

4. Оцінка складності алгоритмів

При розробці програми дуже важливо провести **аналіз алгоритмів**. Аналіз полягає в тому, щоб передбачити необхідні для його виконання ресурси: **час роботи, об'єм пам'яті, пропускна здатність мережі** тощо. Однак частіше за все визначаються перші два параметри. Часто вирішити одну і ту ж проблему можна за допомогою декількох алгоритмів і потрібно вибрати **найбільш ефективний** з них.

Час роботи будь-якого алгоритму залежить від набору вхідних значень, наприклад, для сортування ста чисел потрібно більше часу, ніж для сортування десяти чисел. Таким чином, в загальному випадку час роботи алгоритму збільшується зі збільшенням **розміру вхідних даних**, тому загальноприйнята практика – представляти час роботи програми як функцію, залежну від кількості вхідних елементів.

Приклад 1

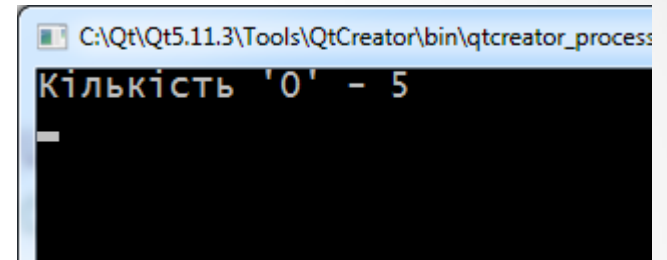
```
#include <stdio.h>
#include <windows.h>
#include <stdint.h>

// Знаходження кількості нулів в масиві
int main()
{
    SetConsoleCP(CP_UTF8);
    SetConsoleOutputCP(CP_UTF8);

    uint32_t arr[] = {0, 1, 0, 5, 7, 2, 5, 0, 7, 7, 1, 0, 3, 4, 6, 0, 3};
    uint8_t len = sizeof(arr)/sizeof(arr[0]);
    uint8_t count = 0;

    for(uint8_t i=0; i < len; i++) {
        if(arr[i] == 0) count++;
    }
    printf("Кількість '0' - %u\n", count);

    return 0;
}
```



Приклад 2

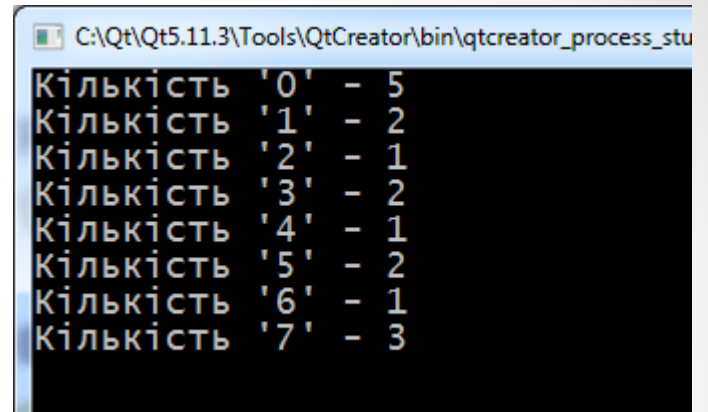
```
#include <stdio.h>
#include <windows.h>
#include <stdint.h>

// Знаходження кількості кожного значення у масиві
//1й варіант
int main() {
    SetConsoleCP(CP_UTF8);
    SetConsoleOutputCP(CP_UTF8);

    uint32_t arr[] = {0, 1, 0, 5, 7, 2, 5, 0, 7, 7, 1, 0, 3, 4, 6, 0, 3};
    uint8_t len = sizeof(arr)/sizeof(arr[0]);

    const uint8_t digit = 8;
    uint8_t count[digit]{};

    for(uint8_t j=0; j < digit; j++) {
        for(uint8_t i=0; i < len; i++) {
            if(arr[i] == j) count[j]++;
        }
        printf("Кількість '%u' - %u\n", j, count[j]);
    }
    return 0;
}
```

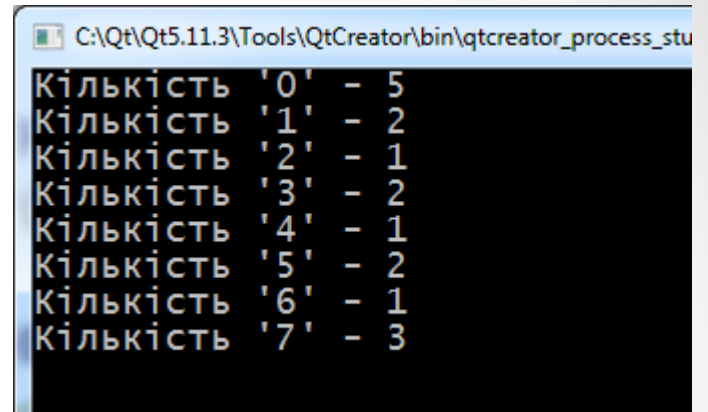


```
C:\Qt\Qt5.11.3\Tools\QtCreator\bin\qtcreator_process_stu
Кількість '0' - 5
Кількість '1' - 2
Кількість '2' - 1
Кількість '3' - 2
Кількість '4' - 1
Кількість '5' - 2
Кількість '6' - 1
Кількість '7' - 3
```

Приклад 2

```
// Знаходження кількості кожного значення у масиві  
//2й варіант
```

```
int main()  
{  
    SetConsoleCP(CP_UTF8);  
    SetConsoleOutputCP(CP_UTF8);  
  
    uint32_t arr[] = {0, 1, 0, 5, 7, 2, 5, 0, 7, 7, 1, 0, 3, 4, 6, 0, 3};  
    uint8_t len = sizeof(arr)/sizeof(arr[0]);  
  
    const uint8_t digit = 8;  
    uint8_t count[digit]{};  
  
    for(uint8_t i=0; i < len; i++) {  
        count[arr[i]]++;  
    }  
  
    for(uint8_t j=0; j < digit; j++) {  
        printf("Кількість '%u' - %u\n", j, count[j]);  
    }  
    return 0;  
}
```



```
C:\Qt\Qt5.11.3\Tools\QtCreator\bin\qtcreator_process_stu  
Кількість '0' - 5  
Кількість '1' - 2  
Кількість '2' - 1  
Кількість '3' - 2  
Кількість '4' - 1  
Кількість '5' - 2  
Кількість '6' - 1  
Кількість '7' - 3
```

Поняття **розмір вхідних даних** залежить від задачі, що розглядається. У багатьох задачах, таких як сортування – це розмір сортованого масиву, а для перемножування двох цілих чисел – це загальна кількість розрядів, яка необхідна для представлення вхідних чисел тощо.

Час роботи алгоритму для тих чи інших вхідних даних вимірюється в кількості елементарних операцій ("кроків", які необхідно виконати). Елементарні операції, в загальному випадку, є машинно-залежними. Однак будемо виходити з точки зору, згідно з якою час виконання різних рядків алгоритму може відрізнятися, але один той самий рядок виконується за фіксований час.

Наприклад:

```
for(int i=0; i< n; ++i) { //t1
    // певні дії          //t2
}
```

$$T(n) = t1*(n+1) + t2*n$$

Слід також звернути увагу на такий параметр як **швидкість росту (порядок росту)** часу роботи алгоритму, тому що цей параметр показує наскільки сильно збільшується час роботи алгоритму при збільшенні кількості вхідних даних. **Швидкість зростання** складності алгоритму визначається старшим, домінуючим членом формули обчислення часу виконання.

Таким чином для аналізу ефективності алгоритмів використовується наступні критерії:

- **часова ефективність** – час, який витрачається на алгоритм для вирішення поставленого завдання;
- **просторова ефективність** – додатковий обсяг пам'яті даних, який необхідний алгоритму при вирішенні поставленого завдання.

При аналізі часової ефективності виникає ряд проблем:

- система команд процесора і її час виконання різний, тому що система команд процесора є машинно-залежною;
- оптимізація різних компіляторів і інтерпретаторів різна;
- складність отримання строгої математичної залежності кількості ітерацій алгоритму від розміру вхідних даних.

При аналізі просторової ефективності виникає також ряд проблем:

- архітектура ЕОМ різна;
- оптимізація різних компіляторів і інтерпретаторів різна;
- типи вхідних даних можуть бути різними.

Для вирішення проблеми аналізу часової ефективності застосовують наступні підходи (спрощення):

- час роботи алгоритму розглядається як $T(n) = C_t \cdot t(n)$, де C_t – константа, яка є машинно-залежною величиною і відповідає часу виконання елементарної операції; $t(n)$ – функціональна залежність між розміром вхідних даних та кількістю елементарних операцій;
- для дослідження функціональної залежності $t(n)$ – використовують оцінку асимптотичної складності алгоритму (асимптотичний порядок росту алгоритму).

Що ж таке асимптотична складність алгоритму? Розглядаючи вхідні дані досить великих розмірів для оцінки швидкості росту часу роботи алгоритму, ми тим самим вивчаємо асимптотичну ефективність алгоритмів. Це означає, що нас цікавить тільки те, як час роботи алгоритму зростає зі збільшенням розміру вхідних даних до нескінченності. Зазвичай алгоритм, ефективний в асимптотичному сенсі, буде ефективним для всіх вхідних даних, за винятком дуже маленьких.

Для того щоб можна було порівнювати між собою швидкості зростання і класифікувати їх, були введені умовні позначення:

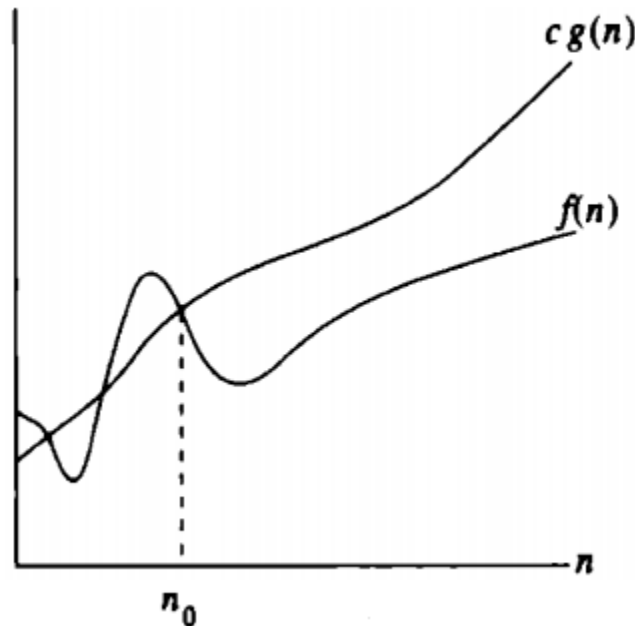
$O(n)$ (вимовляється як “О велике”);

$\Omega(n)$ (“Омега велике”);

$\Theta(n)$ (“Тета велике”).

Асимптотична верхня межа. O - нотація

Нехай $t(n)$ – це функція (час виконання в гіршому випадку для деякого алгоритму) з вхідними даними розміру n . Функція $t(n)$ має порядок $O(g(n))$ (належить до класу функцій $g(n)$), якщо існують такі константи $c > 0$ і $n_0 \geq 0$, що для всіх $n \geq n_0$ виконується умова $t(n) \leq c \cdot g(n)$. У такому випадку говорять, що $t(n)$ має **асимптотичну верхню межу** $g(n)$. Важливо підкреслити, що це визначення вимагає існування константи c , що працює для всіх n !



Розглянемо приклад для визначення виразу верхньої межі складності алгоритму. Припустимо, є алгоритм, час виконання якого задається у вигляді: $t(n) = an^2 + bn + c$, де всі константи позитивні.

Слід зауважити, що для всіх $n \geq 1$ істинні умови $bn \leq bn^2$ і $c \leq cn^2$. Отже, можна записати $t(n) = an^2 + bn + c \leq an^2 + bn^2 + cn^2 = (a + b + c) \cdot n^2$. Це нерівність в точності відповідає вимозі визначення O-нотації: $t(n) \leq c_x n^2$, де $c_x = a + b + c$, отже верхня межа складності алгоритму відповідає: $t(n) = O(n^2)$.

Найбільш часто зустрічаються наступні оцінки складності алгоритмів:

- $O(1)$ – обчислювальна складність алгоритму не залежить від розміру вхідних даних (константний порядок зростання);
- $O(n)$ – обчислювальна складність алгоритму лінійно зростає зі збільшенням вхідного масиву (лінійний порядок зростання);
- $O(\log n)$ – обчислювальна складність алгоритму зростає логарифмічно зі збільшенням розміру вхідного масиву (логарифмічний порядок зростання);
- $O(n \log n)$ – обчислювальна складність алгоритму зростає лінійно-логарифмічно зі збільшенням розміру вхідного масиву (лінеарітмічний порядок зростання);
- $O(n^2)$ – обчислювальна складність алгоритму зростає квадратично зі збільшенням вхідного масиву (квадратичний порядок зростання);
- $O(n^K)$ – обчислювальна складність алгоритму зростає поліноміально зі збільшенням вхідного масиву (поліноміальний порядок зростання);
- $O(2^n)$ – обчислювальна складність алгоритму зростає експоненціально зі збільшенням вхідного масиву (експоненціальний порядок зростання);

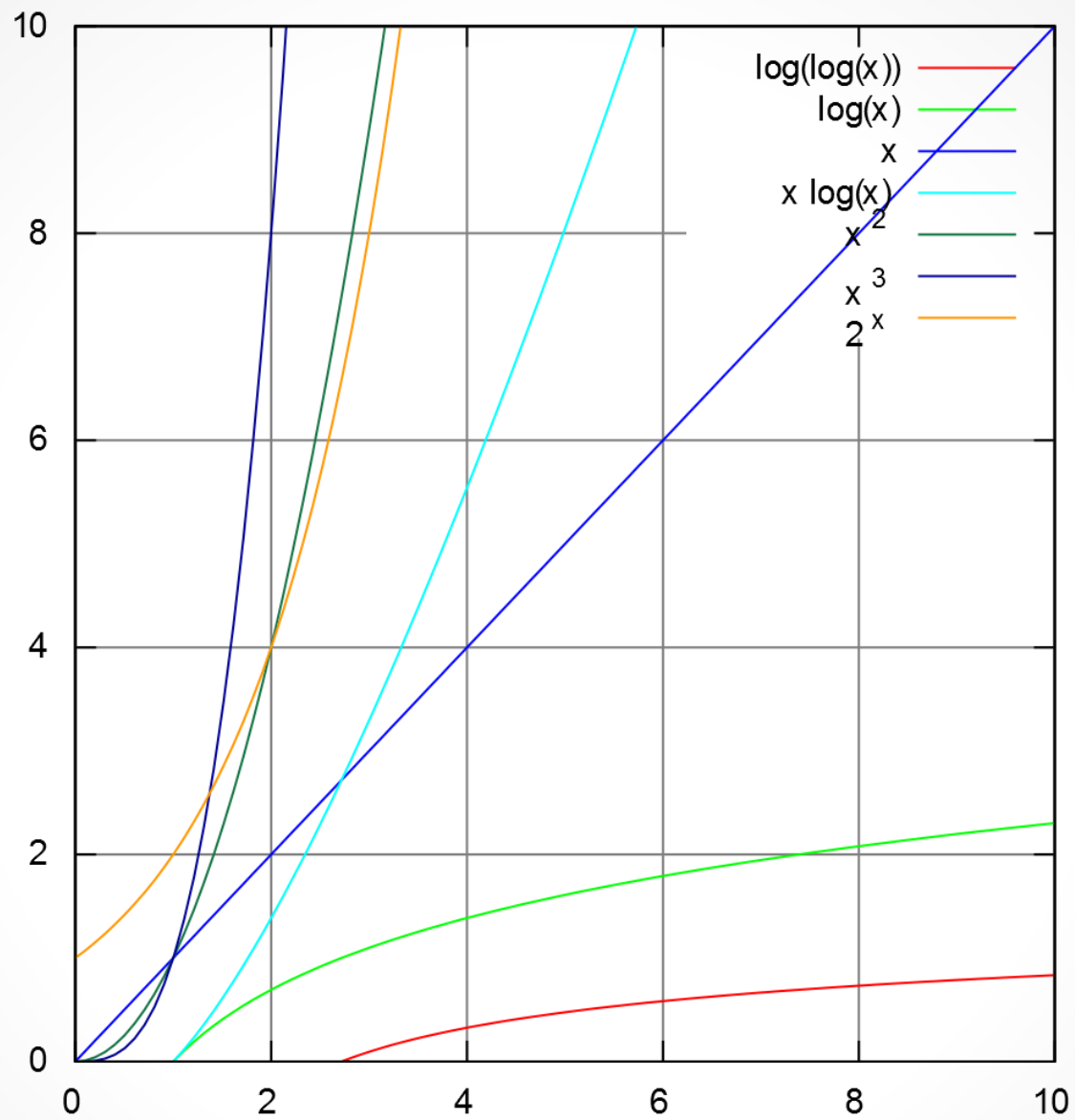


Рисунок 1

Припустимо, що масив з 1000000 об'єктів на заданому ЕОМ сортується 10ms. Потрібно оцінити верхню межу часу виконання, якщо потрібно впорядкувати масив з 5000000 елементів. Вважаємо, що оцінка складності алгоритму визначена і дорівнює $O(n^2)$.

Розрахунок. Відомо, що час виконання розраховується за виразом:

$$T(n) = C_t \cdot t(n), \text{ де } t(n) = O(n^2).$$

$$\text{Отже } T(n) \leq C_t \cdot C_x g(n) = C_t \cdot C_x \cdot n^2.$$

Тоді можна записати:

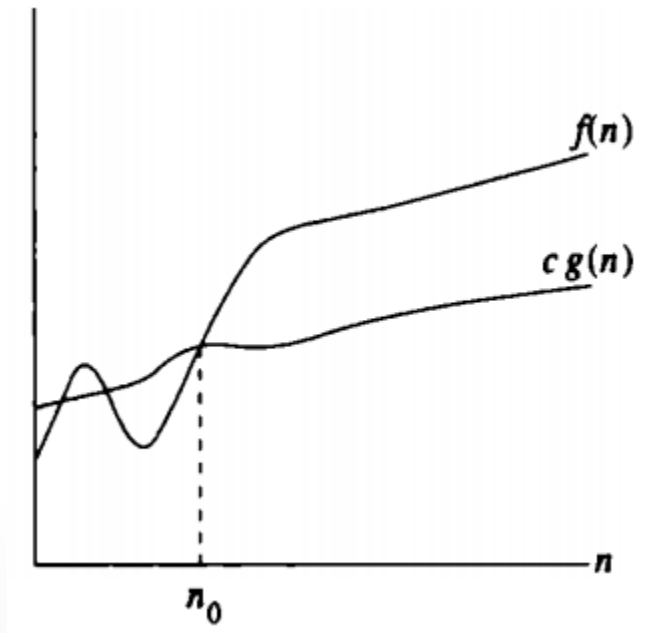
$$t_2/t_1 \leq C_t \cdot C_x \cdot n_2^2 / C_t \cdot C_x \cdot n_1^2 = n_2^2/n_1^2.$$

Звідки слідує, що:

$$t_2 \leq t_1 \cdot n_2^2/n_1^2 = 10 \cdot 5000000^2/1000000^2 = 10 \cdot 25 = 250 \text{ ms}.$$

Асимптотична нижня межа. Ω - нотація

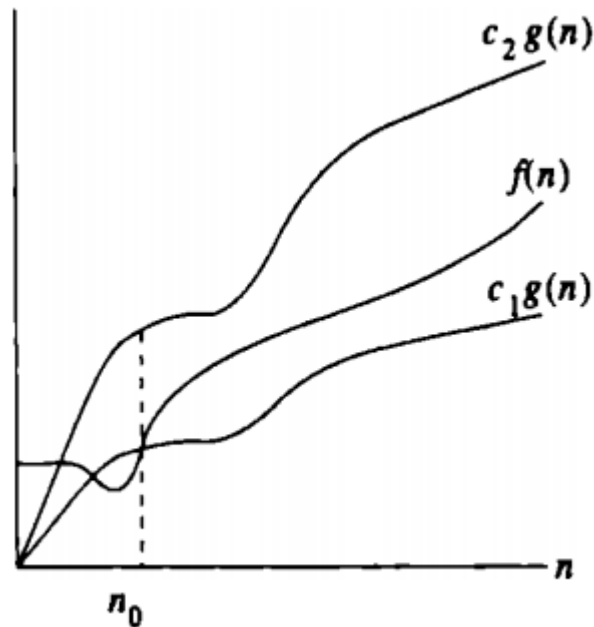
Нехай $t(n)$ – це функція (час виконання в кращому випадку для деякого алгоритму) з вхідними даними розміру n . Функція $t(n)$ має порядок $O(g(n))$ (належить до класу функцій $g(n)$), якщо існують такі константи $c > 0$ і $n_0 \geq 0$, що для всіх $n \geq n_0$ виконується умова $t(n) \geq c \cdot g(n)$. У такому випадку говорять, що $t(n)$ має **асимптотичну нижню межу** $g(n)$. Важливо підкреслити, що це визначення вимагає існування константи c , що працює для всіх n !



Асимптотична нижня та верхня (точна) межа.

Θ - нотація

Нехай $t(n)$ – це функція (час виконання) з вхідними даними розміру n . Функція $t(n)$ має порядок $O(g(n))$ (належить до класу функцій $g(n)$), якщо існують такі константи $c_1 > 0$, $c_2 > 0$ і $n_0 \geq 0$, що для всіх $n \geq n_0$ виконується умова $c_1 \cdot g(n) \leq t(n) \leq c_2 \cdot g(n)$. У такому випадку говорять, що $t(n)$ має **асимптотичну нижню та верхню межу** $g(n)$. Важливо підкреслити, що це визначення вимагає існування константи c , що працює для всіх n !



Приклад 3. O(1)

```
#include <stdio.h>
#include <windows.h>
#include <stdint.h>
```

$$T(n) = t_1 + t_2 = C; \Rightarrow \\ t(n) = O(1).$$

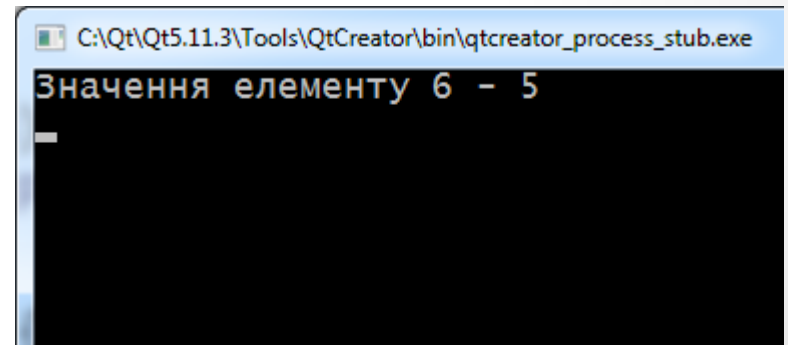
```
// Отримання значення з елементу масиву
```

```
int main()
{
    SetConsoleCP(CP_UTF8);
    SetConsoleOutputCP(CP_UTF8);

    uint32_t arr[] = {0, 1, 0, 5, 7, 2, 5, 0, 7, 7, 1, 0, 3, 4, 6, 0, 3};
    uint8_t len = sizeof(arr)/sizeof(arr[0]);
    uint8_t num = 6;
    uint32_t val = 255;

    if(num < len) {
        val = arr[num];
    }

    printf("Значення елементу %u - %u\n", num, val);
    return 0;
}
```



Приклад 4. $O(n)$

```
#include <stdio.h>
#include <windows.h>
#include <stdint.h>

// Розрахунок суми елементів масиву
uint32_t sum(uint8_t i, uint32_t * m) {
    if(i == 0)
        return m[0];
    return m[i] + sum(i-1, m);
}

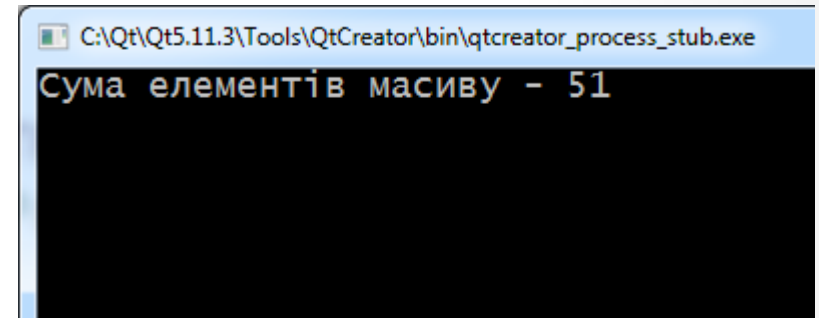
int main() {
    SetConsoleCP(CP_UTF8);
    SetConsoleOutputCP(CP_UTF8);

    uint32_t arr[] = {0, 1, 0, 5, 7, 2, 5, 0, 7, 7, 1, 0, 3, 4, 6, 0, 3};
    uint8_t len = sizeof(arr)/sizeof(arr[0]);

    uint32_t s = sum(len-1, &arr[0]);

    printf("Сума елементів масиву - %u\n", s);
    return 0;
}
```

$$\begin{aligned} T(n) &= t_1 * n + t_2 * n + t_3 + t_4 * (n-1) = \\ &= (t_1 + t_2 + t_4) * n + (t_3 - t_4) = C_1 * n + C_2; \\ \Rightarrow t(n) &= O(n). \end{aligned}$$



Приклад 5. $O(n^2)$

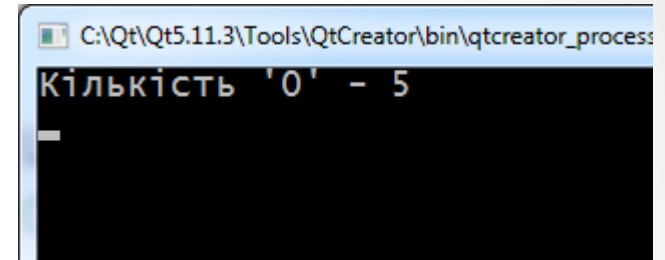
$T(n) = ?$

```
#include <stdio.h>
#include <windows.h>
#include <stdint.h>

// Знаходження кількості нулів в 2-мірному масиві
int main()
{
    SetConsoleCP(CP_UTF8);
    SetConsoleOutputCP(CP_UTF8);

    uint32_t arr[][4] = {{0, 1, 0, 5}, {7, 2, 5, 0}, {7, 7, 1, 0}, {3, 4, 6, 0}};
    uint8_t row = sizeof(arr)/sizeof(arr[0]);
    uint8_t col = sizeof(arr[0])/sizeof(arr[0][0]);
    uint8_t count = 0;

    for(uint8_t i=0; i < row; i++) {
        for(uint8_t j=0; j < col; j++) {
            if(arr[i][j] == 0) count++;
        }
    }
    printf("Кількість '0' - %u\n", count);
    return 0;
}
```



Приклад 6. $O(2^n)$

$T(n) = ?$

```
#include <stdio.h>
#include <windows.h>
#include <stdint.h>
```

```
// Розрахунок складної суми елементів масиву
uint32_t sum(uint8_t i, uint32_t * m) {
```

```
    if(i == 0)
        return m[0];
    if(i == 1)
        return m[0] + m[1];
```

```
    return sum(i-2, m) + sum(i-1, m);
```

```
}
```

```
int main(){
    uint32_t arr[] = {0, 1, 0, 5, 7, 2, 5, 0, 7, 7, 1, 0, 3, 4, 6, 0, 3};
    uint8_t len = sizeof(arr)/sizeof(arr[0]);
```

```
    uint32_t s = sum(len-1, &arr[0]);
```

```
    printf("Сума елементів масиву - %u\n", s);
    return 0;
```

```
}
```

