

TiDA 03: Tablice deterministyczne i testy powtarzalności

Cel zajęć

Na tych zajęciach:

- Zrozumiesz pojęcia **powtarzalności eksperymentu**.
 - Poznasz lepiej funkcje **srand(seed)** i **rand()**.
 - Porównasz czasy sortowania różnych tablic (posortowane, odwrócone, losowe).
 - Wprowadzenie pojęcia **złożoności obliczeniowej $O(n^2)$** i **złożoności pamięciowej**.
-

Część teoretyczna

Losowość w programowaniu

Funkcja **rand()** generuje liczby **pseudolosowe** – nie są naprawdę losowe, lecz **zależne od ziarna (ang. seed)**.

 Jeśli użyjesz tego samego seeda, otrzymasz zawsze **ten sam ciąg liczb**.
Dzięki temu można powtarzać eksperymenty w sposób kontrolowany.

Przykład

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

int main() {
    srand(123); // ustawiamy ziarno (seed)

    for (int i = 0; i < 10; i++) {
        std::cout << rand() % 100 << " ";
    }

    return 0;
}
```

Efekt:

Za każdym uruchomieniem programu z tym samym seedem (123) wyniki będą identyczne.
Zmiana seeda = zmiana ciągu liczb.

Powtarzalność eksperymentu

W testowaniu wydajności chcemy, by wyniki dało się **powtórzyć**.
Dlatego zapisujemy: - długość tablicy,

- ziarno (seed),
 - sposób pomiaru czasu,
 - wersję kompilacji (Debug / Release),
 - architekturę (x86 / x64).
-

◆ Typowe rozkłady danych wejściowych

Dla sortowania warto porównywać trzy przypadki:

1. **Już posortowana tablica** – najlepszy przypadek.
 2. **Odwrotnie posortowana tablica** – najgorszy przypadek.
 3. **Losowa tablica** – przypadek przeciętny.
-

💻 Część praktyczna – przykład kodu

◆ Generowanie tablic deterministycznych

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

void fill_random(int* tab, int n, unsigned int seed) {
    srand(seed);
    for (int i = 0; i < n; i++) {
        tab[i] = rand() % 1000;
    }
}

void fill_sorted(int* tab, int n) {
    for (int i = 0; i < n; i++) {
        tab[i] = i;
    }
}

void fill_reversed(int* tab, int n) {
    for (int i = 0; i < n; i++) {
        tab[i] = n - i;
    }
}
```

◆ Prosta funkcja sortująca (Bubble Sort)

```
void bubble_sort(int* tab, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (tab[j] > tab[j + 1]) {
                std::swap(tab[j], tab[j + 1]);
            }
        }
    }
}
```

◆ Pomiar czasu sortowania

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

long long measure_time(int* tab, int n) {
    auto start = high_resolution_clock::now();
    bubble_sort(tab, n);
    auto end = high_resolution_clock::now();
    return duration_cast<microseconds>(end - start).count();
}
```

◆ Test porównawczy

```
int main() {
    const int N = 5000;
    int* tab = new int[N];

    fill_sorted(tab, N);
    std::cout << "Posortowana: " << measure_time(tab, N) << " us\n";

    fill_reversed(tab, N);
    std::cout << "Odwrócone: " << measure_time(tab, N) << " us\n";

    fill_random(tab, N, 123);
    std::cout << "Losowa: " << measure_time(tab, N) << " us\n";

    delete[] tab;
}
```

📊 Analiza wyników

- **Posortowana** tablica sortuje się najszybciej,
- **Odwrotnie posortowana** – naj wolniej,
- **Losowa** ma czas pośredni.

To pokazuje, że nawet prosty algorytm ma **różną złożoność w praktyce**.

[Z] Złożoność obliczeniowa

◆ Co oznacza O(...)

Notacja **O(...)** (czyt. *big O*) opisuje, jak szybko rośnie czas działania algorytmu, gdy zwiększamy liczbę danych wejściowych.

Nie chodzi o sekundy, lecz o **tempo wzrostu**.

◆ O(1) – czas stały

Czas nie zależy od liczby elementów.

```
int get_first(int* tab) {
    return tab[0];
}
```

→ Złożoność: **O(1)**

◆ O(n) – czas liniowy

Czas rośnie proporcjonalnie do liczby elementów.

```
void print_all(int* tab, int n) {
    for (int i = 0; i < n; i++)
        std::cout << tab[i] << " ";
}
```

→ Złożoność: **O(n)**

◆ O(n^2) – czas kwadratowy

Dla każdego elementu wykonujemy pętlę po pozostałych elementach.

```
void bubble_sort(int* tab, int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (tab[j] > tab[j + 1])
                std::swap(tab[j], tab[j + 1]);
}
```

→ Złożoność: **O(n^2)**

◆ Porównanie różnych złożoności

Złożoność	Tempo wzrostu	Przykład
O(1)	Stał	Odczyt elementu z tablicy
O(log n)	Logarytmiczny	Przeszukiwanie binarne
O(n)	Liniowy	Przejście przez tablicę
O(n log n)	Log-liniowy	QuickSort, MergeSort
O(n^2)	Kwadratowy	Bubble Sort
O(2^n)	Bardzo szybki wzrost	Naiwny rekurencyjny Fibonacci
O(n!)	Superwykładniczy	Pełne przeszukiwanie permutacji

ZADANIA

1.  Zmień wartość seed w `fill_random()` i zobacz, jak zmienia się tablica.
2.  Sprawdź na komputerach kolegów obok czy dla tego samego seeda mająte same wyniki w tabeli 10 elementowej
3.  Porównaj czasy sortowania dla:
 - posortowanej tablicy,
 - odwrotnie posortowanej,
 - losowej.
4.  Dodaj do `bubble_sort()` warunek przerywania, gdy w danym przebiegu nie było zamiany.
5.  Zmierz czasy dla różnych rozmiarów tablicy (np. 1000, 2000, 5000, 10000).
6.  Wyjaśnij, dlaczego test z tym samym seed jest ważny dla wiarygodności wyników.