TiDA 07: Benchmarki – jak mierzyć wydajność programów?

Cel zajęć

Na tych zajęciach poznacie zasady rzetelnego mierzenia wydajności programów w Pythonie i C++. Nauczycie się wykonywać powtarzalne eksperymenty, interpretować wyniki i unikać typowych pułapek. Ćwiczenia pozwolą porównywać działanie tych samych operacji w dwóch różnych językach.

Dlaczego w ogóle mierzymy wydajność?

Wyobraźcie sobie, że chcecie sprawdzić, **jak szybko potraficie zrobić kanapkę**. Ale warunki za każdym razem są inne:

- raz masło jest miękkie, raz twarde jak skała,
- raz chleb leży pod ręką, a innym razem musicie biec po niego do drugiego pokoju,
- czasem ktoś co chwilę Was zagaduje,
- a czasem nóż akurat wylądował w zmywarce i trzeba szukać innego.

Gdybyście za każdym razem mieli inne warunki, wyniki pomiaru byłyby kompletnie nieporównywalne.

Tak samo jest z programami:

- czasem komputer jest obciążony innymi procesami,
- czasem dane są już „pod ręką" procesora, a czasem trzeba je dopiero znaleźć,
- czasem kompilator wykona optymalizację, a czasem nie,
- czasem kod jest gorący i gotowy, a innym razem dopiero się „rozgrzewa".

Właśnie dlatego benchmark musi być **kontrolowanym eksperymentem** — takim, w którym:

- masło jest *zawsze* tak samo miękkie,
- chleb jest *zawsze* na tym samym talerzu,
- nikt Wam nie przeszkadza,
- i zawsze używacie tego samego noża.

Czyli warunki muszą być powtarzalne, by można było porównywać wyniki.

Benchmark to sposób na stworzenie **kontrolowanego, powtarzalnego eksperymentu**, który pozwala porównywać różne rozwiązania.

Co wpływa na wynik benchmarku?

W Pythonie

- interpreter musi się „rozgrzać”,
- garbage collector może włączyć się w dowolnym momencie,
- niektóre operacje są napisane w C i działają bardzo szybko,
- losowość wymaga ustawienia ziarna (seed), aby uniknąć przypadkowych różnic.

Python działa jak kuchenka gazowa: czasem płomień jest ciut większy, czasem mniejszy — wyniki mogą delikatnie falować.

W C++

- w trybie Debug kompilator nie optymalizuje kodu i działa on dużo wolniej,
- w trybie Release kod jest optymalizowany i działa znacznie szybciej,
- kompilator może usunąć nieużywany kod (tzw. dead code elimination).

C++ jest jak precyzyjna kuchenka indukcyjna: jeśli warunki ustawimy odpowiednio, uzyskujemy bardzo przewidywalne rezultaty.

Wspólne źródła zakłóceń

- inne procesy działające w tle,
- przełączanie zadań przez system,
- pierwsze uruchomienie kodu (zimny cache),
- ruch danych między pamięcią RAM a pamięcią podręczną CPU.

i Co to jest „cold cache”?

To określenie nie ma nic wspólnego z temperaturą procesora. „Cold cache” oznacza po prostu, że **dane nie są jeszcze załadowane do pamięci podręcznej CPU**, więc pierwsze wykonanie kodu może być wolniejsze. Kolejne wykonania są szybsze, bo dane są już „pod ręką” procesora.

Czynniki mające wpływ na wydajność (Podsumowanie)

- Predykcja skoków (branch prediction)
- Nietrafienia predykcji (branch misprediction)
- Pipeline procesora i jego zatrzymania (pipeline stalls / pipeline flush)
- Hazard danych (data hazards: RAW, WAR, WAW)
- Pamięć podręczna CPU (cache L1/L2/L3)
- Nietrafienia w cache (cache misses)
- Różnice między danymi „zimnymi” a „rozgrzanymi” (cold cache / warm cache)
- Lokalność danych (data locality) – przestrzenna i czasowa
- Różnice w algorytmach sortowania i ich zachowaniu w praktyce
- Różnice między złożonością teoretyczną a rzeczywistą wydajnością

- Optymalizacje kompilatora (inlining, loop unrolling, eliminacja martwego kodu)
- Tryb kompilacji Debug vs Release
- Ograniczenia wynikające z przepustowości pamięci (memory-bound)
- Ograniczenia wynikające z mocy obliczeniowej CPU (compute-bound)
- Wpływ innych procesów i scheduler'a systemu operacyjnego
- Wpływ generatorów liczb pseudolosowych i różnic między implementacjami
- Różnice architekturalne między językami (interpretowane vs kompilowane)
- Wpływ rozgrzewki kodu (warmup phase) na wyniki benchmarków

🔍 Cechy dobrego benchmarku

- **Powtarzalność** – te same warunki dają podobne wyniki.
- **Uczciwość** – porównywane są identyczne operacje.
- **Wielokrotne powtórzenia** – jeden pomiar nic nie znaczy.
- **Dokładny opis eksperymentu** – jak mierzono, na jakich danych, na jakim sprzęcie.

To tak jak w laboratorium chemicznym: trzeba zadbać o powtarzalne warunki, inaczej eksperyment jest bezwartościowy.

Benchmarkowanie w Pythonie

Dlaczego `timeit`?

`timeit`:

- wykonuje wiele powtórzeń kodu,
- działa powtarzalnie,
- minimalizuje wpływ losowych opóźnień systemu.

Nie stosujemy `time.time()` do pojedynczego pomiaru — to jak używanie kuchennego minutnika do mierzenia sprintu.

Przykład – mierzenie czasu sortowania listy

```
import timeit
import random

def create_random_list(n, seed):
    random.seed(seed)
    return [random.randint(0, 1_000_000) for _ in range(n)]

setup_code = """
from __main__ import create_random_list
"""

test_code = """
lst = create_random_list(5000, 123)
lst.sort()
"""

print(min(timeit.repeat(stmt=test_code,
                        setup=setup_code,
                        repeat=5,
                        number=1)))
```

- `repeat=5` – liczba serii pomiarów
- `number=1` – ile razy w serii wykonywany jest kod
- `min(...)` – wybieramy wynik najmniejszy (najmniej zakłócony)

Benchmarkowanie w C++

Mierzenie czasu za pomocą `std::chrono`

```cpp
#include <chrono>

using namespace std::chrono;

auto start = high_resolution_clock::now();
// kod testowany
auto end = high_resolution_clock::now();

auto micros = duration_cast<microseconds>(end - start).count();
```

Dlaczego tryb Release jest obowiązkowy?

W trybie Debug:

- kod jest spowalniany przez dodatkowe sprawdzenia,
- kompilator nie stosuje optymalizacji,
- wyniki są często 5–20 razy wolniejsze.

To jak ocenianie auta sportowego podczas jazdy z zaciągniętym hamulcem, albo jak pomiar czasu na 100m sprintera biegnacego w gumowcach.

Przykład – sortowanie w C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <cstdlib>
using namespace std;
using namespace std::chrono;

vector<int> create_random_vector(int n, unsigned seed) {
    srand(seed);
    vector<int> v(n);
    for (int i = 0; i < n; i++)
        v[i] = rand() % 1'000'000;
    return v;
}

long long measure_sort(vector<int> v) {
    auto start = high_resolution_clock::now();
    sort(v.begin(), v.end());
    auto end = high_resolution_clock::now();
    return duration_cast<microseconds>(end - start).count();
}

int main() {
    auto v = create_random_vector(5000, 123);
    cout << measure_sort(v) << " us\n";
}
```

⚖️ Porównywanie Pythona i C++

❗ Ważna informacja dotycząca losowości

Ten sam seed w Pythonie i C++ nie daje tych samych losowych danych.

Dlaczego?

- Python używa generatora pseudolosowego Mersenne Twister (ale w własnej implementacji),
- `rand()` w C++ najczęściej używa generatora LCG,
- oba języki inaczej „rozprowadzają” seed wewnątrz generatora.

👉 Ustawienie `seed=123` w obu językach NIE gwarantuje identycznych wartości.

---

i Jak uzyskać identyczne dane wejściowe?

Jedyna w pełni pewna metoda:

Generujemy dane w jednym języku (np. Pythonie), zapisujemy je do pliku (CSV, TXT), a następnie wczytujemy je zarówno w Pythonie, jak i w C++.

Dzięki temu obie implementacje pracują na dokładnie tych samych liczbach.

Zadania

Zadanie 1 – Benchmark Python

W Pythonie zmierzcie czas sortowania listy 5000 elementów w trzech wariantach:

- lista posortowana,
- lista odwrócona,
- lista losowa (wczytana z pliku).

Dla każdego przypadku wykonajcie **5 pomiarów** i zapiszcie najlepszy.

Zadanie 2 – Benchmark C++

W C++ wykonajcie benchmark sortowania tych samych danych:

- posortowanych,
- odwróconych,
- losowych (wczytanych z pliku).

Pamiętajcie o kompilacji w trybie Release.

Zadanie 3 – Porównanie wyników

Przygotujcie tabelę:

Dane wejściowe	Python (ms)	C++ (ms)	Różnica
posortowane			
odwrócone			
losowe			

Zadanie 4 – Stabilność wyników

Porównajcie, jak bardzo wyniki różnią się pomiędzy kolejnymi próbami:

- w Pythonie,
- w C++.

Zastanówcie się: który język jest stabilniejszy i dlaczego?

◈ Zadanie 5 – Dodatkowa operacja

Wybierzcie jedną dodatkową operację, np.:

- sumowanie listy/wektora,
- wyszukiwanie elementu,
- mnożenie małych macierzy.

Wykonajcie pomiar w obu językach i porównajcie wyniki.

📝 Sprawozdanie – co powinno zawierać?

📄 Niezbędne elementy

- opis celu i przebiegu eksperymentu,
- kod wykorzystany w benchmarkach (Python + C++),
- opis sposobu generowania danych i użyty plik,
- tabela z wynikami,
- krótkie wnioski.

❓ Pytania do wniosków

- Dlaczego Python i C++ nie generują takich samych „losowych” danych przy tym samym seedzie?
- Który przypadek danych (posortowany/odwrócony/losowy) był najszybszy i dlaczego?
- Dlaczego w C++ tryb Release jest niezbędny?
- W którym języku wyniki były stabilniejsze?
- Jakie czynniki mogły zakłócić pomiary?