новоград-Волинський",

"Бердичів",

"Шепетівка",

"Умань",

"Черкаси",

"Полтава",

"Суми",

"Миргород",

"Рівне",

"Вінниця",

"Кременчук",

"Харків",

"Луцьк",

"Хмельницький",

"Тернопіль",

};

public static byte[,] MatrizaSmezhnosti = // Создаем матрицу

{

{0, 135, 78, 128, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 80, 38, 115, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 115, 146, 181, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 175, 109, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 100, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 73, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 105, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 130, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 68, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 110, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 104 },

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

};

public const int n = 19; // кол-во вершин матрицы

public static bool[] Marked = new bool[n]; // масив используемых вершин

public static List<int> List = new List<int>(); // стек

public static Queue<int> Queue = new Queue<int>(); // очередь в стеке

public static List<int> Distance = new List<int>(); // растояние между городами

public static void DFS(int f, int sum = 0) //алгоритм DFS

{

if (!Marked[f])

{

List.Add(f);

Marked[f] = true;

for (int i = 0; i < n; i++) //проход всех вершин графа

{

if (MatrizaSmezhnosti[f, i] != 0)

{

sum += MatrizaSmezhnosti[f, i]; //добавление вершин в стек

DFS(i, sum);

foreach (int town in List)

{

Console.Write($"{Routes[town]} - ");

}

Console.Write($"{sum}\n");

sum -= MatrizaSmezhnosti[f, i]; //убираем со стека верхнии вершины

List.RemoveAt(List.Count - 1);

}

}

}

}

public static void BFS(int f) //алгоритм BFS

{

string start = Routes[f];

Marked[f] = true; // вершину пройдено

Queue.Enqueue(f);

Distance.Add(0);

while (Queue.Count != 0) //добавляем в очередь вершины

{

f = Queue.Dequeue();

for (int i = 0; i < n; i++) //цикл обхода всех вершин графа

{

if (MatrizaSmezhnosti[f, i] != 0 && !Marked[i])

{

Marked[i] = true; // вершину пройдено

Queue.Enqueue(i);

Distance.Add(Distance[f] + MatrizaSmezhnosti[f, i]);

Console.WriteLine(start + " -> " + Routes[i] + ": " + Distance[i]);

}

}

}

}

}

}

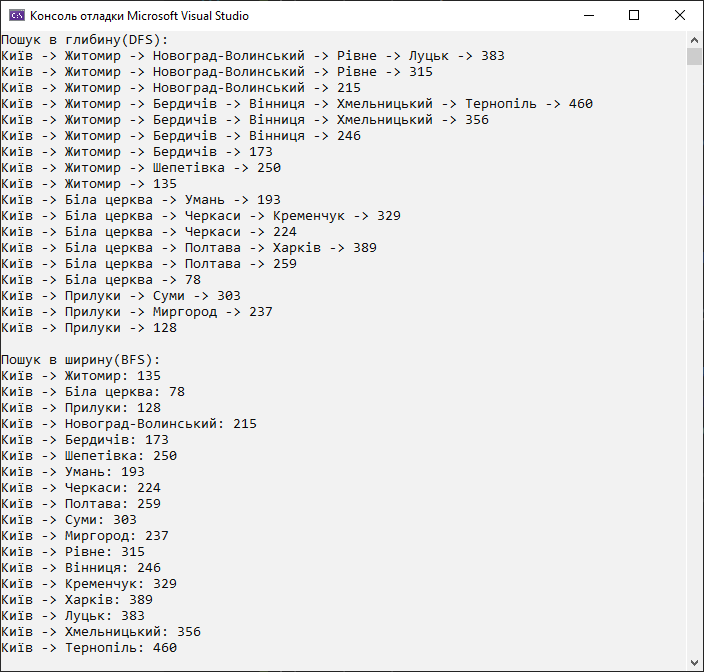


Рисунок 7.4 – Резльтат програми

Висновок: отже існує 2 алгоритми обходу матриці суміжностей , DFS BFS.

Графи дають великі можливості для реалізації найрізноманітніших задач.