МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Києво-Могилянська академія»
Факультет інформатики

Лабораторна робота № 4
з Процедурного програмування (на базі Сі/Сі++)
Функції обчислення n-го числа Фібоначчі.
Програмування, дослідження та порівняння

Студентки 2 курсу, групи 1,
спеціальності 121 «Інженерія програмного забезпечення»
Асташенкової Катерини Олександрівни

Викладач:
доц. В. В. Бублик

м. Київ
2025

Зміст

1. Порівняння реалізацій піднесення до степеня

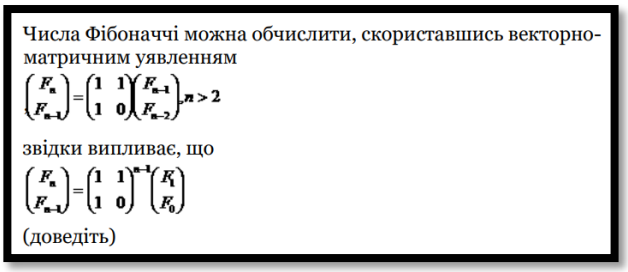
Відповідно до завдання, було реалізовано два варіанти (дві функції) алгоритму швидкого піднесення до степеня.

	power	***powerRecursive***
Підхід	«Знизу догори»: будує результат, обробляючи біти один за одним	«Згори донизу»: розбиває велику проблему на менші
Використання пам'яті	O(1) (константна): використовує лише фіксований набір змінних	O(log n) (через стек викликів): кожен рекурсивний виклик додає кадр у стек
Продуктивність	Зазвичай швидша, оскільки цикл ефективніший за рекурсивні виклики	Трохи повільніша через накладні витрати на виклик функцій
Читабельність	Може бути складнішою для розуміння	Більш інтуїтивна та ближча до математичного формулювання

- *Функція fibonacciIterativeQuick* у програмі використовує *power*
- *Функція fibonacciRecQuick* у програмі використовує *powerRecursive*

Важливе уточнення перед початком роботи: *використовуємо ряд чисел Фібоначчі, починаючи з нуля: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...*

2. Доведення векторно-матричного уявлення



Нехай $M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ та $V_n = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$.

Потрібно довести, що $V_n = M^{(n-1)} \times V_1$, $\forall\ n \geq 1$, де $V_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

($n \geq 1$, оскільки якщо $n = 1$, то F_0 визначено)

Доведемо методом математичної індукції: шаблон умови ✓.

База індукції: n=1;

ліва частина (ЛЧ): $V_1 = \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$;

права частина (ПЧ): $M^0 \times V_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$;

ЛЧ = ПЧ ✓.

Індукційне припущення: припустимо, що для n=k $V_n = M^{(n-1)} \times V_1$, $\forall\ n \geq 1$, тобто $V_k = M^{(k-1)} \times V_1$, $\forall\ k \geq 1$.

Індукційний перехід: доведемо, що n=k+1 $V_n = M^{(n-1)} \times V_1$, $\forall\ n \geq 1$, тобто

$\boxed{V_{k+1} = M^k \times V_1,\ \forall\ k \geq 0.}$

Доведення: $V_{k+1} = \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}$,

за рекурсивним визначенням чисел Фібоначчі $F_{k+1} = F_k + F_{k-1}$, тоді

$$V_{k+1} = \begin{pmatrix} F_k + F_{k-1} \\ F_k \end{pmatrix} = \begin{pmatrix} 1 \times F_k + 1 \times F_{k-1} \\ 1 \times F_k + 0 \times F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_k \\ F_{k-1} \end{pmatrix} = M \times V_k =$$

⤵ *(індукційне припущення)*

$$= M \times M^{k-1} \times V_1 = M^k \times V_1 .$$

Отже, $\boxed{V_{k+1} = M^k \times V_1}$, що і є твердженням індукційного переходу.

Доведено.

3. Опис підходів для обчислення n-го числа Фібоначчі

3.1. Ітеративний метод (класичний)

fibonacciIterative

- **Складність**: O(n) за часом, O(1) за пам'яттю.
- **Принцип**: послідовне обчислення F(1), F(2), … , F(n), що зберігає два попередні значення.
- **Переваги**: простота, мінімальне використання пам'яті.
- **Недоліки**: повільний для дуже великих n.

3.2. Ітеративний швидкий (матричний) метод

fibonacciIterativeQuick

- **Складність**: O(log n) за часом, O(1) за пам'яттю.
- **Принцип**: обчислення M^{n-1} за допомогою ітеративної функції швидкого піднесення до степеня *power*.
- **Переваги**: дуже швидко, не використовує стек рекурсії.
- **Недоліки**: складніша реалізація порівняно з класичним.

3.3. Рекурсивний метод (класичний)

fibonacciRec

- **Складність**: $O(2^n)$ за часом, O(n) за пам'яттю (глибина рекурсії).
- **Принцип**: пряме застосування формули $F_n = F_{n-1} + F_{n-2}$.
- **Переваги**: найпростіший код, відповідає математичному визначенню.
- **Недоліки**: експоненційна складність, величезна кількість повторних обчислень, практично не придатний приблизно з $n > 46$ через обмеження стеку (в разі передачі до функції більших значень виникає проблема *Stack Overflow*).

3.4. Рекурсивний швидкий (матричний) метод

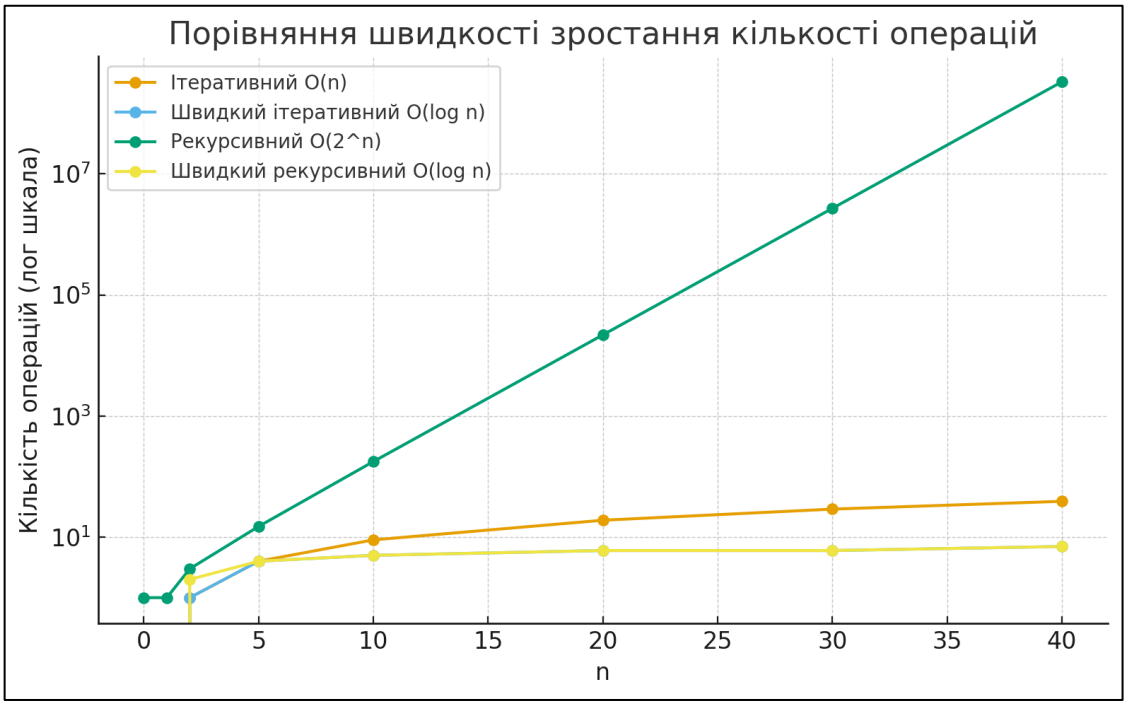
fibonacciRecQuick

- **Складність**: O(log n) за часом, O(log n) за пам'яттю (глибина рекурсії).
- **Принцип**: обчислення M^{n-1} за допомогою рекурсивної функції швидкого піднесення до степеня *powerRecursive*.
- **Переваги**: дуже швидко, демонструє рекурсивний підхід до алгоритму O(log n).
- **Недоліки**: використовує стек, що може бути більш небезпечно за ітеративний варіант.

4. Порівняння швидкості зростання різних показників для різних методів пошуку n-го числа Фібоначчі

4.1. Кількість операцій

n	Ітеративний O(n)	Швидкий ітеративний O(log n)	Звичайна рекурсія $O(2^n)$	Швидка рекурсія O(log n)
0	0	0	1	0
1	0	0	1	0
2	1	1	3	2
5	4	4	15	4
10	9	5	177	5
20	19	6	21891	6
30	29	6	2692537	6
40	39	7	331160281	7

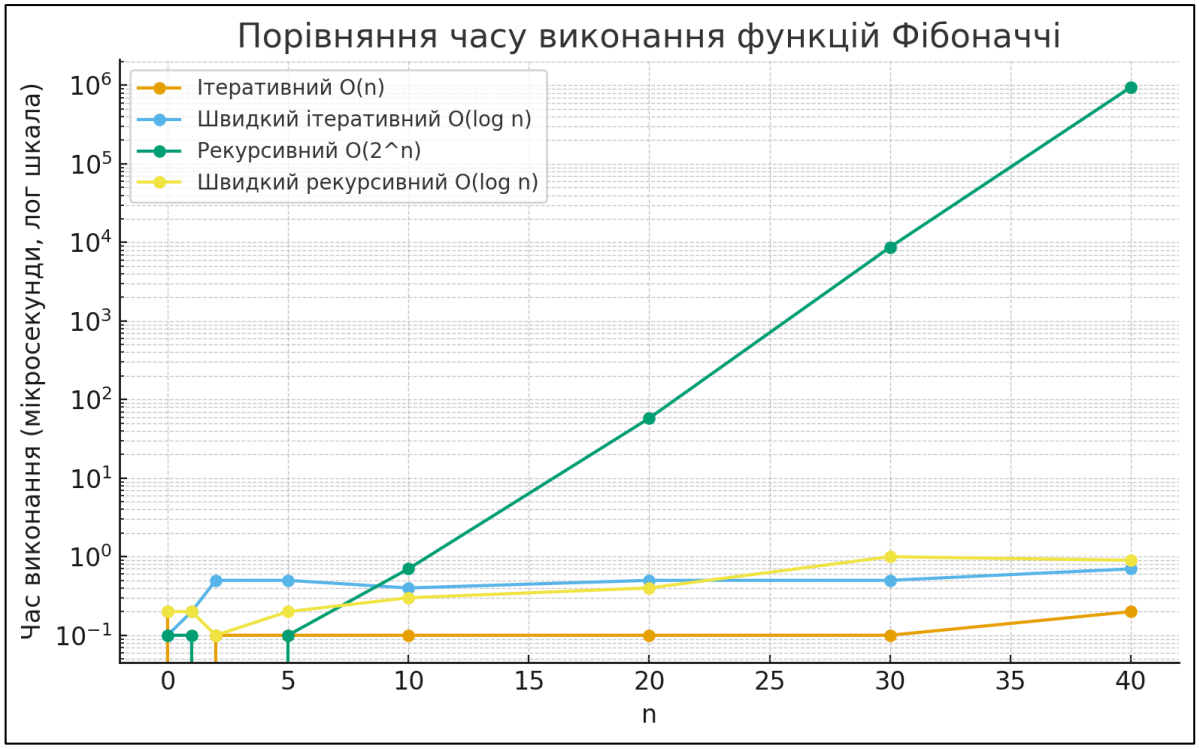


Швидкий ітеративний і швидкий рекурсивний підходи виконують однакову кількість математичних операцій на більших числах, тому що вони роблять однакову кількість множень матриць за тією самою формулою швидкого піднесення, яку ми доводили раніше, а лічильник у коді рахує саме ці множення.

Різниця між цими функціями полягає в організації коду, споживанні пам'яті та часі виконання. Ітеративний варіант ефективніший на практиці, бо не має накладних витрат на рекурсію, хоча за кількістю арифметичних операцій підходи ідентичні, як було зауважено раніше.

4.2. Час виконання (приблизно, на основі конкретного запуску програми)

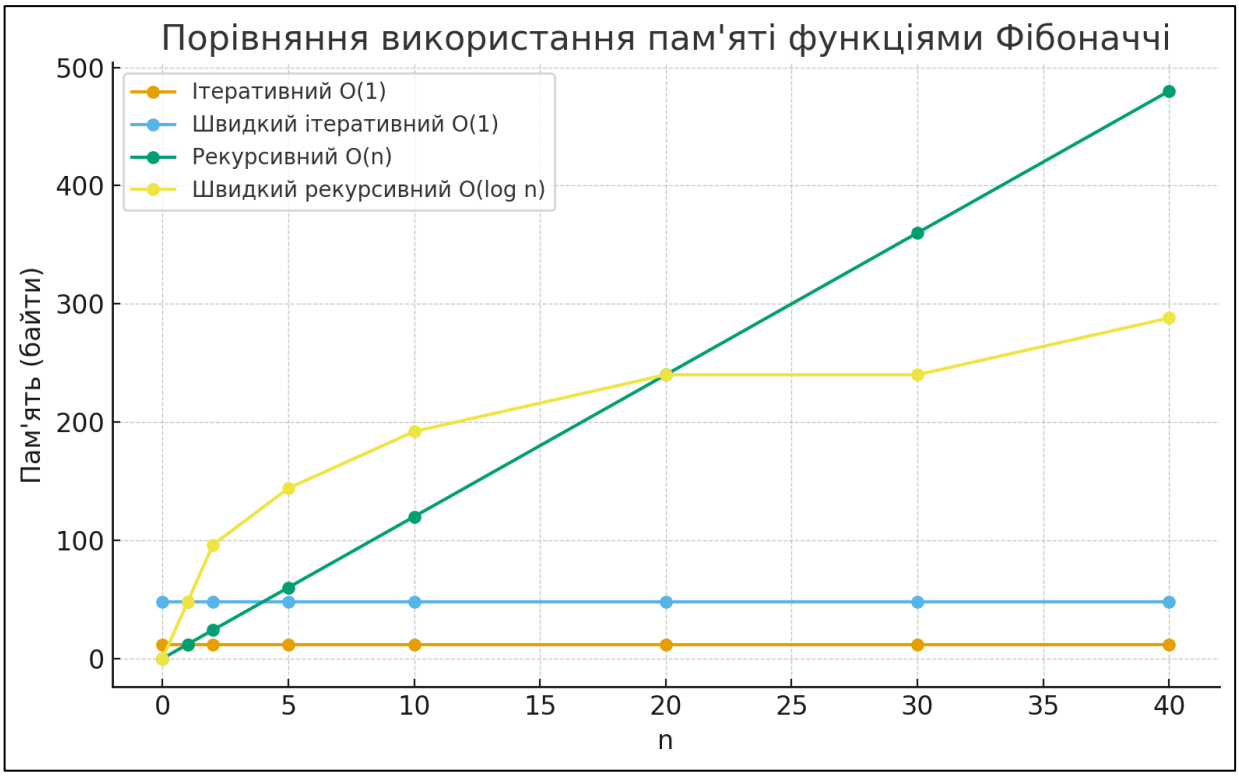
n	Ітеративний, µs	Швидкий ітеративний, µs	Звичайна рекурсія, µs	Швидка рекурсія, µs
0	0,1	0,3	0,1	0,1
1	0,1	0,2	0	0
2	0,2	0,4	0,1	0,2
5	0,1	0,5	0,2	0,4
10	0,1	0,5	0,8	0,4
20	0,2	0,5	57,7	0,4
30	0,2	0,5	7594,8	1,1
40	0,2	0,7	953603	0,9



Спостерігаємо «вибухову» дію звичайного рекурсивного підходу.

4.3. Використання пам’яті (теоретично й приблизно)

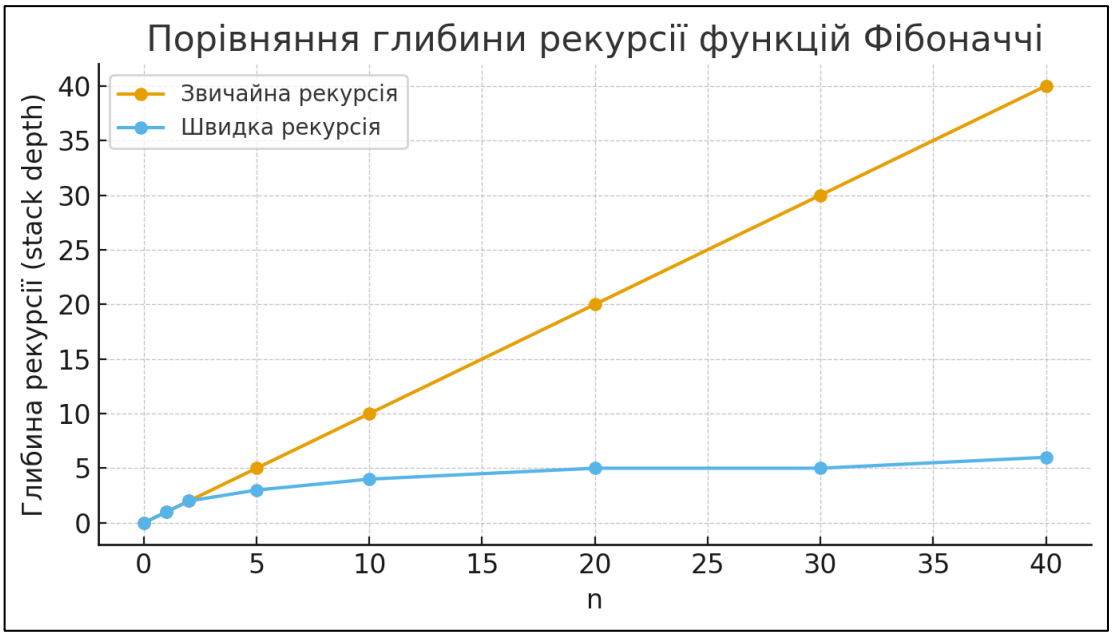
n	Ітеративний, байт	Швидкий ітеративний, байт	Звичайна рекурсія, байт	Швидка рекурсія, байт
0	12	48	0	0
1	12	48	12	48
2	12	48	24	96
5	12	48	60	144
10	12	48	120	192
20	12	48	240	240
30	12	48	360	240
40	12	48	480	288



Ітеративні функції використовують сталу кількість байтів пам’яті, а рекурсії збільшують використання пам’яті внаслідок постійного виклику.

4.4. Глибина рекурсії у стеку (теоретично й приблизно)

n	**Звичайна рекурсія**, кількість рівнів виклику (кадрів)	**Швидка рекурсія,** кількість рівнів виклику (кадрів)
0	0	0
1	1	1
2	2	2
5	5	3
10	10	4
20	20	5
30	30	5
40	40	6



Звичайна рекурсія суттєво швидше і більше поглиблює звернення до стеку під час роботи, тому може швидше привести програму до «точки» Stack Overflow.

5. Висновки

5.1. Математичний аспект

Доведено, що числа Фібоначчі можна обчислювати через піднесення матриці до степеня. Це дозволяє застосувати алгоритм швидкого зведення у степінь (*binary exponentiation*) і знизити часову складність з O(n) до O(log n). Також важливо, що *O(log(n))*-алгоритми використовують набагато менше пам'яті стеку, ніж *O(n)*-алгоритми.

5.2. Алгоритмічний і практичний аспекти

Реалізовано та протестовано чотири методи обчислення чисел Фібоначчі, що демонструють різні асимптотичні складності:

1) **простий ітеративний** — оптимальний для малих n;
2) **швидкий ітеративний (матричний)** — найкращий вибір для будь-яких n завдяки швидкості та O(1) складності за пам'яттю;
3) **простий рекурсивний** — не ефективний, краще уникати;
4) **швидкий рекурсивний (матричний)** — демонструє рекурсивний підхід швидкого методу, використовуючи стек.

5.3. Обмеження

- За умовою лабораторної роботи досліджували функції саме на типі *int.*

```
struct Matrix2x2
{
  int _11, _12, _21, _22;
};
struct Vector2
{
  int _1, _2;
};
```

- Максимальне число Фібоначчі, яке може вміститися у тип *int* (32 біти), — це F_{46} = 1 836 311 903, оскільки діапазон *signed int* — від -2 147 483 648 до 2 147 483 647, а наступний елемент ряду Фібоначчі F_{47} = 2 971 215 073, що вже більше за діапазон можливих значень типу *int.*
- Для обчислення F_n в разі $n > 46$ потрібне використання типів більшої місткості (*long long, unsigned long long — 64 біти, для цих типів максимум* F_{92} *та* F_{93} *відповідно*), а для зовсім великих чисел — бібліотек довільної точності, наприклад *BigInteger*. Проте загалом під час виконання цієї лабораторної роботи навіть на типі *int* нам удалося досить детально дослідити відмінності в ефективності різних функцій обчислення n-го числа Фібоначчі.

Додаток 1. Програмний код

```
#include <iostream>
#include "fibonacci.h"
using namespace std;

// Fibonacci range: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...

int main()
{
    test();
    cout << "\n\nProgram completed." << endl;
    return 0;
}
```

```
#pragma once

struct Matrix2x2 {
    int _11, _12, _21, _22;
};

struct Vector2 {
    int _1, _2;
};

int fibonacciIterative(int n, int& count);
int fibonacciIterativeQuick(int n, int& count);
int fibonacciRec(int n, int& count);
int fibonacciRecQuick(int n, int& count);

void test();
```

```
#include <iostream>
#include <cassert>
#include <iomanip>
#include <chrono> // in order to check time
#include <windows.h>
#include <psapi.h> // in order to check memory usage
#include "fibonacci.h"

using namespace std;
using namespace std::chrono;

static size_t getCurrentMemoryUsage() {
	PROCESS_MEMORY_COUNTERS info;
	if (GetProcessMemoryInfo(GetCurrentProcess(), &info, sizeof(info))) {
		return info.WorkingSetSize;
	}
	return 0;
}

// Matrix-vector multiplication
static Vector2 multiply(const Matrix2x2& M, const Vector2& v, int& count) {
	count++; // counting the multiplying operations
	Vector2 result{};
	result._1 = M._11 * v._1 + M._12 * v._2;
	result._2 = M._21 * v._1 + M._22 * v._2;
	return result;
}

// Matrix-matrix multiplication
static Matrix2x2 multiply(const Matrix2x2& A, const Matrix2x2& B) {
```

```
        Matrix2x2 result{};
        result._11 = A._11 * B._11 + A._12 * B._21;
        result._12 = A._11 * B._12 + A._12 * B._22;
        result._21 = A._21 * B._11 + A._22 * B._21;
        result._22 = A._21 * B._12 + A._22 * B._22;
        return result;
}

// Iterative matrix power
static Matrix2x2 power(const Matrix2x2& M, int n, int& count) {
        if (n == 0) return { 1, 0, 0, 1 };
        if (n == 1) return M;
        Matrix2x2 result = { 1, 0, 0, 1 };
        Matrix2x2 base = M;
        while (n > 0) {
                count++;
                if (n % 2 == 1) result = multiply(result, base);
                base = multiply(base, base);
                n /= 2;
        }
        return result;
}

// Recursive matrix power
static Matrix2x2 powerRec(const Matrix2x2& M, int n, int& count) {
        count++;
        if (n == 0) return { 1, 0, 0, 1 };
        if (n == 1) return M;
        Matrix2x2 half = powerRec(M, n / 2, count);
        Matrix2x2 halfSquared = multiply(half, half);
        if (n % 2 == 0) {
```

```
		return halfSquared;
	}
	else {
		return multiply(M, halfSquared);
	}
}

int fibonacciIterative(int n, int& count) {
	assert(n >= 0 && "n must be non-negative");
	if (n <= 1) return n;
	int prev = 0, current = 1;
	for (int i = 2; i <= n; i++) {
		count++;
		int next = prev + current;
		prev = current;
		current = next;
	}
	return current;
}

int fibonacciIterativeQuick(int n, int& count) {
	assert(n >= 0 && "n must be non-negative");
	if (n <= 1) return n; // optimization

	Matrix2x2 M = { 1, 1, 1, 0 };
	Vector2 v = { 1, 0 };
	Matrix2x2 Mn = power(M, n - 1, count);
	Vector2 result = multiply(Mn, v, count);
	return result._1;
}
```

```
int fibonacciRec(int n, int& count) {
	count++;
	assert(n >= 0 && "n must be non-negative");
	if (n <= 1) return n;
	return fibonacciRec(n - 1, count) + fibonacciRec(n - 2, count);
}

int fibonacciRecQuick(int n, int& count) {
	assert(n >= 0 && "n must be non-negative");
	if (n <= 1) return n; // optimization

	Matrix2x2 M = { 1, 1, 1, 0 };
	Vector2 v = { 1, 0 };
	Matrix2x2 Mn = powerRec(M, n - 1, count);
	Vector2 result = multiply(Mn, v, count);
	return result._1;
}

void test() {
	Matrix2x2 M = { 1, 1, 1, 0 };

	// checking iterative and recursive power implementations
	for (int n = 0; n <= 10; n++) {
		int counter1 = 0, counter2 = 0;
		Matrix2x2 iterative = power(M, n, counter1);
		Matrix2x2 recursive = powerRec(M, n, counter2);
		assert(iterative._11 == recursive._11);
		assert(iterative._12 == recursive._12);
		assert(iterative._21 == recursive._21);
		assert(iterative._22 == recursive._22);
	}
```

```
	int testCases[] = { 0, 1, 2, 5, 10, 20, 30, 40 };
	int expected[] = { 0, 1, 1, 5, 55, 6765, 832040, 102334155 };
	int numTests = sizeof(testCases) / sizeof(testCases[0]);

	// structure for storing results
	struct TestResult {
		int n;
		int exp;
		int operations[4];
		double times[4];
		int memory[4];
		int stackDepth[4];
		int results[4];
	};
	TestResult allResults[8]{};

	for (int i = 0; i < numTests; i++) {
		if (i != 0) cout << "\n_____" <<
endl;
		int n = testCases[i];
		int exp = expected[i];
		allResults[i].n = n;
		allResults[i].exp = exp;

		cout << "n = " << n << " (Expected Fibonacci number: " << exp << ")\n" << endl;

		int c1 = 0;
		auto start1 = high_resolution_clock::now();
		int r1 = fibonacciIterative(n, c1);
		auto end1 = high_resolution_clock::now();
```

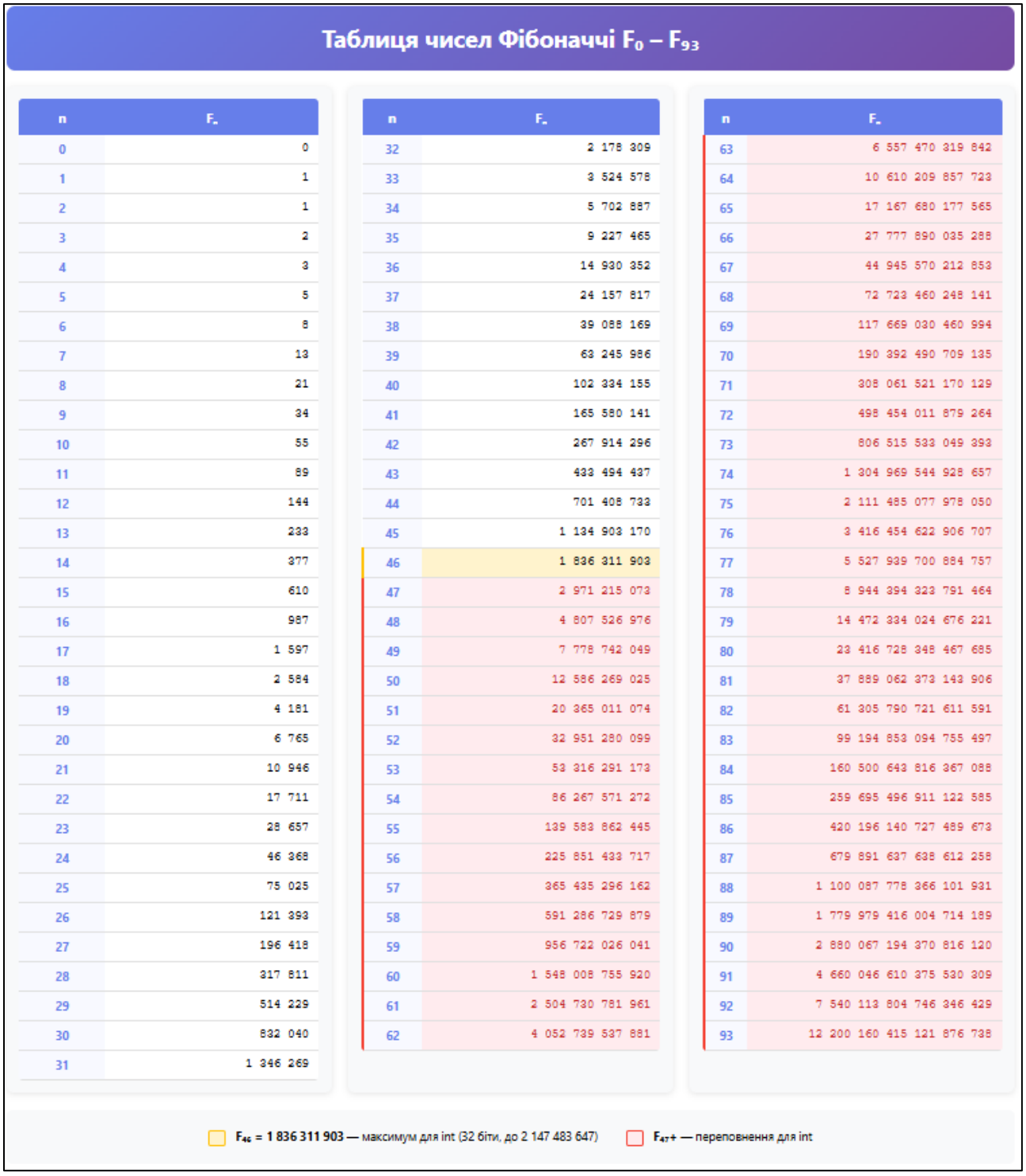
```
double time1 = duration<double, micro>(end1 - start1).count();

int c2 = 0;
auto start2 = high_resolution_clock::now();
int r2 = fibonacciIterativeQuick(n, c2);
auto end2 = high_resolution_clock::now();
double time2 = duration<double, micro>(end2 - start2).count();

int c3 = 0;
auto start3 = high_resolution_clock::now();
int r3 = fibonacciRec(n, c3);
auto end3 = high_resolution_clock::now();
double time3 = duration<double, micro>(end3 - start3).count();

int c4 = 0;
auto start4 = high_resolution_clock::now();
int r4 = fibonacciRecQuick(n, c4);
auto end4 = high_resolution_clock::now();
double time4 = duration<double, micro>(end4 - start4).count();

// storing results
allResults[i].operations[0] = c1;
allResults[i].operations[1] = c2;
allResults[i].operations[2] = c3;
allResults[i].operations[3] = c4;

allResults[i].times[0] = time1;
allResults[i].times[1] = time2;
allResults[i].times[2] = time3;
allResults[i].times[3] = time4;
```

```
allResults[i].results[0] = r1;
allResults[i].results[1] = r2;
allResults[i].results[2] = r3;
allResults[i].results[3] = r4;

cout << "Results" << endl;
cout << "  Iterative:       " << r1 << endl;
cout << "  Quick iterative: " << r2 << endl;
cout << "  Recursive:       " << r3 << endl;
cout << "  Quick recursive: " << r4 << endl;

// all results must match expected Fibonacci numbers
assert(r1 == exp);
assert(r2 == exp);
assert(r3 == exp);
assert(r4 == exp);

cout << "\nTime Complexity" << endl;
cout << "  Iterative (O(n)):           " << c1 << " operations, "
          << time1 << " microseconds" << endl;
cout << "  Quick iterative (O(log n)):  " << c2 << " operations, "
          << time2 << " microseconds" << endl;
cout << "  Recursive (O(2^n)):         " << c3 << " operations, "
          << time3 << " microseconds" << endl;
cout << "  Quick recursive (O(log n)):  " << c4 << " operations, "
          << time4 << " microseconds" << endl;

// theoretical memory usage estimation
int stackDepthRec = n;
int stackDepthRecQuick = (int)ceil(log2(n + 1)); // "divide and conquer"
```

```
            int memIterative = 3 * sizeof(int); // prev, current, next
            int memIterativeQuick = sizeof(Matrix2x2) * 2 + sizeof(Vector2) * 2; // result, base;
v, result
            int memRecursive = stackDepthRec * (sizeof(int) * 3); // n, count, return
            int memRecQuick = stackDepthRecQuick * (sizeof(Matrix2x2) * 3); // M, half,
halfSquared

            allResults[i].memory[0] = memIterative;
            allResults[i].memory[1] = memIterativeQuick;
            allResults[i].memory[2] = memRecursive;
            allResults[i].memory[3] = memRecQuick;

            allResults[i].stackDepth[0] = 1;
            allResults[i].stackDepth[1] = 1;
            allResults[i].stackDepth[2] = stackDepthRec;
            allResults[i].stackDepth[3] = stackDepthRecQuick;

            cout << "\nSpace Complexity (theoretical)" << endl;
            cout << "  Iterative (O(1)):           " << memIterative << " bytes" << endl;
            cout << "  Quick iterative (O(1)):     " << memIterativeQuick << " bytes" << endl;
            cout << "  Recursive (O(n)):           " << memRecursive
                 << " bytes (stack depth: " << stackDepthRec << ")" << endl;
            cout << "  Quick recursive (O(log n)): " << memRecQuick
                 << " bytes (stack depth: " << stackDepthRecQuick << ")" << endl;
        }

        // ASSERT-BASED COMPARISON
        // (if quick functions perform worse than expected, this will trigger an assertion)
        double tolerance = 2.0;

        for (int i = 0; i < numTests; i++) {
```

```
            int n = allResults[i].n;

            // The quick iterative implementation should not require
            // significantly more operations than the simple iterative one
            if (n > 10) { // checking only for larger n where differences are more noticeable
                assert(allResults[i].operations[1] <= allResults[i].operations[0] * tolerance);
            }

            // The quick recursive implementation should not require
            // significantly more operations than the simple recursive one
            if (n > 10) { // checking only for larger n where differences are more noticeable
                assert(allResults[i].operations[3] <= allResults[i].operations[2] * tolerance);
            }
        }
}
```

Додаток 2. Таблиця чисел Фібоначчі до 93



Таблиця чисел Фібоначчі $F_0 – F_{93}$

n	F_n
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89
12	144
13	233
14	377
15	610
16	987
17	1 597
18	2 584
19	4 181
20	6 765
21	10 946
22	17 711
23	28 657
24	46 368
25	75 025
26	121 393
27	196 418
28	317 811
29	514 229
30	832 040
31	1 346 269
32	2 178 309
33	3 524 578
34	5 702 887
35	9 227 465
36	14 930 352
37	24 157 817
38	39 088 169
39	63 245 986
40	102 334 155
41	165 580 141
42	267 914 296
43	433 494 437
44	701 408 733
45	1 134 903 170
46	1 836 311 903
47	2 971 215 073
48	4 807 526 976
49	7 778 742 049
50	12 586 269 025
51	20 365 011 074
52	32 951 280 099
53	53 316 291 173
54	86 267 571 272
55	139 583 862 445
56	225 851 433 717
57	365 435 296 162
58	591 286 729 879
59	956 722 026 041
60	1 548 008 755 920
61	2 504 730 781 961
62	4 052 739 537 881
63	6 557 470 319 842
64	10 610 209 857 723
65	17 167 680 177 565
66	27 777 890 035 288
67	44 945 570 212 853
68	72 723 460 248 141
69	117 669 030 460 994
70	190 392 490 709 135
71	308 061 521 170 129
72	498 454 011 879 264
73	806 515 533 049 393
74	1 304 969 544 928 657
75	2 111 485 077 978 050
76	3 416 454 622 906 707
77	5 527 939 700 884 757
78	8 944 394 323 791 464
79	14 472 334 024 676 221
80	23 416 728 348 467 685
81	37 889 062 373 143 906
82	61 305 790 721 611 591
83	99 194 853 094 755 497
84	160 500 643 816 367 088
85	259 695 496 911 122 585
86	420 196 140 727 489 673
87	679 891 637 638 612 258
88	1 100 087 778 366 101 931
89	1 779 979 416 004 714 189
90	2 880 067 194 370 816 120
91	4 660 046 610 375 530 309
92	7 540 113 804 746 346 429
93	12 200 160 415 121 876 738

F_{46} = **1 836 311 903** — максимум для int (32 біти, до 2 147 483 647) F_{47+} — переповнення для int

Джерело побудови таблиці: https://claude.ai/public/artifacts/c891d233-a3c3-4c99-b9be-0fea7f48d925