# Sprawozdanie z zajęć (WZÓR na podstawie lekcji nr 02$^{amb}$)

## Zalecenia dotyczące dostarczanego pliku dokumentu ze sprawozdaniem

Plik sprawozdania powinien być dostarczony w formacie **PDF** lub **DOCX** i powinien być kompletny, bez odwołań do innych dokumentów.

**Nazwa pliku powinna być w następującym formacie:**

```
NAZWISKO_Imię_Przedmiot_NrLekcji_Temat_(Poziom).format
```

**Przykład:**

```
NOWAK_Jan_TiDA_02_Tablice_dynamiczne_i_statyczne_(podstawowy).pdf
```

| Sprawozdanie z zajęć **„Testowanie i dokumentowanie aplikacji”** | |
|---|---|
| **Temat:** | Tablice dynamiczne i statyczne, zarządzanie pamięcią, testy funkcjonalne i wydajnościowe |
| **Wykonawca:** | Jan NOWAK |
| **Data:** | 26.10.2025 |
| **Klasa:** | 4d (technik programista) |
| **Poziom:** | Rozszerzony (dla ambitnych) |

## Cel ćwiczenia

1. Zbadanie granicy pamięci stosu i wpływu wielkości tablicy na wystąpienie błędu *stack overflow*.
2. Porównanie wydajności sortowania tablic statycznych i dynamicznych przy użyciu `sortBubble` i `std::sort`.
3. Analiza wpływu architektury procesora (x86 vs x64) na czas wykonywania programu.
4. Obserwacja rozmieszczenia pamięci tablic (adresy stosu i sterty) przy pomocy debugera.
5. Rozszerzenie testów funkcjonalnych o sprawdzanie unikalności danych w tablicy.

---

## Środowisko testowe

| Parametr | Wartość |
|---|---|
| System operacyjny | Windows 11 Pro 64-bit |
| Kompilator | Microsoft Visual C++ 2022 |
| Tryby kompilacji | x86 (32-bit), x64 (64-bit) |
| Procesor | Intel Core i7-9700 @ 3.0 GHz |
| Pamięć RAM | 16 GB |
| Optymalizacja kompilacji | `/O2` (maksymalna optymalizacja) |

---

## Opis programu

Program tworzy tablice statyczne (na stosie) oraz dynamiczne (na stercie) i wykonuje na nich sortowanie:

- metodą **bąbelkową (Bubble Sort)**,
- oraz **`std::sort()`** z biblioteki standardowej C++.

Dodatkowo wykonano **test funkcjonalny**, który:

- sprawdza, czy tablica istnieje,
- czy wartości mieszczą się w określonym zakresie (`MIN`, `MAX`),
- czy dane są posortowane rosnąco,
- czy nie występują duplikaty (rozszerzony test).

Pomiar czasu wykonania jest realizowany funkcją `measure_us()` opartą o `<chrono>`.

---

## 🧪 Przebieg eksperymentów

### Zadanie 1 – Granica stosu

W programie stopniowo zwiększano rozmiar tablicy statycznej, zaczynając od 10 000 elementów. Dla każdej próby mierzono czas sortowania i obserwowano, czy program kończy się błędem.

| Rozmiar tablicy | Wynik |
|---|---|
| 10 000 | OK |
| 20 000 | OK |
| 40 000 | OK |
| 80 000 | OK (x64), Stack Overflow (x86) |
| 120 000 | Stack Overflow (x64) |

**Wniosek:** W architekturze **x86** błąd *stack overflow* pojawia się już przy ok. **80 000 elementach**, natomiast w **x64** – przy ok. **120 000 elementach**. Wynika to z większego domyślnego rozmiaru stosu w 64-bitowej aplikacji (ok. 8 MB zamiast 1 MB).

---

### Zadanie 2 – Optymalizacja sortowania (`std::sort()`)

Porównano czasy sortowania tablic o rozmiarach 10 000 i 100 000 elementów przy użyciu dwóch algorytmów:

| Algorytm | Typ tablicy | 10 000 elementów (µs) | 100 000 elementów (µs) |
|---|---|---|---|
| `sortBubble` | statyczna | 460 000 | 4 800 000 |
| `sortBubble` | dynamiczna | 490 000 | 5 000 000 |
| `std::sort()` | statyczna | 2 400 | 24 000 |
| `std::sort()` | dynamiczna | 2 500 | 26 000 |

**Wnioski:**

- `std::sort()` (algorytm introspektywny – połączenie QuickSort + HeapSort + InsertionSort) jest **kilkaset razy szybszy** od `sortBubble`.
- Po użyciu `std::sort()` różnica między tablicą statyczną a dynamiczną **praktycznie zanika** – czas dostępu do pamięci przestaje być czynnikiem dominującym.

---

## Zadanie 3 – Porównanie architektur (x86 vs x64)

Dla `std::sort()` i rozmiaru 100 000 elementów uzyskano wyniki:

| Architektura | Typ tablicy | Czas (µs) |
|---|---|---|
| x86 (32-bit) | statyczna | 26 300 |
| x86 (32-bit) | dynamiczna | 27 200 |
| x64 (64-bit) | statyczna | 22 400 |
| x64 (64-bit) | dynamiczna | 23 000 |

**Wnioski:**

- Architektura **x64** była ok. **15–20% szybsza**.
- Powód: większa liczba rejestrów ogólnego przeznaczenia (16 zamiast 8), oraz bardziej wydajny model adresowania pamięci.

---

## Zadanie 4 – Analiza pamięci (debuger)

Przy użyciu Visual Studio sprawdzono adresy tablic w pamięci:

| Typ tablicy | Adres (x64) | Obszar |
|---|---|---|
| Statyczna | 0x00000094F5A0 | Stos |
| Dynamiczna | 0x000001F21B000 | Sterta |

Adresy różnią się o rzędy wielkości — potwierdza to, że stos i sterta znajdują się w **różnych częściach przestrzeni adresowej**. Wersja 64-bitowa dodatkowo rozkłada obszary pamięci znacznie szerzej, co zwiększa bezpieczeństwo (ASLR – Address Space Layout Randomization).

---

## Zadanie 5 – Rozszerzenie testów funkcjonalnych

Dodano funkcję wykrywającą duplikaty w tablicy:

```
bool czyBezDuplikatow(int* tablica, int rozmiar) {
    for (int i = 1; i < rozmiar; ++i)
        if (tablica[i - 1] == tablica[i])
            return false;
    return true;
}
```

Włączono ją do `testFunkcjonalny`:

```
return czyTablicaPoprawna(tablica, rozmiar, MIN, MAX) &&
       czyTablicaPosortowana(tablica, rozmiar) &&
       czyBezDuplikatow(tablica, rozmiar);
```

**Wynik:** Przy danych losowych sporadycznie występowały duplikaty (ok. 0,3% przypadków), co zostało poprawnie wykryte.

## Analiza zbiorcza

| Obserwacja | Wniosek |
|---|---|
| Stos ma ograniczony rozmiar | Przy dużych tablicach powoduje *stack overflow* |
| Sterta jest wolniejsza w alokacji, ale bezpieczniejsza | Brak limitu rozmiaru tablic |
| `std::sort()` likwiduje różnice między stosem a stertą | CPU i cache dominują nad czasem dostępu do pamięci |
| Architektura x64 działa szybciej | Więcej rejestrów, większa przestrzeń adresowa |
| Test unikalności poprawnie wykrywa duplikaty | Działa poprawnie po sortowaniu danych |

## Wnioski końcowe

1. **Granica stosu** w 64-bitowych programach jest znacznie większa, jednak mimo to dla dużych tablic należy używać alokacji dynamicznej.
2. **`std::sort()`** znacząco przewyższa ręcznie implementowane algorytmy – różnice rzędu setek razy.
3. Architektura **x64** daje realny wzrost wydajności przy dużych danych, głównie przez lepsze wykorzystanie rejestrów i pamięci podręcznej.
4. Testy funkcjonalne i wydajnościowe muszą być prowadzone równolegle, bo szybki kod nie zawsze jest poprawny logicznie.
5. Zrozumienie różnic między stosem a stertą jest kluczowe przy analizie błędów pamięci i optymalizacji programów w C++.

📎 Załącznik – fragment kodu z pomiarem czasu

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