TiDA 03amb – Tablice deterministyczne, testy powtarzalności i analiza złożoności

---

Cele rozszerzone

1. Poznać **pseudolosowość deterministyczną** i pojęcie *ziarna (seed)*.
2. Zrozumieć ideę **powtarzalności eksperymentu** w testowaniu wydajności.
3. Umieć porównywać **różne rozkłady danych wejściowych**.
4. Umieć formalnie zinterpretować **złożoność obliczeniową i pamięciową** algorytmu.
5. Napisać kod **modularny i generyczny** (z użyciem szablonów i `std::chrono`).

Pseudolosowość deterministyczna

W komputerach nie istnieje *prawdziwa losowość*.
Funkcja `rand()` generuje liczby **pseudolosowe**, czyli takie, które *wydają się losowe*, ale w rzeczywistości są **wynikiem deterministycznego algorytmu**.

Ten algorytm zawsze zaczyna od **wartości początkowej (ziarna)**.
Dlatego:

- przy tym samym **seed** → zawsze ten sam ciąg liczb,
- przy innym **seed** → inny ciąg.

To kluczowe, jeśli chcemy testować wydajność i mieć **powtarzalne wyniki**.

Przykład

```cpp
#include <iostream>
#include <cstdlib>

int main() {
    unsigned int seed = 2025;
    srand(seed);

    std::cout << "Seed: " << seed << "\n";
    for (int i = 0; i < 10; i++)
        std::cout << rand() % 100 << " ";
}
```

Efekt: ten sam seed → identyczna sekwencja liczb.
Zmiana seeda → zmiana całej tablicy.

⚙️🔀 Generowanie różnych rozkładów danych

Aby sprawdzić wpływ danych na wydajność, generujemy tablice o różnych **dystrybucjach**:

```cpp
#include <algorithm> // std::iota, std::reverse
#include <numeric>   // std::iota
#include <vector>
#include <cstdlib>

enum class Distribution { SORTED, REVERSED, RANDOM };

std::vector<int> generate_data(size_t n, Distribution dist, unsigned int seed = 0) {
    std::vector<int> data(n);

    switch (dist) {
        case Distribution::SORTED:
            std::iota(data.begin(), data.end(), 0);
            break;
        case Distribution::REVERSED:
            std::iota(data.begin(), data.end(), 0);
            std::reverse(data.begin(), data.end());
            break;
        case Distribution::RANDOM:
            srand(seed);
            for (auto& x : data) x = rand() % 10000;
            break;
    }

    return data;
}
```

🔍 Zwróć uwagę:

- `std::iota` generuje sekwencję 0,1,2,...
- `std::reverse` odwraca kolejność.
- Użycie `enum class` poprawia czytelność i bezpieczeństwo typów.

⏱️ Szablonowy pomiar czasu dowolnej funkcji

```cpp
#include <chrono>
using namespace std::chrono;

template <typename F, typename... Args>
long long measure_us(F&& fn, Args&&... args) {
    auto start = high_resolution_clock::now();
    std::invoke(std::forward<F>(fn), std::forward<Args>(args)...);
    auto end = high_resolution_clock::now();
    return duration_cast<microseconds>(end - start).count();
}
```

📘 `std::invoke` umożliwia wywołanie dowolnej funkcji lub lambdy z przekazanymi argumentami. Dzięki temu `measure_us()` działa z każdą funkcją – nie tylko z `bubble_sort()`.

Algorytm sortowania (Bubble Sort)

```cpp
template <typename T>
void bubble_sort(std::vector<T>& v) {
    for (size_t i = 0; i < v.size() - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < v.size() - i - 1; j++) {
            if (v[j] > v[j + 1]) {
                std::swap(v[j], v[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break; // optymalizacja best-case → O(n)
    }
}
```

Zwróć uwagę:

- wersja generyczna (działa dla dowolnego typu porównywalnego),
- użyto optymalizacji: zatrzymanie, gdy tablica już posortowana.

Porównanie wyników

```cpp
#include <iostream>

int main() {
    const size_t N = 5000;
    const unsigned int SEED = 123;

    auto sorted = generate_data(N, Distribution::SORTED);
    auto reversed = generate_data(N, Distribution::REVERSED);
    auto random = generate_data(N, Distribution::RANDOM, SEED);

    std::cout << "SORTED:   " << measure_us(bubble_sort<int>, sorted) << " us\n";
    std::cout << "REVERSED: " << measure_us(bubble_sort<int>, reversed) << " us\n";
    std::cout << "RANDOM:   " << measure_us(bubble_sort<int>, random) << " us\n";
}
```

Interpretacja wyników

Typowe wyniki (dla N = 5000):

Rozkład danych	Czas (µs)	Przypadek	Złożoność
Posortowana	~10 000	Najlepszy	O(n)
Losowa	~90 000	Średni	$O(n^2)$
Odwrócona	~160 000	Najgorszy	$O(n^2)$

Zachowanie potwierdza teorię:

- algorytm bąbelkowy ma złożoność **kwadratową** w większości przypadków,
- tylko w sytuacji idealnej (już posortowane dane) zachowuje się **liniowo**.

📈 Złożoność obliczeniowa (ujęcie formalne)

🔷 Definicja:

Niech `T(n)` oznacza liczbę elementarnych operacji dla danych wejściowych rozmiaru `n`.
Mówimy, że:

> T(n) ∈ O(f(n)) ⇔ istnieją takie stałe `c > 0` i n_0, że dla każdego `n > n₀`
> zachodzi `T(n) ≤ c * f(n)`

czyli — dla dużych danych, algorytm nie rośnie szybciej niż pewna wielokrotność `f(n)`.

🔷 Przykład

Dla sortowania bąbelkowego:

`T(n) ≈ n*(n-1)/2  →  T(n) ≈ 0.5 * n²`

Pomijamy stałą `0.5`, bo w notacji dużego O interesuje nas **tempo wzrostu**, nie dokładny współczynnik.

➡️ Zatem `T(n) ∈ O(n²)`

🔷 Złożoność pamięciowa

Bubble Sort nie używa dodatkowych struktur danych oprócz kilku zmiennych pomocniczych.
Dlatego:
➡️ **Pamięciowa złożoność = O(1)** (działa „w miejscu”).

🧠 8️⃣ Powtarzalność eksperymentu

Aby wyniki testów były **wiarygodne**, należy zadbać o:

- ten sam **seed**,
- to samo **środowisko (Release/Debug, x86/x64)**,
- identyczny **rozmiar tablicy i algorytm**,
- **kilkukrotne uruchomienie** i obliczenie średniego czasu.

📘 Przykład pomiaru wielokrotnego:

```
template <typename F, typename... Args>
double measure_avg_us(int trials, F&& fn, Args&&... args) {
    double total = 0.0;
    for (int i = 0; i < trials; i++)
        total += measure_us(std::forward<F>(fn), std::forward<Args>(args)...);
    return total / trials;
}
```

🎓 Zadania (do oddania w formie sprawozdania)

1. 🧪 Dodaj pomiar średniego czasu z 5 powtórzeń (`measure_avg_us`).
2. 📈 Zbadaj, jak rośnie czas sortowania dla `N = 1000, 2000, 4000, 8000, 16000` i wykreśl wykres log–log.
3. 🔍 Zmodyfikuj kod, aby testował inne algorytmy (np. `std::sort`, `std::stable_sort`) i porównaj wyniki.
4. 💡 Wyjaśnij, dlaczego powtarzalność eksperymentu jest kluczowa w nauce i inżynierii oprogramowania.