Automatyczny benchmark: porównanie wydajności C++ i Pythona

Opis ogólny

Zespół dwuosobowy ma przygotować **w pełni zautomatyzowany mini–system testowy**, który na dowolnym komputerze:

1. Przygotuje dane testowe (wejściowe) do pomiarów.
2. Zbierze informacje o środowisku uruchomieniowym.
3. Skompiluje i uruchomi program testowy w C++.
4. Uruchomi równoważny program testowy w Pythonie.
5. Zweryfikuje poprawność wykonania zadania w obu przypadkach (test funkcjonalny)
6. Zbierze wyniki z obu programów (z ich outputu).
7. Wygeneruje automatycznie **raport z testów** (w Markdown).

Projekt ma podsumować treści z całego cyklu zajęć: sortowanie, pomiar czasu, parsowanie outputu, praca z CSV, kompilacja z linii komend, automatyzacja w Pythonie, raport w formacie MD.

Cel projektu

- Przećwiczenie w praktyce:
    - pisania kodu testowego w C++ i Pythonie,
    - pomiaru czasu wykonania,
    - pracy ze skryptami automatyzującymi (Python),
    - kompilacji z linii komend (w tym z parametrami optymalizacji),
    - parsowania wyników i pracy z plikami tekstowymi / CSV,
    - tworzenia raportu technicznego w Markdown.

Zakres funkcjonalny – co system ma robić automatycznie

Po uruchomieniu głównego skryptu (np. `run_benchmark.py`) system powinien:

1. **Przygotować dane testowe**

- Wygenerować dane do testów (np. losowe tablice liczb całkowitych o różnych rozmiarach: 10 000, 50 000, 100 000, ...).
- Zapisać dane do pliku/plików (np. CSV lub prosty tekst), tak aby:
    - program C++ i program Python korzystały z **tych samych danych**,
    - można było powtórzyć test (deterministyczne dane).

2. **Zebrać informacje o środowisku**

- Skrypt powinien zebrać informacje i zapisać je do raportu, m.in.:
    - system operacyjny (np. Windows 10/11, Linux, wersja),
    - architekturę (np. x86_64),
    - procesor:
        - nazwa/model CPU,
        - liczba rdzeni/wątków (jeśli da się łatwo odczytać),
- pamięć RAM (przynajmniej przybliżona informacja, jeśli dostępna),
- wersja kompilatora C++ (np. `cl`, `g++`, `clang++` + numer wersji),
- zastosowane **flagi kompilacji** (np. `/O2`, `/Od`, `-O2`, `-O0`),
- wersja Pythona (np. `3.11.x`),
- używane biblioteki (jeśli jakieś dodatkowe są wymagane – wypisać nazwy).

3. **Skompilować i uruchomić część C++**

- Skrypt w Pythonie ma wywołać kompilator C++ z linii komend, np.:
    - Windows: `cl /O2 main.cpp /Fe:benchmark_cpp.exe`
    - Linux: `g++ -O2 main.cpp -o benchmark_cpp`
- Program C++ powinien:
    - wczytać przygotowane wcześniej dane z pliku,
    - wykonać wybrane operacje testowe (np. sortowanie tablicy, obliczenia na tablicach, kilka różnych rozmiarów danych),
    - **zmierzyć czas** (np. `std::chrono`),
    - wypisać wyniki w **ustrukturyzowany sposób**, np. w jednym wierszu: `ALGORYTM;ROZMIAR;CZAS_MS;LICZBA_POWTÓRZEŃ` (lub podobny, ale konsekwentny format),
    - upewnić się, że wynik można łatwo sparsować (np. CSV lub linie z ustalonym formatem).
    - zweryfikować poprawność działania algorytmu (czy dane są posortowane)

4. **Uruchomić część w Pythonie**

- Skrypt ma uruchomić równoważny program w Pythonie:
    - ten sam algorytm / operacja co w C++,
    - te same dane wejściowe,
    - pomiar czasu (np. `time.perf_counter()` z wieloma powtórzeniami),
    - wypisanie wyników w **tym samym formacie**, co w C++ (lub bardzo zbliżonym).
- Zweryfikować poprawność działania algorytmu (czy dane są posortowane)
- Dane wyjściowe dla Pythona również powinny zawierać:
    - nazwę algorytmu,
    - rozmiar danych,
    - czas wykonania,
    - liczbę powtórzeń.

5. **Zebrać wyniki z outputu**

- Główny skrypt Pythona:
    - uruchamia program C++ (np. `subprocess.run`), odczytuje jego `stdout`,
    - uruchamia program Pythona (jako osobny proces lub funkcje wewnątrz), odczytuje wyniki,
    - parsuje tekst wyjściowy,
    - zapisuje całość do:
        - pliku CSV **lub**
        - struktury w Pythonie (lista słowników / lista wierszy), a następnie do CSV/MD.

6. **Wygenerować automatyczny raport**

- Raport w formacie **Markdown** (np. `report_<data>.md`), zawierający:
  - tytuł i krótki opis testu,
  - sekcję **Środowisko testowe** (zebrane informacje o systemie, kompilatorze, Pythonie),
  - sekcję **Parametry testu**:
    - jakie algorytmy badano (np. sortowanie bąbelkowe, std::sort vs. sort Pythona),
    - jakie rozmiary danych,
    - liczba powtórzeń dla każdego pomiaru,
    - użyte flagi kompilacji,
  - sekcję **Wyniki**:
    - tabele w Markdown, np.:

```
| Język  | Algorytm | Rozmiar | Średni czas [ms] | Odchylenie / min-max |
|--------|----------|---------|------------------|----------------------|
| C++    | sort     | 10000   | 1.23             | 1.20–1.26            |
| Python | sort     | 10000   | 5.67             | 5.50–5.80            |
```

  - wykresy mogą być opcjonalne (np. wygenerowane przez Pythona do plików PNG i wstawione jako obrazki w MD),
  - sekcję **Wnioski**:
    - porównanie wydajności C++ vs Python,
    - komentarz dotyczący wpływu rozmiaru danych na czas,
    - krótka refleksja o roli optymalizacji kompilatora i interpretacji Pythona.

Wymagania techniczne

- Główny „orchestrator" projektu: **Python** (jeden skrypt uruchamia wszystko).
- Program C++:
  - jeden plik `main.cpp` lub mały projekt, ale kompilowany z linii komend,
  - pomiar czasu za pomocą `std::chrono`.
- Program Python:
  - użycie `time.perf_counter()` lub analogicznej funkcji,
  - możliwość wielokrotnego powtarzania testu dla lepszej stabilności pomiaru.
- Raport:
  - plik `.md` tworzony automatycznie przez skrypt (nie ręcznie w edytorze),
  - nazwa raportu powinna zawierać datę/godzinę lub timestamp.

Proponowany podział pracy w zespole

To tylko sugestia – zespół może podzielić się inaczej, ale:

- **Osoba A (C++ / wydajność)**:
    - implementuje kod testowy w C++,
    - dba o prawidłowy pomiar czasu i format outputu,
    - testuje różne flagi kompilacji (np. `-O0`, `-O2`) i przygotowuje wyniki.
- **Osoba B (Python / automatyzacja / raport)**:
    - tworzy skrypt Pythona, który:
        - generuje dane testowe,
        - wywołuje kompilację C++ i uruchamia binarkę,
        - uruchamia część Pythonową benchmarku,
        - parsuje output,
        - generuje raport w MD.
    - organizuje strukturę katalogów (np. `src_cpp`, `src_py`, `data`, `reports`).
- **Wspólnie**:
    - ustalają format danych wejściowych,
    - ustalają format outputu (tak, aby był łatwy do parsowania),
    - piszą sekcję „Wnioski” w raporcie.

Minimalny zakres (wersja podstawowa)

- Jeden typ zadania: np. **sortowanie tablicy** liczb całkowitych.
- Kilka rozmiarów danych (np. 10 000, 50 000, 100 000).
- Jedna konfiguracja kompilatora (np. tylko `/O2` lub `-O2`).
- Kilka powtórzeń pomiaru dla każdego przypadku (np. 5–10).
- Automatyczne wygenerowanie raportu z tabelą porównawczą C++ vs Python.

Rozszerzenia (dla chętnych/ambitnych)

Nie są obowiązkowe, ale podnoszą ocenę:

- Porównanie **dwóch algorytmów** w każdym języku (np. sortowanie bąbelkowe vs sortowanie biblioteczne).
- Porównanie **różnych flag kompilacji** (np. `-O0` vs `-O2`) i omówienie wpływu na wynik.
- Dodanie **wykresów** (czas od rozmiaru danych) generowanych z Pythona i dołączonych do raportu.
- Zapis wyników pośrednich w osobnym pliku CSV (do późniejszej analizy).
- Krótka analiza błędów pomiaru (wariancja, min, max, „zimna” i „ciepła” pamięć cache, kilka uruchomień).

Dostarczenie wyników do sprawdzenia:

- Kody źródłowe wraz z przykładowymi raportami MD (zarejestrowanymi na kilku różnych komputerach) mają się znajdować w spakowanym pliku zip
- Nazwa katalogu projektu oraz jednocześnie nazwa pliku zim to nazwa sekcji i słowa "Benchmark", np. `Brawo Benchmark`, `Echo Benchmark` etc.

Kryteria oceny

Ocena z tego mini projektu będzie posiadała wagę = 2

Przykładowe kryteria:

- Poprawność techniczna:
    - czy wszystko da się uruchomić jednym poleceniem / jednym skryptem,
    - czy kompilacja C++ działa z linii komend,
    - czy dane wejściowe są współdzielone między C++ i Pythonem.
- Jakość pomiaru:
    - czy wykonywane są wielokrotne powtórzenia,
    - czy format wyników jest spójny i parsowalny.
- Automatyzacja:
    - minimalna liczba ręcznych kroków (uruchamiam skrypt, reszta dzieje się sama).
- Raport:
    - czy jest kompletny (cel, środowisko, parametry, wyniki, wnioski),
    - czy użyto poprawnie składni Markdown (nagłówki, listy, tabele, formatowanie kodu).
- Współpraca:
    - podział zadań,
    - czy widać wkład obu osób.

Pluton Benchmarkowy – skład sekcji

Sekcja Alfa

- **DĄBROWSKI Adam**
- **LEWANDOWSKI Adam**

Sekcja Bravo

- **MIŁUCH Hubert**
- **WIERZBANOWSKA Emilia**

Sekcja Charlie

- **MARKOWSKI Adrian**
- **ŻURAWICZ Franciszek**

Sekcja Delta

- **KACZOROWSKA Magda**
- **WALCZAK Filip**

Sekcja Echo

- **BARAŃSKI Tytus**
- **LISIECKI Gabriel**

Sekcja Foxtrot

- **DĄBROWSKI Grzegorz**
- **NAWROCKI Paweł**

Sekcja Golf

- **GRUBBA Natalia**
- **ZAJĄC Adrian**

Sekcja Hotel

- **LATAŁA Franciszek**
- **STOLC Karol**

Sekcja India

- **BACZYŃSKI Jakub**
- **RADLAK Maciej**

Sekcja Juliett

- **KAMIŃSKI Maciej**
- **SOKOŁOWSKI Karol**

Sekcja Kilo

- **MASHOVETS Mykola**
- **ŻBIKOWSKI Nataniel**

Sekcja Lima

- **MIERZWA Daniel**
- **SCHWARZ Leon**

Sekcja Mike

- **ROSPONDEK Szymon**
- **ŚLUGAJSKA Alicja**

Sekcja November

- **GRZEGORCZYK Maciej**
- **KACZMARCZYK Patryk**

Sekcja Oscar

- **MICUN Adam**
- **SKIERKA Tymoteusz**

Sekcja Papa

- **PALIŃSKA Zofia**
- **RAMPALSKI Jan**