**Uniwersytet Warszawski**
Wydział Matematyki, Informatyki i Mechaniki

**Marcin Potkański**
Nr albumu: 209402

# Analiza porównawcza bajtkodów i maszyn wirtualnych świata języków obiektowych: Java vs Python.

**Praca magisterska**
**na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem
**dr. Jacka Chrząszcza**
Instytut Informatyki

Warszawa, Grudzień 2025

## Streszczenie

W pracy porównano batjkody Javy i Pythona. Omówiono również maszyny wirtualne, na których wykonywany jest kod pośredni tych języków. Szacuje się czas wykonania bajtkodów tych samych programów napisanych w badanych językach programowania na maszynach wirtualnych przy różnych ustawieniach optymalizacji. Programy te pochodzą ze strony geeksforgeeks [6] a także CLBG [7]. Wskazano co sprawia, że Java jest szybsza od Pythona. Omówiono JIT wykorzystywany przez JVM jak i jego odpowiednik w Pythonie, który pojawił się dopiero w wersji 3.13. Wspomniano o optymalizacjach. W pracy skupiono się na najpopularniejszej (napisanej w C) implementacji języka Python jaką jest CPython.



## Słowa kluczowe

Java, Python, JVM, PVM, maszyna wirtualna, bajtkod


## Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

## Klasyfikacja tematyczna

D. Software

## Tytuł pracy w języku angielskim

Comparative analysis of bytecodes and virtual machines in the world of object-oriented languages: Java vs Python.

# Spis treści


Java
1
python™

# Rozdział 1

# Wprowadzenie

Java i Python to bardzo popularne obecnie języki programowania. Na rynku pracy jest duże zapotrzebowanie na programistów zarówno jednego, jak i drugiego języka. W niniejszej pracy porównano bajtkody i maszyny wirtualne owych języków w celu wskazania szczegółów, którymi się różnią. Porówano je też pod względem zużycia pamięci.

Praca składa się z dziesięciu rozdziałów. W rozdziale 2 przypomniano podstawowe pojęcia związane z językami Java i Python takie jak bajtkod czy maszyny wirtualne. Omówienie instrukcji bajtkodów zawarto w rozdziale 3. W rozdziale 4 dokonano wstępnego porównania JVM i PVM. W rozdziale 5 omówiono architekturę maszyn wirtualnych Javy i Pythona. W rozdziale 6 porównano JIT Javy i Pythona. W rozdziale 7 przedstawiono zarządzanie pamięcią. Rozdział 8 zawiera porównanie czasu działania prostych fragmentów kodu. W rozdziale 9 porównano czas działania większych programów. Rozdział 10 zawiera podsumowanie.

## 1.1. Historia Javy

James Gosling, Mike Sheridan i Patrick Naughton zainicjowali projekt języka Java w czerwcu 1991 r. Java została pierwotnie zaprojektowana dla telewizji interaktywnej, ale była wówczas zbyt zaawansowana dla branży cyfrowej telewizji kablowej. Gosling zaprojektował Javę ze składnią w stylu C/C++, którą programiści systemów i aplikacji mogliby uznać za znajomą.

Firma Sun Microsystems wydała pierwszą publiczną implementację jako Java 1.0 w 1996 roku. Zapewniała funkcjonalność WORA (ang. Write Once, Run Anywhere), udostępniała darmowe środowisko uruchomieniowe na popularnych platformach. Dość bezpieczna i wyposażona w konfigurowalne zabezpieczenia pozwalała na ograniczenia dostępu do sieci i plików. Popularne przeglądarki internetowe wkrótce włączyły możliwość uruchamiania apletów Javy na stronach internetowych, a Java szybko stała się popularna. Kompilator Java 1.0 został ponownie napisany w Javie przez Arthura van Hoffa w celu ścisłej zgodności ze specyfikacją języka Java 1.0.

Wraz z pojawieniem się desktopowej wersji Java 2 (wydanej początkowo jako J2SE 1.2 w grudniu 1998 - 1999, przemianowanej w 2004 na Java SE), nowe wersje miały wiele konfiguracji zbudowanych dla różnych typów platform. J2EE zawierała technologie i interfejsy API dla aplikacji korporacyjnych, które zwykle działają w środowiskach serwerowych, natomiast J2ME zawierało interfejsy API zoptymalizowane pod kątem aplikacji mobilnych.

| Rok | Wersja | Zmiany |
|---|---|---|
| 1995 | | Java przeznaczona dla telewizji interaktywnej |
| 1995 | | The Green Team inicjuje rozwój Javy |
| 1995 | | „Oak” - początkowa nazwa Javy |
| 1995 | | Zmieniono nazwę na „Java” ze względu na problemy ze znakiem towarowym |
| 1996 | JDK 1.0 | Pierwsza oficjalna wersja, podstawowe funkcje języka (OOP), JVM, podstawowe biblioteki |
| 1997 | JDK 1.1 | Istotne usprawnienia, włączając inner classes, JavaBeans, JDBC, and RMI. |
| 1998 | J2SE 1.2 | Przemianowano na "Java 2 Platform, Standard Edition"(J2SE). Dodano Swing GUI toolkit, Collections framework, JIT compiler. |
| 2000 | J2SE 1.3 | Poprawiona wydajność, Java Naming i Directory Interface (JNDI). |
| 2002 | J2SE 1.4 | Dalsza poprawa wydajności. Wprowadzoano regular expressions, XML parsing, Non-blocking I/O, Logging API. |
| 2004 | Java SE 5 | Przemianowano na „Java Platform, Standard Edition” (Java SE) Główna aktualizacja: Generics, metadata annotations, ulepszona for loop. |
| 2006 | Java SE 6 | Skupienie na poprawie wydajności, scripting support (via JSR 223). |
| 2011 | Java SE 7 | Dodano try-with-resources, NIO 2.0, i the diamond operator. |
| 2014 | Java SE 8 | Lambda expressions, Stream API, new Date/Time API, default methods. |
| 2017 | Java SE 9 | Wprowadzono modularność (Project Jigsaw), JShell, HTTP/2 support. |
| 2018 | Java SE 10 | Local-variable type inference (var), JEP 286. |
| 2018 | Java SE 11 | LTS release, new HTTP client, usunięto moduły Java EE. |
| 2019 | Java SE 12 | Shenandoah garbage collector, JVM constants API. |
| 2019 | Java SE 13 | Text Blocks (preview), dynamic CDS archives. |
| 2020 | Java SE 14 | Records (preview), helpful NPE, pattern matching dla instanceof. |
| 2020 | Java SE 15 | Sealed classes (preview), Hidden classes, Foreign Function API (incubator). |
| 2021 | Java SE 16 | Records (finalized), Pattern Matching for switch (preview). |
| 2021 | Java SE 17 | Wersja LTS, Sealed classes (finalized), silna enkapsulacja wnętrza JDK. |
| 2022 | Java SE 18 | Simple Web Server, vector API (incubator). |
| 2022 | Java SE 19 | Virtual threads (preview), Foreign function & memory API (preview). |
| 2023 | Java SE 20 | Record patterns (preview), Virtual threads (preview). |
| 2023 | Java SE 21 | Virtual threads (finalized), sequenced collections, structured concurrency (preview). |
| 2024 | Java SE 22 | Usprawnienia do projektu Loom, nowe uaktualnienia API. |
| 2024 | Java SE 23 | Usprawniony pattern matching, Foreign Function & Memory API (finalized), poprawiona skalowalność. |

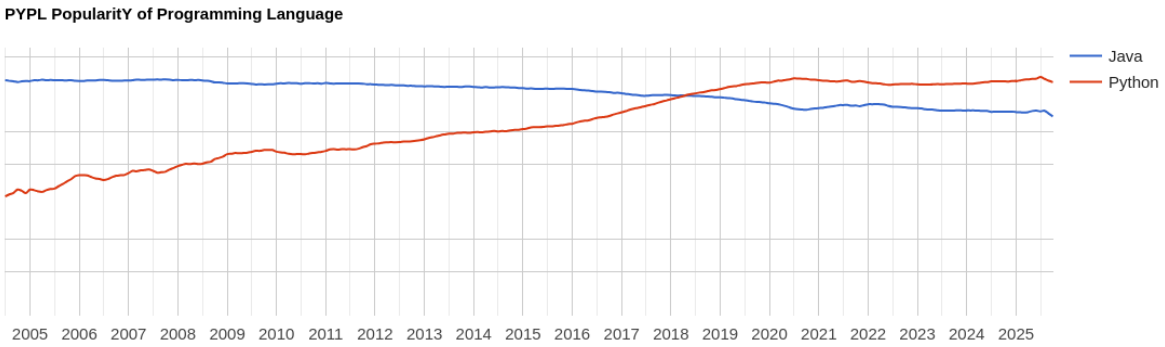
## 1.2. Historia Pythona

Pythona stworzył we wczesnych latach 90. Guido van Rossum – jako następcę języka ABC, stworzonego w Centrum voor Wiskunde en Informatica (CWI – Centrum Matematyki i Informatyki w Amsterdamie). Van Rossum jest głównym twórcą Pythona, choć spory wkład w jego rozwój pochodzi od innych osób. Z racji kluczowej roli, jaką van Rossum pełnił przy podejmowaniu ważnych decyzji projektowych, często określano go przydomkiem „Benevolent Dictator for Life" (BDFL).

Nazwa języka nie pochodzi od zwierzęcia, lecz od serialu komediowego emitowanego w latach siedemdziesiątych przez BBC – „Monty Python's Flying Circus" (Latający cyrk Monty Pythona). Projektant, będąc fanem serialu i poszukując nazwy krótkiej, unikalnej i nieco tajemniczej, uznał tę za świetną.

| Rok | Wersja | Zmiany |
|---|---|---|
| 1991 | Python 0.9.0 | Wersja początkowa z podstawowymi typami danych jak list, dict, string |
| 1994 | Python 1.0 | Wprowadzono lambda, map, filter, reduce |
| 2000 | Python 2.0 | Dodano list comprehensions, wsparcie dla Unicode, garbage collection |
| 2008 | Python 3.0 | Gruntowna przebudowa, lepsze wsparcie dla Unicode, bardziej spójna składnia |
| 2018 | Python 3.7 | Data classes, async/await, context variables |
| 2020 | Python 3.8 | Walrus operator, positional-only parameters, ulepszenia f-stringów |
| 2021 | Python 3.9 | Type hinting generics, nowy parser, moduł zoneinfo |
| 2022 | Python 3.10 | Strukturalny pattern matching, precyzyjne lokalizowanie błędów |
| 2023 | Python 3.11 | Poprawa wydajności, exception groups |
| 2025 | Python 3.13 | Eksperymentalny kompilator JIT (Just-In-Time), możliwość wyłączenia Global Interpreter Lock (GIL), nowy i ulepszony interaktywny interpreter (REPL), inkrementalny zbieracz śmieci |
| 2025 | Python 3.14 | Nowy interpreter, który można włączyć opcjonalnie (opt-in) i który działa od 3 do 5% szybciej niż domyślny |

## 1.3. Popularność i znaczenie




Rysunek 1.1: źródło: https://pypl.github.io/PYPL.html

Na wykresie uwidoczniono łączną liczbę wyszukiwań tutoriali danego języka w Google. Patrząc na załączoną grafikę widać, że na przestrzeni ostatnich dwóch dekad Java traciła na popularności a Python zyskiwał. Gdzieś w połowie 2018 roku, Python prześcignął w popularności Javę.

**Worldwide**, Oct 2025 :

| Rank | Change | Language | Share | 1-year trend |
|---|---|---|---|---|
| 1 | | Python | 28.97 % | -0.5 % |
| 2 | | Java | 13.94 % | -1.5 % |
| 3 | ↑ | C/C++ | 10.54 % | +3.6 % |
| 4 | ↑↑↑↑↑↑↑ | Objective-C | 7.05 % | +4.5 % |
| 5 | ↓↓ | JavaScript | 6.33 % | -1.7 % |

Rysunek 1.2: źródło: https://pypl.github.io/PYPL.html

Na tym wykresie wskazano procentowy udział danego języka we wszystkich wyszukiwaniach tutoriali. Jak widać na załączonej grafice, Python i Java zajmują dwa pierwsze miejsca na liście popularności języków. Sprawia to, że obydwa te języki mają obecnie bardzo duże znaczenie w programowaniu. Ich łączny udział w rynku wynosi około 43% przy czym Python jest 2 razy popularniejszy od Javy (stan na październik 2025).


Java
2
python™

# Rozdział 2

# Podstawowe pojęcia

## 2.1. Maszyna wirtualna

**Maszyna wirtualna** (ang. *virtual machine*, **VM**) – przyjęta nazwa środowiska uruchomieniowego programów.

Maszyna wirtualna kontroluje i obsługuje wszystkie odwołania uruchamianego programu bezpośrednio do sprzętu lub systemu operacyjnego. Dzięki temu program uruchomiony na maszynie wirtualnej działa jak na rzeczywistym sprzęcie.

Wykonywanym programem może być pojedyncza aplikacja, jak i cały system operacyjny lub nawet kolejna maszyna wirtualna. Odizolowanie ich przez VM od maszyny fizycznej odróżnia ją od klasycznego systemu operacyjnego.

### 2.1.1. Rodzaje maszyn wirtualnych

Maszyny wirtualne to m.in.:

- interpretery, szczególnie interpretery kodu bajtowego
- kompilatory JIT
- emulatory rzeczywiście istniejącego sprzętu, np. emulatory konsol.

Różnice między poszczególnymi typami takich maszyn są płynne. Np. wirtualna maszyna Javy jest powszechnie znana jako samodzielny interpreter, ale ponieważ istniały komputery, które potrafiły wykonywać programy w kodzie bajtowym Javy bezpośrednio, można ją także traktować jako emulator tych maszyn. Ponadto kompilator JIT również jest rozwiązaniem wykorzystywanym podczas interpretacji kodu bajtowego Javy.

## 2.2. Bajtkod

**Kod bajtowy** (ang. *bytecode*) – nazwa reprezentacji kodu używanej przez maszyny wirtualne oraz przez niektóre kompilatory. Kod składa się z ciągu instrukcji (których kody operacji mają zwykle długość jednego bajtu), które nie odpowiadają bezpośrednio instrukcjom procesora i mogą zawierać instrukcje wysokiego poziomu. W przeciwieństwie do kodu źródłowego wymagają jednak analizy tylko poszczególnych operacji.


Java
3
python™

# Rozdział 3

# Omówienie instrukcji bajtkodów

## 3.1. Instrukcje bajtkodu Javy

Każdy bajtkod składa się z jednego bajtu reprezentującego kod operacji (**opcode**) oraz zera lub większej liczby bajtów reprezentujących **operandy**.

Spośród 256 możliwych kodów operacji, w 2015 r. 202 są w użyciu (∼79%), 51 jest zarezerwowanych do wykorzystania w przyszłości (∼20%), a 3 instrukcje (∼1%) są trwale zarezerwowane do wykorzystania przez implementacje JVM. Dwie z nich (`impdep1` i `impdep2`) mają na celu zapewnienie pułapek dla oprogramowania i sprzętu specyficznego dla implementacji. Trzecia jest używana przez debugery do implementowania punktów przerwania.

### 3.1.1. Instrukcje można podzielić na następujące grupy:

- ładujące i zapisujące (np. `aload_0`, `istore`),
- arytmetyczne i logiczne (np. `ladd`, `fcmpl`),
- zmieniające typ (np. `i2b`, `d2i`),
- przenoszące sterowanie (np. `ifeq`, `goto`),
- tworzące obiekt i manipulujące nim (np. `new`, `putfield`),
- wywołujące metodę i wracające z niej (np. `invokespecial`, `areturn`).

Są również instrukcje służące do wyspecjalizowanych czynności, takich jak zwracanie wyjątków, synchronizacja itd. Wiele instrukcji ma też prefiksy lub sufiksy związane z typami zmiennych, na których pracują. Oto ich lista:

| Prefiks/ Sufiks | Typ w języku Java | Opis | Przyjmowane wartości |
|---|---|---|---|
| i | int | 32-bitowy typ całkowity ze znakiem | $-2^{31}..2^{31}$-1 |
| l | long | 64-bitowy typ całkowity ze znakiem | $-2^{63}..2^{63}$-1 |
| s | short | 16-bitowy typ całkowity ze znakiem | $-2^{15}..2^{15}$-1 |
| b | byte | 8-bitowy typ całkowity ze znakiem | -128..127 |
| c | char | 16-bitowy typ całkowity bez znaku | 0..65535 |
| f | float | 32-bitowy typ zmiennoprzecinkowy | N/A |
| d | double | 64-bitowy typ zmiennoprzecinkowy | N/A |
| z | boolean | 1-bitowy typ Boolowski | 0 lub 1 |
| a | Object i pochodne | Referencja do instancji klasy Object | N/A |

## 3.2. Struktura bajtkodu Javy

Każda instrukcja ma następującą postać:

```
<opcode> [ <operand1> [ <operand2>... ]]
```

W przypadku korzystania z narzędzia `javap` otrzymuje się plik z wpisami o następującej postaci:

```
<index> <opcode> [ <operand1> [ <operand2>... ]] [<comment>]
```

Przykład wywołania narzędzia `javap`:

```
class Simple {
  public static void main(String argv[]) {
    for (int i=1;i<=10;i++){
      System.out.println(i);
    }
  }
}
$ javac Simple.java
$ javap -c Simple
```

```
Method Simple()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 iconst_1
   1 istore_1
   2 goto 15
   5 getstatic #2 <Field java.io.PrintStream out>
   8 iload_1
   9 invokevirtual #3 <Method void println(int)>
   12 iinc 1 1
   15 iload_1
   16 bipush 10
   18 if_icmple 5
   21 return
```

Pole <**index**> jest indeksem kodu operacji instrukcji w tablicy zawierającej bajty kodu wirtualnej maszyny Java dla tej metody. Alternatywnie, <**indeks**> można traktować jako

offset bajtowy od początku metody. Pole <**opcode**> jest mnemonikiem kodu operacji instrukcji, a zero lub więcej <**operandN**> to operandy instrukcji. Opcjonalny <**komentarz**> podaje się w składni komentarza na końcu wiersza:

```
8   bipush 100      // Push int constant 100
```

Pole <**indeks**> poprzedzające każdą instrukcję może być celem instrukcji przekazania sterowania. Na przykład instrukcja `goto 8` przekazuje sterowanie do instrukcji o indeksie 8. Rzeczywiste argumenty instrukcji przesyłania sterowania wirtualnej maszyny Java są przesunięte w stosunku do adresów rozkazów (<**opcode**>) tych instrukcji; te operandy są wyświetlane przez `javap`, ponieważ łatwiej jest odczytać przesunięcia w ich metodach.

Przed operandem reprezentującym indeks w puli stałych czasu wykonywania (**run-time constant pool**) stawiamy znak hash (#) i po instrukcji dodajemy komentarz identyfikujący element z puli stałych, do którego się odwołujemy, na przykład:

```
10  ldc #1          // Push float constant 100.0
```

albo:

```
9   invokevirtual #4    // Method Example.addTwo(II)I
```

## 3.3. Instrukcje bajtkodu Pythona

Przedstawimy instrukcje bajtkodu Pythona na przykładzie oficjalnej i najpopularniejszej implementacji tego języka CPython, napisanej w języku C. Dokładny zestaw instrukcji zależy od wesji CPythona. Począwszy od CPythona 3.6 wszystkie instrukcje bajtkodu zajmują dokładnie 2 bajty i mają format: `<INSTRUCTION> <ARGUMENT>` (po 1 bajcie). Istnieje pewien magiczny numer który określa, czy instrukcja wymaga argumentu. Jeśli liczbowa reprezentacja instrukcji jest mniejsza od tego numeru to nie ma ona argumentu. W CPython'ie 3.6 tym magicznym numerem jest 90 np. `BINARY_POWER` nie ma argumentu, jako że jego bajtowa reprezentacja wynosi 19. Jeśli instrukcja nie potrzebuje argumentu, to ten bajt argumentu jest po prostu ustawiony na 0.

Format: `<INSTRUCTION> <ARGUMENT>` obsługuje 256 różnych kodów operacji (opcode), co jest wystarczające. Jednak ogranicza to argument operacji (oparg) do wartości 8-bitowej, co już nie jest wystarczające. Aby obejść to ograniczenie, używany jest kod operacji EXTENDED_ARG, który pozwala poprzedzić dowolną instrukcję jednym lub większą liczbą dodatkowych bajtów danych.

Na przykład poniższa sekwencja jednostek kodu ustawi opcode na LOAD_CONST, a oparg na 65538 (czyli 0x1_00_02 w zapisie szesnastkowym):

```
EXTENDED_ARG  1
EXTENDED_ARG  0
LOAD_CONST    2
```

Kompilator powinien ograniczać się do maksymalnie trzech prefiksów EXTENDED_ARG, aby wynikowy oparg mieścił się w 32 bitach, jednak interpreter tego nie sprawdza.

Seria jednostek kodu rozpoczynająca się od zera do trzech kodów EXTENDED_ARG, a kończąca się głównym kodem operacji, nazywana jest kompletną instrukcją, aby odróżnić ją od pojedynczej jednostki kodu, która zawsze ma dwa bajty.

### 3.3.1. Instrukcje CPythona można podzielić na następujące grupy:

- Operacje unarne, które pobierają element ze szczytu stosu, dokonują operację a następnie umieszczają wynik na szyczcie stosu np. `UNARY_POSITIVE` – znak plus (+x), `UNARY_NOT` – negacja logiczna (not x).

- Operacje binarne, które, pobierają dwa elementy ze stosu: `TOS1` i `TOS`, wykonują operację (np. dodawanie) i wynik odkładają na stosie np. `BINARY_ADD` (x + y), `BINARY_MULTIPLY` (x * y) .

- Operacje w miejscu, które działają podobnie do operacji binarnych, ale próbują wykonać operację „w miejscu" (np. zamiast x + y robią x += y, jeśli x na to pozwala). Wynik operacji może zostać zapisany bezpośrednio w pierwszym (starszym) elemencie na stosie (TOS1), jeśli ten element na to pozwala. Przykłady takich operacji to `INPLACE_ADD` (x += y), `INPLACE_MULTIPLY` (x *= y).

- Operacje na listach / sekwencjach (np. slicing), które manipulują wycinkami (slice'ami) list lub innych sekwencji. W niektórych przypadkach wymagają wielu elementów na stosie (np. start, end, container). Przykłady: `BUILD_SLICE` – tworzy obiekt slice(start, stop), `STORE_SUBSCR` – obj[index] = value.

- Rozmaite inne operacje, niekiedy wymagające argumentów (np. `LOAD_CONST`, `POP_TOP`).

Warto jeszcze wspomnieć o adaptacyjnym bajtkodzie (ang. *adaptive bytecode*), mechaniźmie wprowadzonym w CPythonie 3.11, który polega na tym, że interpreter dynamicznie podmienia pewne instrukcje na szybsze, wyspecjalizowane wersje, dostosowane do danych, które faktycznie występują w czasie wykonania programu. Zapewnia on znaczny wzrost wydajności (10–60%).

### 3.3.2. Działanie adaptacyjnego bajtkodu w CPythonie 3.11

1. **Kompilacja:** Kod źródłowy Pythona jest tłumaczony na bajtkod. Interpreter zastępuje wybrane instrukcje ich wersjami adaptacyjnymi, np. `LOAD_ATTR → LOAD_ATTR_ADAPTIVE`.

2. **Obserwacja:** Adaptacyjny opcode wykonuje się normalnie, ale monitoruje typy danych i zapisuje informacje w *inline cache*.

3. **Specjalizacja:** Po wykryciu powtarzalnego wzorca (np. ten sam typ obiektu), interpreter zastępuje instrukcję szybszą wersją, np. `LOAD_ATTR_ADAPTIVE → LOAD_ATTR_INSTANCE_VALUE`.

4. **Wykonanie zoptymalizowane:** Specjalizowany opcode pomija kosztowne sprawdzenia i działa szybciej.

5. **De-specjalizacja:** Jeśli warunki się zmienią (np. inny typ), interpreter przywraca wersję adaptacyjną i cykl zaczyna się od nowa.

## 3.4. Struktura bajtkodu Pythona

Typowa linia wyjścia `dis.dis()` wygląda na przykład tak:

```
  2           0 LOAD_FAST                0 (x)
              2 RETURN_VALUE
```

Można ją rozłożyć na następujące kolumny:

| Linia źródłowa | Offset bajtkodu | Instrukcja | Argument | Wartość argumentu |
|---|---|---|---|---|
| 2 | 0 | LOAD_FAST | 0 | x |
| 2 | 2 | RETURN_VALUE | - | - |

### 3.4.1. Wyjaśnienie kolumn

- **Linia źródłowa:** numer wiersza w kodzie Python, z którego pochodzi instrukcja.
- **Offset bajtkodu:** pozycja instrukcji w ciągu bajtkodu (`co_code`).
- **Instrukcja:** nazwa operacji bajtkodu, np. `LOAD_FAST`, `RETURN_VALUE`.
- **Argument:** indeks w tabeli zmiennych lokalnych (`co_varnames`), stałych (`co_consts`) itp.
- **Wartość argumentu:** nazwa zmiennej lub wartość stałej, jeśli jest znana.

## 3.5. Porównanie bajtkodów

Bajtkod Javy przechowywany jest w pliku o rozszerzeniu .class, Python natomiast umieszcza bajtkod w pliku .pyc. Java cechuje się wysoką stabilnością między wersjami — format bajtkodu zmienia się bardzo rzadko i w sposób kompatybilny. W przypadku Pythona jest ona niska — format bajtkodu zmienia się często, nawet między wersjami minor. Jeśli chodzi o wsteczną kompatybilność to w przypadku Javy jest ona wysoka - pliki .class wygenerowane w starszej wersji Javy działają na nowszej JVM bez zmian. Kod skompilowany w Javie 8 działa na JVM Javy 17 bez rekompilacji. W przypadku Pythona wsteczna kompatybilność jest niska — pliki .pyc z jednej wersji Pythona zwykle nie działają poprawnie w innej wersji.

Maszyna wirtualna Javy obsługuje ponad 200 różnych bajtkodów, podczas gdy PVM rozróżnia 118 bajtkodów (CPython 3.6). Implementacje bajtkodów zarówno jednego jak i drugiego języka wykorzystują maszyny stosowe do wykonywania obliczeń. W przypadku Javy nazywa się go **operand stack**. W Pythonie jest to **value stack** (zwany też **data stack** lub **evaluation stack**).

Już na poziomie bajtkodów można dostrzec istotne różnice między statyczną naturą Javy a dynamiczną naturą Pythona.

Niech za przykład posłużą instrukcje `iadd` Javy i `BINARY_ADD` Pythona.

W przypadku Javy zasadne wydaje się rozpatrzenie kodu emitowanego przez tzw. CppInterpreter i TemplateInterpreter. CppInterpreter jest niezależny od platformy, natomiast TemplateInterpreter używa kodu specyficznego dla platformy.

Poniżej znajduje się kod implementacji operacji `iadd` dla CppInterpretera (plik *bytecodeInterpreter.cpp*).

```
#define OPC_INT_BINARY(opcname, opname, test)
      CASE(_i##opcname):
          if (test && (STACK_INT(-1) == 0)) {
              VM_JAVA_ERROR(vmSymbols::java_lang_ArithmeticException()
   ,
                            "/ by zero");
          }
          SET_STACK_INT(VMint##opname(STACK_INT(-2),
```

```
                                        STACK_INT(-1)),
                                        -2);
            UPDATE_PC_AND_TOS_AND_CONTINUE(1, -1);
            // to samo dla long zamiast int

    OPC_INT_BINARY(add, Add, 0);
    // inne operatory
```

W pliku *bytecodeInterpreter_zero.inline.hpp* znajduje się kod operacji dodawania.

```
inline jint BytecodeInterpreter::VMintAdd(jint op1, jint op2) {
  return op1 + op2;
}
```

Poniżej znajduje się z kolei kod implementacji operacji `iadd` dla TemplateInterpretera (plik *templateTable_x86.cpp*).

```
void TemplateTable::iop2(Operation op) {
  transition(itos, itos);
  switch (op) {
  case add  :                    __ pop_i(rdx); __ addl (rax, rdx);
   break;
  case sub  : __ movl(rdx, rax); __ pop_i(rax); __ subl (rax, rdx);
   break;
  case mul  :                    __ pop_i(rdx); __ imull(rax, rdx);
   break;
  case _and :                    __ pop_i(rdx); __ andl (rax, rdx);
   break;
  case _or  :                    __ pop_i(rdx); __ orl  (rax, rdx);
   break;
  case _xor :                    __ pop_i(rdx); __ xorl (rax, rdx);
   break;
  case shl  : __ movl(rcx, rax); __ pop_i(rax); __ shll (rax);
   break;
  case shr  : __ movl(rcx, rax); __ pop_i(rax); __ sarl (rax);
   break;
  case ushr : __ movl(rcx, rax); __ pop_i(rax); __ shrl (rax);
   break;
  default   : ShouldNotReachHere();
  }
}
```

Na tych przykładach widać że maszyna wirtualna Javy wymaga by liczby, na których operuje były 32-bitowymi liczbami całkowitymi a maszyna wirtualna Pythona wywołuje zaś dynamicznie odpowiedni kod aby dodać 2 obiekty na stosie bazując na ich typach czasu wykonania. Pokazano to na poniższym kodzie:

```
TARGET(BINARY_ADD) {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;

    if (PyUnicode_CheckExact(left) &&
            PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(left, right, f, next_instr);
    }
```

```
    else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
```

Python sprawdza czy lewy i prawy operand są instancjami Unicode np. stringami. Dokonuje tego sprawdzając typ ich obiektu. Jeśli obydwa operandy są stringami to je łączy. W przeciwnym przypadku wywoływane jest `PyNumber_Add()`.

```
PyObject *
PyNumber_Add(PyObject *v, PyObject *w)
{
    // NB_SLOT(nb_add) expands to "offsetof(PyNumberMethods, nb_add)"
    PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));
    if (result == Py_NotImplemented) {
        PySequenceMethods *m = Py_TYPE(v)->tp_as_sequence;
        Py_DECREF(result);
        if (m && m->sq_concat) {
            return (*m->sq_concat)(v, w);
        }
        result = binop_type_error(v, w, "+");
    }
    return result;
}
```

`PyNumberAdd()` najpierw próbuje przepowadzić operację dodawania na operandach `v` i `w` (dwa wskaźniki do `PyObject`) wywołując `binary_op1(v, w, NB_SLOT(nb_add))`. Jeśli rezultatem wywolania jest `Py_NotImplemented`, próbuje połączyć operandy jako sekwencję.

```
static PyObject *
binary_op1(PyObject *v, PyObject *w, const int op_slot)
{
    PyObject *x;
    binaryfunc slotv = NULL;
    binaryfunc slotw = NULL;

    if (Py_TYPE(v)->tp_as_number != NULL)
        slotv = NB_BINOP(Py_TYPE(v)->tp_as_number, op_slot);
    if (!Py_IS_TYPE(w, Py_TYPE(v)) &&
        Py_TYPE(w)->tp_as_number != NULL) {
        slotw = NB_BINOP(Py_TYPE(w)->tp_as_number, op_slot);
        if (slotw == slotv)
            slotw = NULL;
    }
    if (slotv) {
        if (slotw && PyType_IsSubtype(Py_TYPE(w), Py_TYPE(v))) {
            x = slotw(v, w);
            if (x != Py_NotImplemented)
                return x;
            Py_DECREF(x); /* can't do it */
```

```
22              slotw = NULL;
23          }
24          x = slotv(v, w);
25          if (x != Py_NotImplemented)
26              return x;
27          Py_DECREF(x); /* can't do it */
28      }
29      if (slotw) {
30          x = slotw(v, w);
31          if (x != Py_NotImplemented)
32              return x;
33          Py_DECREF(x); /* can't do it */
34      }
35      Py_RETURN_NOTIMPLEMENTED;
36  }
```

## Wyjaśnienie działania funkcji `binary_op1` przy dodawaniu

1. **Funkcja `binary_op1` nie dodaje bezpośrednio** wartości. Jej zadaniem jest wybranie odpowiedniej funkcji operatora binarnego dla danych typów.

2. **Pobranie slotów binarnych:**

   - `slotv` – funkcja binarna przypisana do obiektu $v$.
   - `slotw` – funkcja binarna przypisana do obiektu $w$ (jeśli typy są różne).

3. **Mechanizm wyboru funkcji:**

   - Makro `NB_BINOP(Py_TYPE(v)->tp_as_number, op_slot)` zwraca wskaźnik do właściwej funkcji, np. `nb_add` dla dodawania.

4. **Pierwsza próba – podtyp:**

   - Warunek: `slotv` nie może być NULL.
   - Jeśli `w` jest podtypem `v` i `slotw` istnieje, wywołuje się `slotw(v, w)`.
   - Dla typów `int` `slotw(v, w)` odpowiada funkcji `int_add(v, w)` – faktycznej implementacji dodawania liczb całkowitych.
   - Jeśli wynik nie jest `Py_NotImplemented`, jest zwracany od razu.
   - W przeciwnym razie przechodzi się do wywołania `slotv`.

5. **Wywołanie slotu typu bazowego:**

   - Wywołuje się `slotv(v, w)`, czyli dla typu `int` również `int_add(v, w)`.
   - Jeśli wynik nie jest `Py_NotImplemented`, zwracany jest jako wynik operacji.

6. **Drugie wywołanie slotu w (jeśli wcześniejsze nie powiodło się):**

   - Jeśli wcześniejsze wywołania nie zwróciły wyniku i `slotw` istnieje, wywołuje się `slotw(v, w)` po raz drugi.
   - Jeśli wynik nie jest `Py_NotImplemented`, zwracany jest jako wynik operacji.

7. **Brak implementacji:**
    - Jeśli żaden slot nie zwrócił wyniku, funkcja zwraca `Py_RETURN_NOTIMPLEMENTED`.
8. **Podsumowanie:**
    - `binary_op1` to *dispatcher* – wybiera odpowiednią funkcję binarną w zależności od typów obiektów.
    - Faktyczne dodawanie odbywa się wewnątrz funkcji `nb_add` (lub analogicznej dla innych typów), a nie w `binary_op1`.


Java
4
python™

# Rozdział 4

# Wstępne porównanie JVM i PVM

Do określenia maszyn wirtualnych Javy i Pythona stosujemy odpowiednio skróty: JVM (Java Virtual Machine) i PVM (Python Virtual Machine). JVM jest implementacją specyfikacji Java Virtual Machine. PVM jest z kolei używana przez CPythona, standardową implementację Pythona. Maszyny te służą podobnym celom, ale jednocześnie zasadniczo się różnią.

## 4.1. Języki programowania:

- **JVM:** JVM została zaprojektowana specjalnie do uruchamiania kodu bajtowego Javy. Chociaż istnieją inne języki, które można skompilować do kodu bajtowego Javy (np. Kotlin, Scala), JVM obiera za cel przede wszystkim programy Javy.

- **PVM:** PVM jest przeznaczony do uruchamiania kodu bajtowego Pythona wygenerowanego przez CPythona.

## 4.2. Początkowy narzut pamięci

- **JVM:** JVM ma większy początkowy narzut pamięci: ∼30–100 MB. Źródło narzutu: inicjalizacja JVM, klasy, JIT, GC, wątki, Metaspace.

- **PVM:** PVM ma mniejszy początkowy narzut pamięci: ∼10–20 MB. Źródło narzutu: interpreter, tablice wbudowane, cache obiektów, biblioteki stdlib.

## 4.3. Warm-up

- **JVM:** JVM potrzebuje „rozgrzewki", bo interpretuje kod i zbiera profile, zanim JIT go zoptymalizuje. Stąd pierwsze uruchomienia wolne, potem szybsze.

- **PVM:** PVM działa od razu, ale nie przyspiesza w trakcie działania.

## 4.4. Statyczne vs dynamiczne typowanie:

- **JVM:** Java jest językiem statycznie typowanym, co oznacza, że typy zmiennych są znane w czasie kompilacji. JVM wymusza ścisłe sprawdzanie typów w czasie kompilacji.

- **PVM:** Python jest typowany dynamicznie, co oznacza, że typy zmiennych są określane w czasie wykonywania, co sprawia, że kodowanie jest ułatwione i bardziej elastyczne. Może jednak prowadzić do błędów w czasie wykonywania, jeśli problemy z typami nie będą wcześniej odpowiednio rozwiązane.

## 4.5. Zarządzanie pamięcią:

- **JVM:** JVM korzysta z automatycznego zarządzania pamięcią poprzez zbieranie śmieci. Automatycznie zwalnia pamięć dla obiektów, do których nie ma już odniesień.

- **PVM:** PVM wykorzystuje również zbieranie śmieci do zarządzania pamięcią. Zarządzanie pamięcią w Pythonie opiera się na zliczaniu odwołań, które automatycznie zwalnia pamięć, gdy liczba odwołań do obiektu spadnie do zera. Zbieracz śmieci posiada dodatkowe algorytmy wykrywające cykle i jeśli takowe wykryje – to je przerywa. To natomiast powoduje spadek liczby referencji a w konsekwencji sprzątnięcie obiektu.

## 4.6. Obsługa wielowątkowości

- **JVM:** Jest w pełni wspierana, z natywnymi wątkami.

- **PVM:** Ograniczona przez (Global Interpreter Lock - globalna blokada, która ogranicza wykonywanie bajtkodu Pythona do jednego wątku naraz, nawet na komputerach wielordzeniowych.) w CPython aczkolwiek w nowych wersjach (Python 3.13) można go wyłączyć.

## 4.7. Niezależność od platormy:

- **JVM:** Maksyma Java „Write Once, Run Anywhere" jest podstawową zasadą dla języka Java, a kod bajtowy jest niezależny od platformy i jeśli na platformie istnieje implementacja JVM, można na niej uruchomić kod Java. JVM nie zmienia się co wersję Javy.

- **PVM:** Python ma również na celu niezależność od platformy, ale dostępność PVM dla konkretnej platformy może nie być tak powszechna, jak implementacje JVM. Ponadto PVM i bajtkod zmieniają się co wersję Pythona, i dlatego biblioteki Pythonowe nie są dostarczane w postaci bajtkodu, tylko w postaci źródeł.

## 4.8. Wydajność:

- **JVM:** Programy Java są zazwyczaj znane ze swojej wydajności i efektywności. Kompilacja Just-In-Time (JIT) maszyny JVM może zoptymalizować kod bajtowy by zwiększyć szybkość wykonywania.

- **PVM:**

  Python jest ogólnie uważany za wolniejszy niż Java, szczególnie w przypadku zadań związanych z procesorem, ze względu na jego dynamiczne typowanie i interpretacyjną naturę. W wielu aplikacjach łatwość użycia Pythona i dostępne biblioteki dają przewagę nad wadami wydajnościowymi.

Podsumowując, JVM i PVM to maszyny wirtualne, które mają różne filozofie projektowania i nadają się do różnych przypadków użycia. Koncentracja Java na silnym typowaniu i wydajności sprawia, że jest popularna w aplikacjach korporacyjnych, podczas gdy prostota i wszechstronność Pythona sprawia, że jest ulubionym narzędziem do tworzenia skryptów, tworzenia stron internetowych, analizy danych.


Java
5
python™

# Rozdział 5

# Architektura

## 5.1. Architektura maszyny wirtualnej Javy

Programiści Javy wiedzą, że kod bajtowy będzie wykonywany przez JRE (Java Runtime Environment). JRE to z kolei implementacja wirtualnej maszyny Java (JVM), która analizuje kod bajtowy, interpretuje i wykonuje go.

Java została opracowana w oparciu o koncepcję WORA (Write Once Run Anywhere), która działa na maszynie wirtualnej. Kompilator kompiluje plik Java do pliku .class Javy, następnie ten plik .class jest wprowadzany do maszyny JVM, która ładuje i wykonuje plik klasy.

### Jak działa JVM?

JVM jest podzielona na trzy główne podsystemy:

- Podsystem ładowania klas
- Obszar danych czasu wykonywania
- Silnik wykonawczy

## 5.2. Podsystem ładowania klas

Funkcja dynamicznego ładowania klas w Javie jest obsługiwana przez podsystem ładowania klas, który ładuje, linkuje i inicjuje plik klasy przy pierwszym odwołaniu.

### 5.2.1. Ładowanie

Klasy będą załadowane przez ten komponent. Ładowacz klas startowych, Ładowacz klas rozszerzeń, Ładowacz klas aplikacji to trzy klasy modułu ładowania, które pomogą w osiągnięciu tego celu.

- **Ładowacz klas startowych** – Ładowacz klas startowych to kod maszynowy odpowiedzialny za inicjowanie operacji JVM. W wersjach Java do 8 ładował podstawowe pliki Java z rt.jar. Jednak począwszy od Javy 9 ładuje podstawowe pliki Java z obrazu środowiska uruchomieniowego Javy. Ładowacz klas startowych działa niezależnie, bez żadnych nadrzędnych ładowaczy klas.

- **Ładowacz klas platformy**– W wersjach Javy przed Javą 9 istniał moduł ładowacz rozszerzeń, ale od Javy 9 jest on nazywany modułem ładowacz klas platformy. Ładuje on rozszerzenia specyficzne dla platformy z systemu modułów JDK. Ładowacz klas platformy ładuje pliki z obrazu środowiska wykonawczego Java lub z dowolnego innego modułu określonego przez właściwość systemową java.platform lub –module-path.

- **Systemowy ładowacz klas** – Znany również jako ładowacz klas aplikacji, ładuje klasy z classpath aplikacji. Jest potomkiem ładowacza klas platformy. Klasy są ładowane z katalogów określonych przez zmienną środowiskową CLASSPATH, opcję wiersza poleceń -classpath lub -cp.

### 5.2.2. Łączenie

Gdy klasa zostanie załadowana do pamięci, nadal nie jest gotowa do wykonania. Zanim jakiekolwiek metody będą mogły zostać uruchomione, JVM musi przetworzyć klasę dalej. W tym miejscu pojawia się łączenie.

Łączenie następuje po załadowaniu klasy i przygotowaniu jej do wykonania. Proces ten składa się z trzech głównych kroków: weryfikacji, przygotowania i rozwiązania.

- **Weryfikacja** – Pierwszą podfazą fazy łączenia jest weryfikacja (ang. *verifying*). Podczas tej fazy sprawdzana jest poprawność kodu bajtowego. Proces ten wykonywany jest przez weryfikator kodu bajtowego. Sprawdzane są tam między innymi informacje o tym czy nie wywołujemy metod prywatnych z innych klas, czy nie nadpisujemy finalnych klas. Jeśli podczas weryfikacji wystąpi błąd to otrzymamy błąd weryfikacji.

  Proces weryfikacji jest procesem kosztownym w kontekście ładowania klas. Natomiast wykonanie tego sprawdzania pozwala na szybsze działanie naszej aplikacji w w czasie wykonania, ponieważ żadna weryfikacja kodu bajtowego nie musi już zachodzić.

- **Przygotowanie** – Wszystkim zmiennym statycznym przydzielana jest pamięć i wartości domyślne.

- **Rozwiązanie** – Wszystkie symboliczne odniesienia do pamięci są zastępowane początkowymi odniesieniami z obszaru metod.

### 5.2.3. Inicjalizacja

Jest to ostatnia faza ładowania klas, podczas której wszystkim zmiennym statycznym przypisane są początkowe wartości i zostaje wykonany blok statyczny.

JC: nazwy po angielsku?

## 5.3. Obszar danych czasu wykonywania

Obszar danych czasu wykonania jest podzielony na 5 głównych komponentów:

### 5.3.1. Obszar metod

Wszystkie dane na poziomie klasy są tutaj przechowywane, łącznie ze zmiennymi statycznymi. Na każdą maszynę JVM przypada tylko jeden obszar metod i jest to zasób współdzielony.

### 5.3.2. Obszar sterty

Wszystkie obiekty i odpowiadające im zmienne instancji oraz tablice są tutaj przechowywane. Istnieje również jeden obszar sterty na każdą maszynę JVM. Ponieważ obszar metod i obszar sterty są wspólne dla wszystkich wątków, dostęp do danych musi być synchronizowany pomiędzy wątkami.

### 5.3.3. Obszar stosu

Dla każdego wątku zostaje utworzony oddzielny stos wykonania. Dla każdego wywołania metody zostaje utworzony jeden wpis w pamięci stosu, nazywany ramką stosu Wszystkie zmienne lokalne zostają utworzone w pamięci stosu. Obszar stosu jest bezpieczny dla wątków, ponieważ nie jest zasobem współdzielonym. Ramka stosu jest podzielona na trzy podjednostki:

- **Tablica zmiennych lokalnych** – Tablica, w której przechowywane są wartości zmiennych lokalnych danej metody.
- **Stos operandów** – Jeśli do wykonania wymagana jest jakakolwiek operacja pośrednia, stos operandów pełni rolę obszaru roboczego środowiska wykonawczego dla wykonania operacji.
- **Dane ramki stosu** – Tutaj przechowywane są wszystkie symbole odpowiadające metodzie. W przypadku jakiegokolwiek wyjątku, informacja o bloku obsługi wyjątków zachowana jest w danych ramki stosu.

### 5.3.4. Rejestr licznika programu

Każdy wątek ma osobny rejestr licznika programu, aby przechowywać adres aktualnie wykonywanej instrukcji. Po jej wykonaniu rejestr rejestr licznika programu zostaje zaktualizowany i wskazuje następną instrukcję.

### 5.3.5. Stosy metod natywnych

Stos metod natywnych przechowuje informacje o metodach natywnych. Dla każdego wątku tworzony jest oddzielny stos metod natywnych.

## 5.4. Silnik wykonawczy

Kod bajtowy przypisany do obszaru danych czasu wykonywania jest wykonywany przez silnik wykonawczy, który odczytuje kod bajtowy i wykonuje go fragmentami.

### 5.4.1. Interpreter

JC: Ale co ten interpreter robi? Bezpośrednio wykonuje każdą instrukcję

Interpretuje bajtkod linia po linii odpowiednio uaktualniając struktury czasu wykonania, a następnie wykonuje. Działa dość wolno. Dodatkową wadą jest to, że gdy jedna metoda jest wywoływana wiele razy, za każdym razem wymagana jest interpretacja.

### 5.4.2. Kompilator JIT

Służy do zwiększenia wydajności interpretera. Kompiluje kod bajtowy i zmienia go na kod natywny, więc kiedykolwiek interpreter widzi powtarzające się wywołania metod, JIT dostarcza bezpośredni kod natywny dla tej części, więc reinterpretacja nie jest wymagana, a tym samym wydajność jest zwiększona.

- **Generator kodu pośredniego** – tworzy kod pośredni
- **Optymalizator kodu** – odpowiedzialny za optymalizację kodu pośredniego wygenerowanego powyżej
- **Generator kodu docelowego** – odpowiedzialny za generowanie kody maszynowego lub natywnego
- **Profiler** – Specjalny komponent odpowiedzialny za wyszukiwanie hotspotów, czyli czy metoda jest wywoływana wielokrotnie czy nie.

### 5.4.3. Zbieracz śmieci:

Zbieracz śmieci jest odpowiedzialny za zwalnianie pamięci w stercie. Robi to, patrząc na pamięć sterty, identyfikując, które obiekty są używane, a które nie, i usuwając nieużywane obiekty. Proces ten odbywa się w trzech krokach, mianowicie:

- **Oznaczanie:** GC identyfikuje, które części pamięci sterty są używane, a które nie. Odbywa się to poprzez skanowanie wszystkich obiektów w celu zidentyfikowania obiektów odwoływanych i nieodwoływanych.
- **Zwykłe usuwanie:** Zwykłe usuwanie usuwa obiekty nieodwoływane, pozostawiając obiekty odwoływane i wskaźniki do wolnej przestrzeni.
- **Usuwanie i kompaktowanie:** Obiekty nieodwoływane są usuwane, a obiekty odwoływane są kompaktowane razem, co ułatwia i przyspiesza przydzielanie nowej pamięci.

### 5.4.4. Java Native Interface (JNI):

JNI pozwala na dostęp do kodu w C, C++ podczas programowania w Javie. Umożliwia komunikację pomiędzy klasami w Javie a funkcjami natywnymi, napisanymi w C, C++. Dostarcza funkcjonalności dołączania bibliotek C, C++ do maszyny wirtualnej oraz interfejsu tłumaczącego dane pomiędzy warstwą w Javie, a wartstwą natywną.

### 5.4.5. Biblioteki metod natywnych

Jest to zbiór bibliotek natywnych, które są wymagane przez silnik wykonawczy.

## 5.5. Architektura maszyny wirtualnej Pythona

Jak już wspomaniano wcześniej, w pracy skupiono się na najpopularniejszej (napisanej w C) implementacji języka Python jaką jest CPython. Ta sekcja dotyczy kodu źródłowego CPythona w wersji 3.9. Nowsze wersje mogą się różnić w implementacji.

Wywołanie programu Pythona składa się z trzech etapów:

- **Inicjalizacja**
- **Translacja**
- **Interpretacja**

Podczas fazy inicjalizacji, CPython inicjalizuje struktury danych wymagane do działania Pythona. Przygotowuje również typy wbudowane, konfiguruje moduły wbudowane, konfiguruje system importu i wykonuje wiele innych czynności.

Teraz następuje faza translacji. CPython jest interpreterem, a nie kompilatorem w tym sensie, że nie tworzy kodu maszynowego. Zamiast tego, jak wiele innych interpreterów, przed wykonaniem tłumaczy kod źródłowy do postaci pośredniej, aby zmniejszyć potem narzut choćby na parsowanie źródeł. Etap translacji wykonuje te same czynności (oprócz generowania natywnego kodu wykonywalnego) jakie wykonuje typowy kompilator: parsuje kod źródłowy, buduje AST (Abstract Syntax Tree), generuje bajtkod z AST i wykonuje nawet pewne optymalizacje bajtkodu.

Podczas fazy interpretacji następuje wykonanie bajtkodu przez maszynę witrtualną. Wykonanie to ma miejsce w ogromnej pętli, która działa póki są instrukcje do wykonania. Zatrzymuje się ona by pobrać wartość lub gdy wystąpił błąd.

### 5.5.1. Obiekt kodu

Fragmenty kodu, które wykonywane są jako pojedyncza jednostka takie jak moduł lub treść funkcji zwane są blokami kodu. CPython przechowuje informacje na temat tego co kod bloku robi w strukturze zwanej obiekt kodu. Zawiera ona bajtkod i dane takie jak lista nazw zmiennych używanych w bloku. Uruchomienie modułu lub wywołanie funkcji oznacza start obliczeń odpowiadającego obiektu kodu.

Poniżej znajduje się definicja struktury obiektu kodu.

```
1  struct PyCodeObject {
2      PyObject_HEAD
3      int co_argcount;            /* #arguments, except *args */
4      int co_posonlyargcount;     /* #positional only arguments */
5      int co_kwonlyargcount;      /* #keyword only arguments */
6      int co_nlocals;             /* #local variables */
7      int co_stacksize;           /* #entries needed for evaluation
       stack */
8      int co_flags;               /* CO_..., see below */
9      int co_firstlineno;         /* first source line number */
10     PyObject *co_code;          /* instruction opcodes */
11     PyObject *co_consts;        /* list (constants used) */
12     PyObject *co_names;         /* list of strings (names used) */
13     PyObject *co_varnames;      /* tuple of strings (local variable
       names) */
14     PyObject *co_freevars;      /* tuple of strings (free variable
       names) */
15     PyObject *co_cellvars;      /* tuple of strings (cell variable
       names) */
16
17     Py_ssize_t *co_cell2arg;    /* Maps cell vars which are arguments.
        */
18     PyObject *co_filename;      /* unicode (where it was loaded from)
       */
19     PyObject *co_name;          /* unicode (name, for reference) */
```

```
        /* ... more members ... */
};
```

### 5.5.2. Obiekt funkcyjny

Funkcje są nie tylko obiektami kodu. Muszą zawierać dodatkowe informacje takie jak nazwa funkcji, domyślne argumenty i wartości zmiennych zdefiniowanych w ich zasięgu. Te informacje, razem z obiektem kodu są przechowywane w obiekcie funkcji.

Poniżej znajduje się definicja typu struktury obiektu funkcyjnego.

```
typedef struct {
    PyObject_HEAD
    PyObject *func_code;        /* A code object, the __code__
    attribute */
    PyObject *func_globals;     /* A dictionary (other mappings won't
    do) */
    PyObject *func_defaults;    /* NULL or a tuple */
    PyObject *func_kwdefaults;  /* NULL or a dict */
    PyObject *func_closure;     /* NULL or a tuple of cell objects */
    PyObject *func_doc;         /* The __doc__ attribute, can be
    anything */
    PyObject *func_name;        /* The __name__ attribute, a string
    object */
    PyObject *func_dict;        /* The __dict__ attribute, a dict or
    NULL */
    PyObject *func_weakreflist; /* List of weak references */
    PyObject *func_module;      /* The __module__ attribute, can be
    anything */
    PyObject *func_annotations; /* Annotations, a dict or NULL */
    PyObject *func_qualname;    /* The qualified name */
    vectorcallfunc vectorcall;
} PyFunctionObject;
```

### 5.5.3. Obiekt ramki stosu

Kiedy maszyna wirtualna wykonuje obiekt kodu, musi śledzić wartości zmiennych i stale zmieniający się stos wartości. Musi także pamiętać, gdzie zatrzymała wykonywanie bieżącego obiektu kodu, aby wykonać inny i dokąd ma się udać po powrocie. CPython przechowuje te informacje wewnątrz obiektu ramki lub po prostu w ramce. Ramka zapewnia stan, w którym można wykonać obiekt kodu.

Poniżej znajduje się definicja obiektu ramki stosu.

```
struct _frame {
    PyObject_VAR_HEAD
    struct _frame *f_back;      /* previous frame, or NULL */
    PyCodeObject *f_code;       /* code segment */
    PyObject *f_builtins;       /* builtin symbol table (PyDictObject)
    */
    PyObject *f_globals;        /* global symbol table (PyDictObject)
    */
    PyObject *f_locals;         /* local symbol table (any mapping) */
    PyObject **f_valuestack;    /* points after the last local */

```

```
    PyObject **f_stacktop;          /* Next free slot in f_valuestack.
     ... */
    PyObject *f_trace;          /* Trace function */
    char f_trace_lines;         /* Emit per-line trace events? */
    char f_trace_opcodes;       /* Emit per-opcode trace events? */

    /* Borrowed reference to a generator, or NULL */
    PyObject *f_gen;

    int f_lasti;                /* Last instruction if called */
    /* ... */
    int f_lineno;               /* Current line number */
    int f_iblock;               /* index in f_blockstack */
    char f_executing;           /* whether the frame is still
   executing */
    PyTryBlock f_blockstack[CO_MAXBLOCKS]; /* for try and loop blocks
   */
    PyObject *f_localsplus[1];  /* locals+stack, dynamically sized */
};
```

Pierwsza ramka jest tworzona w celu wykonania obiektu kodu modułu. CPython tworzy nową ramkę za każdym razem, gdy musi wykonać inny obiekt kodu. Każda ramka ma odniesienie do poprzedniej ramki. W ten sposób ramki tworzą stos ramek, zwany także stosem wywołań, przy czym bieżąca ramka znajduje się na górze. Po wywołaniu funkcji nowa ramka jest umieszczana na stosie. Po powrocie z aktualnie wykonywanej ramki CPython kontynuuje wykonywanie poprzedniej ramki, zapamiętując jej ostatnią przetworzoną instrukcję. W pewnym sensie maszyna wirtualna CPython nie robi nic innego, jak konstruuje i wykonuje ramki.

Model czystego stosu nie wystarcza do opisania działania Pythona, jeśli wchodzą w grę:

- generatory (`yield`)
- funkcje asynchroniczne (`async def + await`)
- korutyny
- zielone wątki / współprogramy

W tych przypadkach mamy do czynienia z ramkami, które są utrzymywane niezależnie od klasycznego stosu. Dzięki temu możliwe jest zawieszanie i wznawianie funkcji, a także wykonywanie współbieżne bez fizycznego równoległego wykonania kodu.

### 5.5.4. Stan wątku

Stan wątku to struktura danych zawierająca dane specyficzne dla wątku, w tym stos wywołań, stan wyjątku i ustawienia debugowania.

Poniżej znajduje się definicja typu struktury thread state.

```
    typedef struct _ts {
        struct _ts *prev;
        struct _ts *next;
        PyInterpreterState *interp;

        struct _frame *frame;
```

```
        int recursion_depth;
        char overflowed;
        char recursion_critical;
        int tracing;
        int use_tracing;

        Py_tracefunc c_profilefunc;
        Py_tracefunc c_tracefunc;
        PyObject *c_profileobj;
        PyObject *c_traceobj;

        PyObject *curexc_type;
        PyObject *curexc_value;
        PyObject *curexc_traceback;

        PyObject *exc_type;
        PyObject *exc_value;
        PyObject *exc_traceback;

        ...

    } PyThreadState;
```

### 5.5.5. Stan interpretera

Stan interpretera to grupa wątków wraz z danymi specyficznymi dla tej grupy. Wątki współdzielą takie rzeczy, jak załadowane moduły (`sys.modules`), wbudowane elementy (`builtins.__dict__`) i system importowania (`importlib`).

Poniżej znajduje się definicja typu struktury interpreter state.

```
        typedef struct _is {

            struct _is *next;
            struct _ts *tstate_head;

            PyObject *modules;
            PyObject *modules_by_index;
            PyObject *sysdict;
            PyObject *builtins;
            PyObject *importlib;

            PyObject *codec_search_path;
            PyObject *codec_search_cache;
            PyObject *codec_error_registry;
            int codecs_initialized;
            int fscodec_initialized;

            ...

            PyObject *builtins_copy;
            PyObject *import_func;
        } PyInterpreterState
```

### 5.5.6. Stan środowiska uruchomieniowego

Stan środowiska wykonawczego jest zmienną globalną. Przechowuje dane specyficzne dla procesu. Obejmuje to m.in stan CPythona (np. czy został zainicjowany czy nie), dane o zarządzaniu pamięcią i mechanizm `GIL` (globalna blokada w interpreterze CPython, która sprawia, że w danym momencie tylko jeden wątek może wykonywać kod Pythona - bajtkod).

### 5.5.7. Pętla ewaluacyjna

Wykonanie kodu bajtowego Pythona przez maszynę wirtualną wydaje się skomplikowane. W rzeczywistości jedyne co musi robić to iterować po instrukcjach i działać zgodnie z nimi. I to robi funkcja `_PyEval_EvalFrameDefault()`. Zawiera nieskończoną pętlę for (;;), którą nazywamy pętlą obliczeń. Wewnątrz tej pętli znajduje się ogromna instrukcja switch obejmująca wszystkie możliwe kody operacji. Do wykonania każdego kodu operacji potrzebny jest odpowiedni blok przypadków z kodem. Pierwszym zadaniem interpretera jest dekodowanie instrukcji bajtkodu (bytecode). Bajtkod jest przechowywany jako tablica 16-bitowych jednostek kodu (`_Py_CODEUNIT`). Każda taka jednostka zawiera 8-bitowy kod operacji (**opcode**) oraz 8-bitowy argument (**oparg**) — oba są liczbami bez znaku (**unsigned**).

Aby format bajtkodu był niezależny od kolejności bajtów (**endianness**) na różnych maszynach podczas zapisu na dysk, kod operacji (**opcode**) jest zawsze przechowywany jako pierwszy bajt, a argument operacji (**oparg**) jako drugi bajt.

Do wyodrębniania tych dwóch składników z jednostki kodu stosuje się makra:

- `_Py_OPCODE(word)` – do pobierania kodu operacji
- `_Py_OPARG(word)` – do pobierania argumentu operacji

Maszyna wirtualna śledzi następną instrukcję do wykonania za pomocą zmiennej `next_instr`, która jest wskaźnikiem do tablicy instrukcji. Na początku każdej iteracji pętli ewaluacyjnej maszyna wirtualna oblicza następny kod operacji i jego argument. Bierze odpowiednio najmniej znaczący i najbardziej znaczący bajt następnej instrukcji i zwiększa wartość `next_instr`. Funkcja `_PyEval_EvalFrameDefault()` ma prawie 3000 linii, ale jej kod można przedstawić w uproszczonej wersji:

```
PyObject*
_PyEval_EvalFrameDefault(PyThreadState *tstate, PyFrameObject *f, int
    throwflag)
{
    // ... declarations and initialization of local variables
    // ... macros definitions
    // ... call depth handling
    // ... code for tracing and profiling

    for (;;) {
        // ... check if the bytecode execution must be suspended,
        // e.g. other thread requested the GIL

        // NEXTOPARG() macro
        _Py_CODEUNIT word = *next_instr; // _Py_CODEUNIT is a typedef
    for uint16_t
        opcode = _Py_OPCODE(word);
        oparg = _Py_OPARG(word);
```

```
17          next_instr++;
18
19          switch (opcode) {
20              case TARGET(NOP) {
21                  FAST_DISPATCH(); // more on this later
22              }
23
24              case TARGET(LOAD_FAST) {
25                  // ... code for loading local variable
26              }
27
28              // ... 117 more cases for every possible opcode
29          }
30
31          // ... error handling
32      }
33
34      // ... termination
35  }
```


Java
6
python™

# Rozdział 6

# Porównanie JIT Python'a i Javy.

## 6.1. JIT Javy

### 6.1.1. Co to jest kompilacja Just-In-Time (JIT)?

Kompilacja Just-In-Time (JIT) to kluczowy element modelu wykonywania aplikacji w języku Java. Jest to proces, który dynamicznie optymalizuje wydajność aplikacji Java w czasie jej działania. Mówiąc prościej, kompilacja JIT polega na tłumaczeniu bajtkodu Java na natywny kod maszynowy „w locie", czyli dokładnie wtedy, gdy jest on potrzebny do wykonania.

W przeciwieństwie do kompilacji AOT (Ahead Of Time), która kompiluje kod przed uruchomieniem programu — co skutkuje szybszym wykonywaniem pętli, ale wolniejszym startem — JIT tłumaczy kod w czasie rzeczywistym, zapewniając szybszy start programu i dostosowując się do aktywnych ścieżek kodu. Takie dynamiczne podejście pozwala generować zoptymalizowany kod natywny odpowiedni do bieżącego kontekstu wykonywania.

Maszyna Wirtualna Javy (JVM) uruchamia aplikacje Java i współpracuje z kompilatorem JIT. Podczas działania programu JVM komunikuje się z JIT-em, który identyfikuje często używane fragmenty kodu i przekształca bajtkod w wydajny kod maszynowy, co przyspiesza jego wykonanie. Dzięki tej współpracy JVM i JIT zapewniają płynne działanie aplikacji Java i wysoką wydajność.

### 6.1.2. Jak działa JIT:

Zrozumienie, jak działa JIT, pozwala lepiej pojąć szybkość i efektywność Javy. Oto krok po kroku, jak to wygląda:

- Kompilacja kodu źródłowego Javy do bajtkodu:

  Wszystko zaczyna się od kodu źródłowego w języku Java. Zanim możliwe będzie jego uruchomienie, musi zostać przekształcony. Dokonuje tego kompilator Javy, zwykle uruchamiany poleceniem javac, który kompiluje kod do bajtkodu.

- Ładowanie i interpretacja:

  Po uruchomieniu aplikacji Java do działania przystępuje Maszyna Wirtualna Javy (JVM). Na początku ładuje one skompilowane klasy z kodem metod i przygotowuje je do wykonania. Następnie JVM zaczyna interpretację bajtkodu. Taki sposób działania jest skuteczny, jednak dość wolny, zwłaszcza w przypadku często wykorzystywanych fragmentów kodu.

- Aktywacja kompilacji JIT:

  Kiedy aplikacja wywołuje metodę, JVM identyfikuje często używane metody i oznacza je jako „gorące" ścieżki kodu. W przypadku tych „gorących" metod rozpoczyna się aktywacja JIT. Jest to dynamiczna kompilacja zachodząca w czasie rzeczywistym podczas działania programu.

- Kompilacja JIT:

  Kompilator JIT, zawsze gotowy w tle, zaczyna działać. Pobiera kod bajtowy „gorących" metod i przekształca go w wydajny natywny kod maszynowy. Ten natywny kod jest dostosowany do konkretnego kontekstu wykonania, jest wyjątkowo zoptymalizowany i bardzo szybki.

- Zoptymalizowane wykonywanie:

  Dzięki nowemu kodowi natywnemu JVM nie musi już interpretować kodu bajtowego. Zamiast tego bezpośrednio uruchamia wysoko zoptymalizowany kod natywny „gorących" metod. To zoptymalizowane wykonanie jest znacznie szybsze dzięki wydajności natywnego kodu maszynowego. Kompilatory JIT mogą wykonywać różne optymalizacje, takie jak upraszczanie wyrażeń, ograniczanie dostępu do pamięci i używanie wydajnych operacji na rejestrach zamiast bardziej złożonych operacji na stosie.

- Zwiększenie wydajności:

  Końcowym rezultatem jest znaczny wzrost wydajności aplikacji Java. Choć kompilacja JIT zużywa pewną ilość mocy obliczeniowej i pamięci, ta inwestycja szybko się zwraca, ponieważ aplikacja działa znacznie sprawniej. Warto jednak pamiętać, że w początkowych momentach uruchamiania JVM — kiedy wywoływanych jest wiele metod — może wystąpić krótkie opóźnienie startowe spowodowane pracą kompilatora JIT.

  W skrócie, kompilacja JIT dynamicznie przekształca „gorące" ścieżki kodu bajtowego na wysoce zoptymalizowany kod maszynowy, podczas działania aplikacji Java. Ta optymalizacja zapewnia płynne i wydajne działanie programów Java, czyniąc JIT jednym z filarów sukcesu Javy w dostarczaniu wysokiej wydajności i niezależności od platformy.

### 6.1.3. Optymalizacje stosowane przez kompilatory JIT:

Optymalizacja kodu jest siłą kompilacji JIT. Przedstawię kolejne punkty kompilacji JIT.

- Wstawianie funkcji (Inlining):

  Inlining to proces włączania kodu metody wywoływanej w ciało metody wywołującej, co redukuje narzut związany z wywołaniem metody. Ta technika przyspiesza często wykonywane wywołania. Obejmuje optymalizacje takie jak proste wstawianie, inlining na podstawie grafu wywołań, eliminacja rekurencji ogonowej oraz optymalizacje związane z wywołaniami wirtualnymi.

- Lokalne optymalizacje:

  Lokalne optymalizacje usprawniają niewielkie fragmenty kodu, wykorzystując techniki takie jak lokalna analiza przepływu danych, optymalizacja użycia rejestrów oraz uproszczenia typowych konstrukcji języka Java.

- Optymalizacje przepływu sterowania:

  Optymalizacje przepływu sterowania analizują i reorganizują ścieżki wykonania kodu w celu poprawy efektywności. Obejmują one przestawianie instrukcji, ulepszenia związane z pętlami (redukcję, odwracanie, rozwijanie pętli) oraz inteligentniejsze zarządzanie obsługą wyjątków.

- Globalne optymalizacje:

  Globalne optymalizacje działają na całej metodzie, wymagając więcej czasu kompilacji, ale oferując znaczący wzrost wydajności. Obejmują one globalne analizy przepływu danych, eliminację częściowej redundancji, analizę ucieczki (escape analysis) oraz optymalizacje związane z alokowaniem pamięci i zbieraniem śmieci (garbage collection).

- Generowanie kodu natywnego:

  W ostatniej fazie struktury reprezentujące kod są tłumaczone na instrukcje maszynowe, a także stosowane są optymalizacje specyficzne dla danej platformy. Skompilowany kod jest przechowywany w pamięci podręcznej JVM, gotowy do ponownego wykorzystania, co zapewnia szybsze wykonywanie.

  Kompilacja JIT to proces, który przekształca zwykły kod Javy w wydajny, zoptymalizowany, szybki kod maszynowy, gotowy do natychmiastowego wykonania.

JC: nic dotąd nie było o drzewach... Zresztą bajkod i drzewa???

JC: nie wiem czy mu się należy aż tyle entuzjazmu... może po prostu wydajny, zoptymalizowany, szybki takie coś

### 6.1.4. Zalety kompilatora JIT:

- Efektywne wykorzystanie pamięci:

  Kompilatory JIT zużywają mniej pamięci, ponieważ generują natywny kod maszynowy tylko dla wykonywanych ścieżek kodu, zamiast kompilować cały program od razu.

- Dynamiczna optymalizacja kodu:

  Kompilatory JIT optymalizują kod w czasie wykonywania, co pozwala im dostosować się do kontekstu wykonania i poprawić wydajność.

- Wiele poziomów optymalizacji:

  Kompilatory JIT wykorzystują różne poziomy optymalizacji, dopasowując optymalizacje do konkretnego wykonywanego kodu, co może znacznie zwiększyć szybkość wykonywania.

- Zmniejszona liczba błędów strony:

  Generując w razie potrzeby natywny kod maszynowy, kompilatory JIT zmniejszają prawdopodobieństwo błędów stron i operacji we/wy dysku, kosztem chwilowego wzrostu czasu reakcji aplikacji.

JC: zwiększając? Może... kosztem chwilowego wzrostu

### 6.1.5. Wady kompilatora JIT:

- Zwiększona złożoność programu

  Kompilacja JIT powoduje że proces wykonywania programu jest bardziej złożony, co może utrudniać debugowanie i profilowanie.

- Niewielkie korzyści w przypadku krótkiego kodu:

Kompilacja JIT może nie zapewniać znaczących korzyści w przypadku małych fragmentów kodu lub programów o minimalnym czasie wykonania, ponieważ narzut związany z kompilacją może przeważyć nad korzyściami.

- Wykorzystanie pamięci podręcznej:

  Kompilatory JIT zużywają znaczną ilość pamięci podręcznej, co może negatywnie wpłynąć na ogólną wydajność systemu, szczególnie w środowiskach o ograniczonych zasobach.

**Wnioski:**

Zalety JIT to efektywne wykorzystanie pamięci, dynamiczna optymalizacja kodu, wiele poziomów optymalizacji oraz zmniejszona liczba błędów strony, które razem przyczyniają się do poprawy wydajności. Jednakże JIT wprowadza pewną złożoność i może nie przynosić znaczących korzyści w przypadku bardzo krótkich fragmentów kodu. To kompromis, który warto podjąć ze względu na szybkość i elastyczność, jaką wnosi do ekosystemu Javy.

## 6.2. JIT Pythona

CPython od wersji 3.13 obsługuje JIT. Wykorzystuje on technikę copy-and-patch.`JC: w cudzysłowie czy bez?`

### 6.2.1. Co to jest copy-and-patch JIT?

`JC: To zdanie nie wnosi żadnej nowej informacji! Działanie techniki ,,copy-and-patch'' można syntetycznie opisać tak:`

Działanie techniki copy-and-patch można syntetycznie opisać tak:

$$\text{machine\_code}(f) = \text{template}(\mathit{opcodes}) \oplus \text{patch}(\mathit{consts}, \mathit{vars})$$

Innymi słowy:

1. Na etapie kompilacji ze źródeł generowane są szablony kodu maszynowego dla poszczególnych bajtkodów`JC: to jest dobrze to słowo? Jeśli tak, to powinno zostać wprowadzone dużo wcześniej, np. w rozdziale 3.3` (np. LOAD_CONST) z pewnymi „dziurami” (ang. holes) pozostawionymi bez wypełnienia.

2. W czasie wykonywania programu, gdy wystąpi dana funkcja, odpowiedni szablon jest kopiowany (copy), a „dziury” wypełniane są wartościami argumentów (patch).

3. Po tym przygotowaniu kod maszynowy jest wykonywany bezpośrednio, omijając pętlę interpretera.

`JC: Taki pół-przykład nie jest dobry! Jak Pan daje kod, to proszę przeprowadzić czytelnika aż do efektu końcowego, a nie tylko wypisać nazwy bajtokodów(?). Bo co dokładnie jest ilustrowane przez ten ,,przykład'' obecnie?`

### 6.2.2. Dlaczego wybrano tę technikę?

JC: Co to IL? JC: co to hoistowanie pętli?

Pełny JIT (kompilacja bajtokodu → język pośredni → kod maszynowy) oferuje duże możliwości optymalizacji (np. propagacja stałych, hoistowanie pętli - optymalizacja kompilatora, która ma na celu przyspieszenie pętli przez przeniesienie niezmiennych obliczeń poza jej ciało)

lecz jego implementacja w środowisku tak heterogenicznym jak Python jest bardzo kosztowna `JC: Ale czas uruchomienia czy pamięć to ,,koszt'' zupełnie innego rodzaju, niż architektury!?! Nie można tego tak pisać w jednym ciurku.` „Copy-and-patch" stanowi kompromis: mniej skomplikowana implementacja, mniejsze narzuty, przy zachowaniu korzyści wynikających z kodu maszynowego.

### 6.2.3. Jak to działa w CPython 3.13?

Wskazówki techniczne:

- Przy kompilacji interpretera można użyć flagi `-enable-experimental-jit`, co spowoduje wygenerowanie szablonów kodu maszynowego dla bajtokodów.
- Szablony kodu zawierają „dziury" (holes) np. dla argumentu bajtokodu (`oparg`) i dla adresu instrukcji „continue".
- JIT w tej wersji CPythona wymaga LLVM (oraz narzędzia Clang z jego ekosystemu) podczas budowania interpretera.
- W obecnej wersji kompilatora JIT aktywuje się tylko, jeśli funkcja zawiera bajtkod `JUMP_BACKWARD` (np. w pętli while) — co czyni go heurystycznie ograniczonym.

JC: Ale jakiej fazie? Mówimy o JIT do CPython 3.13? Innej fazy nie kojarzę. Może: W obecnej wersji kompilatora

### 6.2.4. Jak to działa na przykładzie

`JC: w tym podrozdziale musi być coś napisane, nie same tabelki. A samych tabelkach nie wiadomo co jest czym. Raczej napisałbym po polsku: kod źródłowy:... bajtkod:... A co to jest to` $T_{LOAD\_CONST}$ `to w ogóle trudno powiedzieć... To są te szablony/templaty opisane wcześniej? Dlaczego w JIT(func) nie występuje machine_code(func), tylko "generuj kod maszynowy"? To po co jest to wcześniejsze? W dodatku skoro kod maszynowy generujemy tylko dla pętli to a=13; return a jest kiepskim przykładem... Może jakąś małą pętlę dałoby się zakodować? Dałoby się wyciągnąć konkretne instrukcje bajtkodu i kodu maszynowego? Np. dla`

`def nibylog(n): i = -1 while n > 0: n //= 2 i += 1 return i`

`Ta funkcja (w C) kompiluje się za pomocą gcc do 10 instrukcji, więc wersja kompilatora Pythona pewnie też się "zmieści". Jeśli wyciąganie konkretnego kodu jest zbyt skomplikowane to proponuję w ogóle usunąć przykład. Bo tak jak pisałem wcześniej, przykład jest dobry jak naprawdę pokazuje co się dzieje, a tu tylko mamy jakieś tajemnicze` $T_c otam$, `które nie wiadomo czym jest...`

### 1. Python interpretuje kod

Funkcja:

```python
def nibylog(n):
    i = -1
    while n > 0:
        n //= 2
        i += 1
    return i
```

Bajtkod funkcji:

```
  3           RESUME                   0

  4           LOAD_CONST               1 (-1)
              STORE_FAST               1 (i)

  5           LOAD_FAST                0 (n)
              LOAD_CONST               2 (0)
              COMPARE_OP             148 (bool(>))
              POP_JUMP_IF_FALSE       18 (to L2)

  6   L1:     LOAD_FAST                0 (n)
              LOAD_CONST               3 (2)
              BINARY_OP               15 (//=)
              STORE_FAST               0 (n)

  7           LOAD_FAST                1 (i)
              LOAD_CONST               4 (1)
              BINARY_OP               13 (+=)
              STORE_FAST               1 (i)

  5           LOAD_FAST                0 (n)
              LOAD_CONST               2 (0)
              COMPARE_OP             148 (bool(>))
              POP_JUMP_IF_FALSE        2 (to L2)
              JUMP_BACKWARD           18 (to L1)

  8   L2:     LOAD_FAST                1 (i)
              RETURN_VALUE
```

### 2. Wykrycie gorącej pętli

- JIT obserwuje wykonywanie kodu. - Jeśli pętla jest powtarzana wiele razy, uznaje ją za **gorącą**.

### 3. Kompilacja gorącej pętli

- JIT kompiluje gorącą pętlę do natywnego kodu procesora. - Pętla ‘while n $>$ 0‘ staje się zestawem instrukcji CPU, np.:

`cmp, shr, inc, jmp`

### 4. Wykonanie natywne

- Dalsze iteracje są wykonywane jako natywne instrukcje, bez pośrednictwa interpretera. - Efekt: znaczące przyspieszenie pętli.

### 5. Schemat przepływu

Python source $\longrightarrow$ Bytecode CPython $\longrightarrow$ gorąca pętla JIT $\longrightarrow$ instrukcje CPU

### 6.2.5. Perspektywy na przyszłość

Implementacja ta otwiera drogę do znacznie głębszych optymalizacji w interpreterze Pythona — przede wszystkim tam, gdzie adaptacyjny interpreter czy proste optymalizacje nie wystarczą. W przyszłości można oczekiwać:

- rozszerzenia heurystyk aktywacji JIT-a (nie tylko pętle),
- bardziej zaawansowanych optymalizacji kodu maszynowego bazującego na szablonach,
- możliwości publikacji JIT-u jako domyślnego w kolejnych wersjach CPython.

### 6.2.6. Zakończenie

Wersja 3.13 interpretera CPython wprowadza opcjonalny JIT oparty na technice „copy-and-patch". Choć obecnie przyspieszenie jest umiarkowane, to krok ten może zapoczątkować duże zmiany. `JC 11-02: Co to jest GIL i free-threaded ??? Może coś takiego: W wydanej w ostatnich tygodniach najnowszej wersji Pythona 3.14, JIT jest wciąż w fazie eksperymental wspieranej jedynie w niektórych wariantach generowania CPythona. Dokładniejsze porównanie w poniższej tabelce.` W wydanej w ostatnich tygodniach najnowszej wersji Pythona 3.14, JIT jest wciąż w fazie eksperymentalnej, wspieranej jedynie w niektórych wariantach generowania CPythona. Dokładniejsze porównanie w poniższej tabelce.

| **Cecha** | **CPython 3.13** | **CPython 3.14** |
|---|---|---|
| Status | Eksperymentalny, ręczny build | Eksperymentalny, binaria dostępne |
| Aktywacja | `-enable-experimental-jit`, `PYTHON_JIT` | `PYTHON_JIT`, introspekcja `sys._jit` |
| Architektura | Copy-and-patch | Copy-and-patch + introspekcja |
| Wydajność | Niekiedy wolniejszy niż interpreter | Podobny zakres, -10% do +20% |
| Zależności | LLVM wymagany do szablonów | PEP 774: prebuilt stencils, mniej zależności |

Tabela 6.1: Porównanie eksperymentalnego JIT w CPython 3.13 vs 3.14

Wyjaśnienie:

- **introspekcja sys._jit** to mechanizm który pozwala na ograniczoną obserwację działania JIT w czasie wykonywania.
- **LLVM (Low Level Virtual Machine)** [26] to `JC 11-02: to znany framework... przydałby się może też link, albo lepiej odwołanie bibliograficzne` framework do kompilacji i optymalizacji kodu
- **LLVM i prebuilt stencils (PEP 774):** W CPython 3.13 eksperymentalny JIT wymagał **LLVM** do generowania szablonów kodu maszynowego (stencils) podczas kompilacji. W CPython 3.14, dzięki **PEP 774**, wprowadzono **prebuilt stencils** – gotowe szablony dołączone do repozytorium, co redukuje zależności od LLVM, upraszcza build i umożliwia łatwe włączenie JIT w gotowych binariach.

`JC 11-02: wspomnienie LLVMa powinno być już wcześniej, przy opisie JITa dla CPythona, a nie dopiero tu. Albo uznajemy to za nieistotny szczegół i pomijamy całkowicie. Ale to chyba zły pomysł, bo to jednak jakaś konkretna różnica 3.13 vs 3.14`


Java
7
python™

# Rozdział 7

# Zarządzanie pamięcią

`JC 11-02: W niniejszym rozdziale przedstawiam...` W niniejszym rozdziale przedstawia się mechanizmy alokacji pamięci, tworzenia obiektów oraz mechanizmy usuwania śmieci stosowane przez wirtualną maszynę Javy i wirtualną maszynę Pythona.

## 7.1. Alokacja pamięci i tworzenie obiektów:

### 7.1.1. JVM

JVM podczas wykonywania kodu wykorzystuje różne (logiczne) obszary pamięci. Te najważniejsze to obszar metod, stos ramek i sterta. Kiedy obiekty są tworzone przy użyciu słowa kluczowego **new**, pamięć jest alokowana na stercie, a konstruktor służy do inicjowania obiektu. Sterta jest podzielona na pokolenia: `JC 11-02: To nie tak. To nie są obszary do zarządzania pamięcią, tylko obszary pamięci, które wymagają zarządzania... Może tak: JVM podczas wykonywania kodu wykorzystuje różne (logiczne) obszary pamięci. Te najważniejsze to obszar metod, stos ramek i sterta. I dalej o new... JC 11-02: Jeśli chodzi o zarządzanie pamiecią, w sensie new i GC to ja bym w ogóle niepisał o Metaspace. Tylko stare i nowe pokolenie, ewentualnie w opisie można dodać coś o drobniejszym podziale (Eden i Survivors)... Tu znalazłem dość szczegółowy i sensowny opis: https://www.digitalocean.com/community/tutorials/java-jvm-memory-model-memory-managem`

#### Młoda Generacja

Młoda Generacja to miejsce, w którym zaczyna się życie wszystkich nowo utworzonych obiektów. Jest zoptymalizowana pod kątem szybkiej alokacji i częstego zbierania śmieci. Ponieważ większość obiektów jest krótkotrwała (np. zmienne lokalne metod czy tymczasowe bufory), ta przestrzeń jest regularnie oczyszczana za pomocą **Minor Garbage Collection (Minor GC)**, która jest zazwyczaj szybka i wydajna.

Podział Young Generation:

- Eden Space – początkowa przestrzeń dla nowych obiektów. JVM umieszcza tutaj nowo tworzone obiekty. Gdy Eden się zapełni, wywoływany jest Minor GC.
- Survivor Spaces (S0 i S1) – dwie przestrzenie dla obiektów, które przetrwały Minor GC. Po każdym Minor GC żywe obiekty z Eden przenoszone są do jednej z przestrzeni

Survivor. Obiekty, które dalej przetrwają, przemieszczają się między S0 i S1 w kolejnych cyklach GC.

#### Stara Generacja

Stara Generacja jest przeznaczona do przechowywania obiektów długotrwałych, czyli takich, które przetrwały wiele Minor GC (mniejszych zbiorów odpadów). Podczas gdy Młoda Generacja obsługuje częste przydzielanie i usuwanie krótkotrwałych danych, Stara Generacja jest miejscem, gdzie ostatecznie przechowywane są obiekty pozostające w użyciu przez dłuższy czas.

`JC 11-02: nie można tak w powietrzu tego zostawić... np. Służy głównie do przechowywania danych globalnych klas.`

### 7.1.2. PVM

`JC 11-02: co to jest prosta składnia z automatycznie zarządzaną pamięcią?!?!?! I jaki związek z zarządzaniem pamięcią ma dynamiczne typowanie!?!? Może wystarczy tyle w pierwszym akapicie: Python, podobnie jak Java, oferuje automatyczne zarządzanie pamięcią.`

`JC 11-02: nie bardzo rozumiem tego schodzenia o jeden poziom... Może prościej: Do alokowania obiektów Pythona, interpreter CPython wykorzystuje API niskopoziomowego alokatora (malloc itp) z poziomu biblioteki C, która przydziela pamięć w systemie operacyjnym.`

`JC 11-02: to dlaczego to jest "prywatna"sterta, skoro używa alokatora z poziomu C ?`

`JC 11-02: Tu by można było wspomnieć o tym jak Python alokuje pamięć dla ramek funkcji (Java ma frame stack, a Python co?), ale niekoniecznie.`

Python, podobnie jak Java, oferuje automatyczne zarządzanie pamięcią. Do alokowania obiektów Pythona, interpreter CPython wykorzystuje API niskopoziomowego alokatora (malloc itp) z poziomu biblioteki C, która przydziela pamięć w systemie operacyjnym.

Sterta Pythona jest prywatna w sensie logicznym. „Prywatna sterta" oznacza, że Python sam kontroluje sposób, w jaki obiekty są rozmieszczane i zarządzane w przydzielonej pamięci, mimo że fizycznie pamięć pochodzi z systemowego alokatora C.

#### Alokacja pamięci dla ramek funkcji w Pythonie

W Pythonie pamięć dla *ramek funkcji* (ang. *frame objects*) jest alokowana na *prywatnej stercie* interpretera CPython.

1. **Tworzenie ramki:** Gdy wywoływana jest funkcja, Python tworzy nową ramkę, która przechowuje zmienne lokalne, referencje do kodu funkcji oraz wskaźnik do funkcji wywołującej.

2. **Alokacja pamięci:** Ramka jest obiektem Pythona, więc jej pamięć przydzielana jest na prywatnej stercie przy użyciu niskopoziomowych funkcji C (`malloc`) lub mechanizmów poolingu Pythona.

3. **Zarządzanie pamięcią:** Po zakończeniu funkcji ramka jest zwalniana, a liczniki referencji obiektów w niej znajdujących się są dekrementowane, co pozwala na automatyczne zwolnienie pamięci.

**Podsumowanie:** Ramki funkcji są krótkotrwałymi obiektami Pythona alokowanymi na prywatnej stercie CPython, z zarządzaniem pamięcią realizowanym przez interpreter i mechanizm *reference counting*.

## 7.2. Odśmiecanie pamięci

### 7.2.1. JVM

#### Fazy zbierania śmieci (Mark, Sweep i Compact)

Niezależnie od używanego algorytmu, większość implementacji GC (Garbage Collector) stosuje wariację procesu **Mark-Sweep-Compact**:

1. **Mark (oznaczanie):** Kolektor skanuje wszystkie żywe referencje i oznacza wszystkie osiągalne obiekty, śledząc je od korzeni GC (GC roots).
2. **Sweep (czyszczenie):** Po zakończeniu oznaczania kolektor odzyskuje pamięć zajmowaną przez obiekty, które nie zostały oznaczone (czyli nieosiągalne).
3. **Compact (opcjonalne, kompakcja):** Aby zmniejszyć fragmentację, niektóre kolektory przesuwają żywe obiekty do ciągłych obszarów pamięci i aktualizują odpowiednio referencje.

Proces ten zapewnia efektywne ponowne wykorzystanie pamięci i minimalizuje fragmentację, co jest szczególnie ważne w aplikacjach działających przez dłuższy czas.

#### Minor GC vs. Major GC vs. Full GC

- **Minor GC:** Zbiera tylko **Młodą Generację (Young Generation)**. Jest częsty i szybki, zazwyczaj obejmuje kopiowanie żywych obiektów do przestrzeni *survivor* lub promowanie ich do **Starej Generacji (Old Generation)**. Minor GC zatrzymuje działanie programu (*stop-the-world*), ale pauzy są zazwyczaj krótkie.
- **Major GC:** Zbiera **Starą Generację (Old Generation)**. Jest mniej częsty, ale kosztowny pod względem czasu i użycia CPU. Te zdarzenia również zatrzymują program i mogą powodować zauważalne przerwy w działaniu aplikacji, jeśli nie są dobrze zarządzane.
- **Full GC:** Zbiera **całą stertę (heap)**, obejmując zarówno Młodą, jak i Starą Generację. W zależności od wersji JVM i używanego kolektora, może również obejmować czyszczenie **Metaspace**. Full GC jest najbardziej zakłócający działanie aplikacji i powinien być unikany w systemach wymagających niskich opóźnień.

`JC 11-02: może głównie, bo klasy z metaspace też są jakoś odśmiecane` na obszarze sterty.

`JC 11-02: dlatego bym ten Metaspace pominął...`

### 7.2.2. PVM

PVM wykorzystuje przede wszystkim mechanizm zliczania referencji do zarządzania pamięcią. Obiekty mają liczbę odwołań, a pamięć jest zwalniana, gdy liczba spada do zera. Wbudowany

garbage collector służy do wykrywania i usuwania nieosiągalnych obiektów z cyklicznymi odniesieniami, które uniemożliwiają zmniejszenie liczby odwołań do zera i obsługuje „zwykłe" usunięcie obiektów.JC 11-02: To to są różne scenariusze???? Może... do wykrywania i usuwania nieosiągalnych obiektów z cyklicznymi odniesieniami, które uniemożliwiają zmniejszenie liczby odwołań do zera i ,,zwykłe'' usunięcie obiektów.


Java
8
python™

# Rozdział 8

# Porównanie czasu działania prostych fragmentów kodu

W celu porównania czasu działania prostych operacji takich jak dodawanie czy mnożenie stosujemy narzędzie perf_counter w przypadku Pythona i polecenie System.nanoTime() w pętli for w przypadku Javy. Do wykonania pomiarów wydajnościowych w tym rozdziale oraz następnych zastosowaliśmy komputer o następującej konfiguracji:

- Procesor AMD Ryzen 5 3600
- 16 GB pamięci RAM
- karta graficzna GeForce GTX 1660
- Płyta główna PRIME B450M-A II

Wykorzystaliśmy następujące oprogramowanie:

- System operacyjny Debian 11
- Java w wersji "23.0.1"z 2024-10-15
- Python 3.13.0 z 2024-10-07

W tym rozdziale korzystamy dodatkowo z narzędzia: JMH (Java Microbenchmark Harness): framework do precyzyjnego benchmarkowania kodu w Javie.

Dlaczego go używać zamiast System.nanoTime()?

1. JVM optymalizuje kod w trakcie działania (JIT), więc proste pomiary mogą być mylące.
2. JMH automatycznie wykonuje warmup, unika dead code elimination i mierzy dokładnie czas/metryki.

## 8.1. Pusta pętla

W przypadku Javy do pomiaru czasu wykonania pustej pętli stosuję następujący kod [Java]:

```java
import java.io.*;

public class Empty {
    // main function
    public static void main(String[] args)
    {
        // Start measuring execution time
        long startTime = System.nanoTime();

        count_function(100_000_000);

        // Stop measuring execution time
        long endTime = System.nanoTime();

        // Calculate the execution time in milliseconds
        long executionTime
            = (endTime - startTime);

        double seconds = (double)executionTime / 1_000_000_000.0;
        System.out.println(seconds + " s");

    }

    public static void count_function(long x)
    {
        for (long i = 0; i < x; i++)
            ;
    }
}
```

Wykonanie powyższego kodu zajmuje około 0,0416 s. W przypadku wyłączenia JIT (java -Xint Empty) czas wykonania wzrasta do około 0,618 s.

Do pomiaru czasu wykonania pustej pętli stosuję następujący kod w przypadku Pythona [Python]:

```python
from time import perf_counter_ns

def my_function():
  for i in range(100_000_000):
    pass

# Start the stopwatch / counter
t1_start = perf_counter_ns()

my_function()

# Stop the stopwatch / counter
t1_stop = perf_counter_ns()

print((t1_stop-t1_start) / 1_000_000_000, 's')
```

Wykonanie powyższego kodu zajmuje około 1,62 s. Jest to więc 1,62/0,0416 = 3,89 raza wolniej niż w przypadku Javy. Po wyłączeniu JIT, Java jest już jedynie 1,62/0,618=2,62 razy szybsza niż Python.

Warto w tym miejscu zaznaczyć, że gdyby kod funkcji my_function znajdował się poza nią, tak jak na poniższym listingu [Python] to program działałby zdecydowanie wolniej. W tym przypadku byłoby to 3,68 s zamiast 1,62 s.

```
from time import perf_counter_ns

# Start the stopwatch / counter
t1_start = perf_counter_ns()

for i in range(100_000_000):
  pass

# Stop the stopwatch / counter
t1_stop = perf_counter_ns()

print((t1_stop-t1_start) / 1_000_000_000, 's')
```

Generalnie przyjmuje się, że szybsze działanie kodu w funkcjach niż poza nimi spowodowane jest przez różnicę prędkości między dostępem do zmiennych globalnych i lokalnych, optymalizację kodu bajtowego, wydajność pamięci podręcznej instrukcji i zarządzanie przestrzenią nazw.

Szczególnie istotny wydaje się czas dostępu do zmiennych. Widać to dobrze na poziomie kodu bajtowego. Na poniższym listingu widzimy kod, który korzysta z bajtkodu STORE_FAST do zachowania aktualnej wartości licznika pętli w zmiennej lokalnej.

```
      L1:     FOR_ITER                3 (to L2)
              STORE_FAST              0 (i)

  6           JUMP_BACKWARD           5 (to L1)
```

Natomiast na poniższym listingu widzimy kod, który korzysta z bajtkodu STORE_NAME do zachowania aktualnej wartości licznika pętli w zmiennej globalnej.

```
      L1:     FOR_ITER                3 (to L2)
              STORE_NAME              4 (i)

  7           JUMP_BACKWARD           5 (to L1)
```

Jak wynika z pomiarów (3,68 s vs 1,62 s), wykorzystanie zmiennych globalnych w pętli znacznie spowalnia program.

## 8.2. Dodawanie w pętli jednej liczby

`JC 11-02: Proponuje odtąd nie wrzucać na listingach funkcji main, może z adnotacją // standardowa treść funkcji main pominięta, może jakoś tak:`

$publicstaticvoidmain(String[]args)//standardowaczmainpominita//...count_function(100_000_000);//...$

W przypadku Javy do pomiaru czasu wykonania dodawania jednej liczby w pętli stosuję następujący kod [Java]:

```
import java.io.*;

public class Addition {
    // main function
```

```
    public static void main(String[] args) //standardowa część main pominięta
    {
        // ...
        count_function(100_000_000);
        // ...
    }

    public static void count_function(long x)
    {
        long j = 0;
        for (long i = 0; i < x; i++)
            j = j + 1;
    }
}
```

Wykonanie powyższego kodu zajmuje około 0,0339 s. W przypadku wyłączenia JIT (java - Xint Addition) czas wykonania wzrasta do około 0,918 s.

Do pomiaru czasu wykonania dodawania jednej liczby w pętli stosuję następujący kod w przypadku Pythona [Python]:

```
from time import perf_counter_ns

def my_function():
  sum = 0
  for i in range(100_000_000):
    sum = sum + 1

# Start the stopwatch / counter
t1_start = perf_counter_ns()


my_function()

# Stop the stopwatch / counter
t1_stop = perf_counter_ns()

print((t1_stop-t1_start) / 1000_000_000, 's')
```

Wykonanie powyższego kodu zajmuje około 3,27 s. Jest to więc 3,27/0,0339=96,5 raza wolniej niż w przypadku Javy. Po wyłączeniu JIT, Java jest już jedynie 10,9/0,918=11,9 szybsza niż Python.

## 8.3. Duże liczby

W przypadku korzystania z bardzo dużych liczb Python okazuje się być szybszy od Javy. Weźmy pod uwagę poniższy kod w Javie [Java]:

```
import java.io.*;
import java.math.BigInteger;

public class Addition4 {
    // main function
    public static void main(String[] args) //standardowa część main pominięta
```

```
    {
        // ...
        count_function(100_000_000);
        // ...
    }

    public static void count_function(long x)
    {
        BigInteger b1, b2;

        b1 = new BigInteger("100");
        int exponent = 100;

        BigInteger result = b1.pow(exponent);
        BigInteger result2 = result.add(BigInteger.valueOf(x));

        b2 = new BigInteger("2000");
        int exponent2 = 2_000;

        BigInteger result3 = b2.pow(exponent2);

        BigInteger sum = BigInteger.ZERO;

        for (BigInteger bi = result; bi.compareTo(result2) < 0; bi =
   bi.add(BigInteger.ONE))
            sum = sum.add(result3);
    }
}
```

Czas jego wykonania wynosi 83 s. Odpowiednik tego kodu w Pythonie wykonuje się w czasie 59,4 s [Python]:

```
from time import perf_counter_ns
import sys

def my_function():
  sum = 0
  r = 2000**2000
  for i in range(100**100, 100**100 + 100_000_000):
    sum = sum + r

# Start the stopwatch / counter
t1_start = perf_counter_ns()

my_function()

# Stop the stopwatch / counter
t1_stop = perf_counter_ns()

print((t1_stop-t1_start) / 1000_000_000, 's')
```

Widać, że implementacja dużych liczb w Pythonie jest lepsza od implementacji BigInteger w Javie. Dobrze widać różnicę w czasie działania dodawania między int, long a BigInteger analizując kod bajtowy dla odpowiednio long:

```
  public static void count_function(long);
```

```
    Code:
       0: lconst_0
       1: lstore_2
       2: lload_2
       3: lload_0
       4: lcmp
       5: ifge          15
       8: lload_2
       9: lconst_1
      10: ladd
      11: lstore_2
      12: goto          2
      15: return
```

i dla BigInteger:

```
  public static void count_function(long);
    Code:
       0: getstatic     #35                 // Field java/math/
  BigInteger.ZERO:Ljava/math/BigInteger;
       3: astore_2
       4: aload_2
       5: lload_0
       6: invokestatic  #41                 // Method java/math/
  BigInteger.valueOf:(J)Ljava/math/BigInteger;
       9: invokevirtual #45                 // Method java/math/
  BigInteger.compareTo:(Ljava/math/BigInteger;)I
      12: ifge          26
      15: aload_2
      16: getstatic     #49                 // Field java/math/
  BigInteger.ONE:Ljava/math/BigInteger;
      19: invokevirtual #52                 // Method java/math/
  BigInteger.add:(Ljava/math/BigInteger;)Ljava/math/BigInteger;
      22: astore_2
      23: goto          4
      26: return
```

Ten drugi jest jak widać bardziej skomplikowany, używa odwołań do statycznych pól klasy BigInteger oraz wywołań metod, których czas wykonania najwyraźniej jest duży. JC 11-02: , używa odwołań do statycznych pól klasy BigInteger oraz wywołań metod, których czas wykonania najwyraźniej jest duży.

## 8.4. Kolekcje

W następnych podpunktach mierzymy czas działania kolekcji.

### 8.4.1. Java Array vs Python List, NumPy Array

Weźmy pod uwagę kod dodawania elementów dwóch tablic JC 11-02: Tu poniżej jest coś nie tak z wcięciami... [Java].

```
1 import java.util.Arrays;
2
3 public class Concat2 {
4
```

```
    public static void main(String[] args) {
        int[] array1 = new int[10000000]; //000000
        int[] array2 = new int[10000000];

        for (int i=0; i < 10000000;i++) {
            array1[i] = i;
            array2[i] = i + 10000000;
        }

        int aLen = array1.length;
        int bLen = array2.length;

        // Start measuring execution time
        long startTime = System.nanoTime();

        int[] result = new int[aLen];

        for (int i = 0; i < aLen; i++) {
            result[i] = array1[i] + array2[i];
        }

        // Stop measuring execution time
        long endTime = System.nanoTime();

        // Calculate the execution time in milliseconds
        long executionTime
            = (endTime - startTime);

        double seconds = (double)executionTime / 1_000_000_000.0;
        System.out.println(seconds + " s");
    }
}
```

Czas działania w Javie wynosi 0.0174. W przypadku Pythona skorzystamy z biblioteki NumPy. Gdy korzystamy z danych liczbowych, biblioteka ta pozwala na znaczne przyspieszenie działań. Dla porządku wywołamy również dodawanie JC 11-02: dodawanie elementów dwóch list, aby wskazać, że jest nieefektywne [Python].

```
import numpy as np
from time import perf_counter_ns

# Define two Python lists
py_list1 = list(range(10000000))
py_list2 = list(range(10000000, 20000000))

# Define two Numpy arrays
np_array1 = np.arange(10000000)
np_array2 = np.arange(10000000, 20000000)

# Adding Python lists
t1_start = perf_counter_ns()
result_list = [a + b for a, b in zip(py_list1, py_list2)]
t1_stop = perf_counter_ns()

print((t1_stop-t1_start) / 1000000000, 's')
```

```
18
19 # Adding Numpy arrays
20 t2_start = perf_counter_ns()
21 result_array = np_array1 + np_array2
22 t2_stop = perf_counter_ns()
23
24 print((t2_stop-t2_start) / 1000000000, 's')
```

Czas działania wynosi 0,416 s dla dodawania `JC 11-02: dodawania` list i 0,0178 s dla dodawania dwóch tablic z biblioteki NumPy. Dzięki bibliotece NumPy kod w Pythonie wykonuje się w zbliżonym czasie do kodu w Javie.

JC 11-02: W kodzie w Pythonie wlicza Pan również czas tworzenia wynikowej listy! Powinien Pan najpierw ją stworzyć, np

$$result_list = [0] * duoimierzyczasptli : foriinrange(...) : result_list[i] = py_list1[i] + py_list2[i]$$

To byłoby porównywalne z testowanym kodem w Javie. Chociaż można by dyskutować, że w obu językach napisaliśmy tak jak to "się robi"w tym języku. BTW. Pomiar w numpy też obejmuje zakładanie tablicy.
KONTRPOMYSŁ: Nie zmieniać kodu w Pythonie, ale wciągnąć zakładanie tablicy do części mierzonej w Javie!
Tak czy siak należałoby się jakoś odnieść do kwestii liczenia bądź nie czasu zakładania tablicy w poszczególnych przypadkach. Zarówno w kodzie Javowym jak i Pythonowym wlicza się czas zakładania tablicy/listy do pomiaru.

### 8.4.2. Dodawanie String'ów do kolekcji

JC 11-02: Zdanie na dwie strony to chyba przesada :) Poniższy program mierzy... i potem: Uzyskaliśmy... Poniższy program [Java] mierzy czas dodawania String'ów do HashSet, TreeSet, HashMap i TreeMap. Są krótkie stringi, zawierające reprezentację dziesiętną kolejnych liczb naturalnych. Tworzone są przed rozpoczęciem pomiaru i w czasie testowania wyciągane z tablicy/list (ze stałym czasem dostępu) w losowej kolejności.

JC 11-02: Dobrze by też było - może właśnie na początku - napisać co to za stringi (krótkie stringi, zawierające reprezentację dziesiętną kolejnych liczb naturalnych) i że są tworzone przed rozpoczęciem pomiaru i w czasie testowania wyciągane z tablicy/list (ze stałym czasem dostępu) w losowej kolejności.

JC 11-02: Wg mnie powinien Pan gdzieś wspomnieć, pewnie trochę wcześniej, że do testowania Javy używa Pan takiego a takiego frameworka. Trzeba podkreślić, że testowanie jest zaawansowane technologicznie ;)

JC 11-02: Dlaczego ma Pan language=C++ a nie language=Java, language=Python ??? Zwłaszcza to drugie dodaje sporo koloru ;)

```
1 package com.avenuecode.snippet;
2
3 import org.openjdk.jmh.annotations.*;
4
5 import java.util.concurrent.TimeUnit;
6
7 import java.io.*;
8
9 import java.util.*;
10
```

```
@State(Scope.Thread)
public class MyBenchmark {

    int x = 10_000_000;
    HashSet<String> counts = new HashSet<String>();
    TreeSet<String> counts2 = new TreeSet<String>();
    HashMap<String, String> map = new HashMap<>();
    TreeMap<String, String> map2 = new TreeMap<>();
    ArrayList<String> mylist = new ArrayList<String>();
    String[] s = new String[x];
    int y = -1;

    /**
     * Setup method to initialize data for the benchmark.
     */
    @Setup(Level.Trial)
    public void setup() {
        for (int i=0; i < x; i++)
            mylist.add(String.valueOf(i));

        Collections.shuffle(mylist);

        for (int i=0; i < x; i++)
            s[i] = mylist.get(i);
    }

    @Fork(value = 1, warmups = 1)
    @Warmup(iterations = 1)
    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    public long _0_HashSetAdd() {
        for (int i=0; i < x; i++)
            counts.add(s[i]);
        y = 0;
        return 0;
    }

    @Fork(value = 1, warmups = 1)
    @Warmup(iterations = 1)
    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    public long _1_TreeSetAdd() {
        for (int i=0; i < x; i++)
            counts2.add(s[i]);
        y = 1;
        return 0;
    }
    @Fork(value = 1, warmups = 1)
    @Warmup(iterations = 1)
    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    public long _2_HashMapPut() {
        for (int i=0; i < x; i++)
            map.put(s[i], s[i]);
        y = 2;
```

```
66          return 0;
67      }
68
69      @Fork(value = 1, warmups = 1)
70      @Warmup(iterations = 1)
71      @Benchmark
72      @BenchmarkMode(Mode.AverageTime)
73      public long _3_TreeMapPut() {
74          for (int i=0; i < x; i++)
75              map2.put(s[i], s[i]);
76          y = 3;
77          return 0;
78      }
79
80      @TearDown(Level.Invocation)
81      public void doTearDown() {
82          switch (y) {
83              case 0:
84                  counts.clear();
85                  break;
86              case 1:
87                  counts2.clear();
88                  break;
89              case 2:
90                  map.clear();
91                     break;
92              case 3:
93                  map2.clear();
94                  break;
95          }
96      }
97
98      /**
99       * Main method to run the benchmark.
100      *
101      * @param args Command line arguments
102      * @throws Exception If an error occurs during benchmark execution
103      */
104     public static void main(String[] args) throws Exception {
105         org.openjdk.jmh.Main.main(args);
106     }
107
108 }
```

**Uzyskane czasy dodawania:**

HashSet:

```
# Benchmark: com.avenuecode.snippet.MyBenchmark._0_HashSetAdd

# Run progress: 0,00% complete, ETA 00:08:00
# Warmup Fork: 1 of 1
# Warmup Iteration   1: 1,333 s/op
Iteration   1: 0,952 s/op
Iteration   2: 0,906 s/op
```

```
Iteration   3: 1,027 s/op
Iteration   4: 1,045 s/op
Iteration   5: 1,048 s/op

# Run progress: 12,50% complete, ETA 00:07:29
# Fork: 1 of 1
# Warmup Iteration   1: 1,496 s/op
Iteration   1: 1,084 s/op
Iteration   2: 1,049 s/op
Iteration   3: 1,011 s/op
Iteration   4: 0,949 s/op
Iteration   5: 0,942 s/op
```

TreeSet:

```
# Benchmark: com.avenuecode.snippet.MyBenchmark._1_TreeSetAdd

# Run progress: 25,00% complete, ETA 00:06:25
# Warmup Fork: 1 of 1
# Warmup Iteration   1: 13,906 s/op
Iteration   1: 12,645 s/op
Iteration   2: 12,154 s/op
Iteration   3: 11,993 s/op
Iteration   4: 12,054 s/op
Iteration   5: 11,838 s/op

# Run progress: 37,50% complete, ETA 00:05:40
# Fork: 1 of 1
# Warmup Iteration   1: 15,035 s/op
Iteration   1: 12,804 s/op
Iteration   2: 12,588 s/op
Iteration   3: 12,710 s/op
Iteration   4: 12,403 s/op
Iteration   5: 12,679 s/op
```

HashMap:

```
# Benchmark: com.avenuecode.snippet.MyBenchmark._2_HashMapPut

# Run progress: 50,00% complete, ETA 00:04:43
# Warmup Fork: 1 of 1
# Warmup Iteration   1: 1,438 s/op
Iteration   1: 0,958 s/op
Iteration   2: 0,889 s/op
Iteration   3: 0,914 s/op
Iteration   4: 0,947 s/op
Iteration   5: 0,988 s/op

# Run progress: 62,50% complete, ETA 00:03:28
# Fork: 1 of 1
# Warmup Iteration   1: 1,312 s/op
Iteration   1: 0,927 s/op
Iteration   2: 0,945 s/op
Iteration   3: 0,916 s/op
Iteration   4: 0,931 s/op
Iteration   5: 0,897 s/op
```

TreeMap:

```
# Benchmark: com.avenuecode.snippet.MyBenchmark._3_TreeMapPut

# Run progress: 75,00% complete, ETA 00:02:17
# Warmup Fork: 1 of 1
# Warmup Iteration   1: 15,210 s/op
Iteration   1: 13,199 s/op
Iteration   2: 13,195 s/op
Iteration   3: 12,313 s/op
Iteration   4: 12,285 s/op
Iteration   5: 12,118 s/op

# Run progress: 87,50% complete, ETA 00:01:10
# Fork: 1 of 1
# Warmup Iteration   1: 15,400 s/op
Iteration   1: 12,677 s/op
Iteration   2: 12,756 s/op
Iteration   3: 12,658 s/op
Iteration   4: 12,662 s/op
Iteration   5: 12,568 s/op
```

JC 11-02: j.w. zdanie nie może być takie długie. Trzeba je podzielić. Poniższy program [Python]: mierzy czas dodawania String'ów do kolekcji Pythona. JC 11-02: Tutaj, skoro to wszystko jest przed pomiarem można by skrócić inicjalizację do s = [str(i) for i in range(10...)]

```
from time import perf_counter
import random
import sys
from sortedcontainers import SortedSet
from sortedcontainers import SortedDict

def my_function():
  s = [str(i) for i in range(10_000_000)]

  random.shuffle(s)

  counts = set()
  sorted_set = SortedSet()
  dict1 = {}
  dict2 = SortedDict()

  for j in range(0,5):

    counts.clear()

    # Start the stopwatch / counter
    t1_start = perf_counter()

    for i in range(0, 10_000_000):
      counts.add(s[i])

    # Stop the stopwatch / counter
    t1_stop = perf_counter()

```

```
    print("set: ", t1_stop-t1_start)


  for j in range(0,5):

    sorted_set.clear()

    # Start the stopwatch / counter
    t1_start = perf_counter()

    for i in range(0, 10_000_000):
      sorted_set.add(s[i])

    # Stop the stopwatch / counter
    t1_stop = perf_counter()

    print("SortedSet: ", t1_stop-t1_start)

  for j in range(0,5):

    dict1.clear()

    # Start the stopwatch / counter
    t1_start = perf_counter()

    for i in range(0, 10_000_000):
      dict1[s[i]] = s[i]

    # Stop the stopwatch / counter
    t1_stop = perf_counter()

    print("Dictionary: ", t1_stop-t1_start)

  for j in range(0,5):

    dict2.clear()

    # Start the stopwatch / counter
    t1_start = perf_counter()

    for i in range(0, 10_000_000):
      dict2[s[i]] = s[i]

    # Stop the stopwatch / counter
    t1_stop = perf_counter()

    print("SortedDict: ", t1_stop-t1_start)

my_function()
```

Uzyskaliśmy następujące czasy dodawania:

```
set:  2.207043123999938
set:  2.200626075999935
set:  2.1995758820003175
set:  2.197942086000239
```

```
SortedSet:  24.410430537999673
SortedSet:  25.729037049
SortedSet:  24.10598873800018
SortedSet:  24.190924873000313
SortedSet:  24.490848085999914
Dictionary:  4.109121431999938
Dictionary:  4.080353903999821
Dictionary:  4.059659220999947
Dictionary:  4.15668830300001
Dictionary:  4.085921346999839
SortedDict:  28.95898643199962
SortedDict:  28.782305533
SortedDict:  28.483486593999714
SortedDict:  28.313258532999953
SortedDict:  28.271739520999745
```

Przyjrzyjmy się jeszcze raz wynikom dla Javy (op oznacza tu jedną iterację, czyli wstawienie 10 000 000 elementów): `JC 11-02: s/op brzmi jak sekund na operację... a to chyba nie na 1 operację, albo nie s a ns... albo trzeba napisać, że op oznacza tu jedną iterację, czyli milion wstawień czy ileś tam.`

```
Benchmark                    Mode  Cnt    Score   Error  Units
MyBenchmark._0_HashSetAdd    avgt    5    1,007 +/- 0,238   s/op
MyBenchmark._1_TreeSetAdd    avgt    5   12,637 +/- 0,584   s/op
MyBenchmark._2_HashMapPut    avgt    5    0,923 +/- 0,070   s/op
MyBenchmark._3_TreeMapPut    avgt    5   12,664 +/- 0,257   s/op
```

Powyższe czasy dodawania elementów do kolekcji zgodne są ze złożonością czasową tych operacji. `JC 11-02: hm... o tym, że są "zgodne ze złożonością"to można by mówić, gdybyśmy porównywali kolekcje większe i mniejsze... Może napisać Różnice w czasach mogą wytłumaczyć różnice w złożoności czasowej tych operacji.` Różnice w czasach mogą wytłumaczyć różnice w złożoności czasowej tych operacji.

```
collection  operation  complexity
HashSet     add        O(1)
TreeSet     add        O(log N)
HashMap     put        O(1)
TreeMap     put        O(log N)

set         add        O(1) (average case)
SortedSet   add        O(log N)
Dictionary  setitem    O(1)
SortedDict  setitem    O(log N)
```

`JC 11-02: wyniki testów trzeba skomentować. Choćby banalnie, że coś tam troszkę szybsze.` Operacja dodawania ma w rozpatrywanych kolekcjach Javy i ich odpowiednikach w Pythonie taką samą złożoność czasową. W związku z tym różnice w czasie wykonania różnią się jedynie o stałą. Java jest szybsza. W Pythonie implementacje struktur SortedSet i SortedDict łączą w sobie cechy tablicy haszującej i drzewa zrównoważonego. W efekcie takie struktury są nieco wolniejsze od ich analogów w Javie (np. TreeSet, TreeMap), które opierają się na czystych drzewach zrównoważonych bez dodatkowej warstwy haszowania.

### 8.4.3. Sprawdzanie czy kolekcja zawiera Stringi

Korzystając z programu podobnego do używanego do szacowania czasu dodawania String'ów do kolekcji [Java] [Python], mierzy się czas sprawdzania czy String należy do kolekcji. Stringi zostały umieszczone w kolekcjach przed rozpoczęciem pomiarów. `JC 11-02: Tu trzeba napisać, że te stringi były w kolekcji, albo że nie były, albo że pół na pół czy coś. Czasy wyszukiwania mogą dość mocno zależeć od tego aspektu.`

**Wyniki dla Javy:**

```
Benchmark                              Mode  Cnt    Score    Error  Units
MyBenchmark._0_HashSetContains         avgt    5    0,594 +/- 0,024   s/op
MyBenchmark._1_TreeSetContains         avgt    5   11,100 +/- 0,154   s/op
MyBenchmark._2_HashMapContainsKey      avgt    5    0,536 +/- 0,017   s/op
MyBenchmark._3_TreeMapContainsKey      avgt    5   11,472 +/- 0,297   s/op
```

**Wyniki dla Pythona:**

```
set:  2.9993871530000433
set:  2.9143904570000814
set:  2.880181250000078
set:  2.879620507999789
set:  2.8789269859998967
SortedSet:  3.3288162799999554
SortedSet:  3.3365371639997647
SortedSet:  3.3687667489998603
SortedSet:  3.368098517999897
SortedSet:  3.353463620000184
Dictionary:  4.0150994229998105
Dictionary:  4.100978073999613
Dictionary:  4.031463173999782
Dictionary:  4.091581018000397
Dictionary:  4.087747910999951
SortedDict:  4.311155410000083
SortedDict:  4.318198027000108
SortedDict:  4.194814053999835
SortedDict:  4.197658074999708
SortedDict:  4.210030474999712
```

Powyższe czasy sprawdzania czy elementy należą do kolekcji zgodne są ze złożonością czasową tych operacji.

```
collection  operation   complexity
HashSet     contains    O(1)
TreeSet     contains    O(log N)
HashMap     containsKey O(1)
TreeMap     containsKey O(log N)

set         in          O(1) (average case)
SortedSet   in          O(1)
Dictionary  in          O(1)
SortedDict  in          O(1)
```

JC 11-02: j.w. W dodatku należy napisać, że najwyraźniej SortedSet i SortedDict zawierają w sobie tablicę hashującą oraz strukturę opartą o kolejność (zapewne drzewo zrównoważone), stąd sprawdzanie w czasie stałym i wstawianie w czasie log N, trochę wolniejsze od analogicznego w Javie. Jako że w Pythonie implementacje struktur SortedSet i SortedDict łączą w sobie cechy tablicy haszującej i drzewa zrównoważonego, sprawdzanie istnienia elementu może być realizowane w czasie stałym (O(1)) dzięki części hashującej. Z tego względu sprawdzanie istnienia elementu jest szybsze w tych kolekcjach w Pythonie niż w Javie.

### 8.4.4. Usuwanie String'ów z kolekcji

Korzystając z programu podobnego do używanego do szacowania czasu dodawania String'ów do kolekcji [Java] [Python], mierzy się czas usuwania String'ów z kolekcji.

**Wyniki dla Javy:**

```
Benchmark                       Mode  Cnt    Score    Error  Units
MyBenchmark._0_HashSetRemove    avgt    5    1,288 +/- 0,153   s/op
MyBenchmark._1_TreeSetRemove    avgt    5   11,823 +/- 2,925   s/op
MyBenchmark._2_HashMapRemove    avgt    5    1,244 +/- 0,052   s/op
MyBenchmark._3_TreeMapRemove    avgt    5   11,110 +/- 1,195   s/op
```

**Wyniki dla Pythona:**

```
set:  2.962287095999727
set:  2.93804236699998
set:  2.975594524999906
set:  2.9539603660000466
set:  2.9535686730000634
SortedSet:  14.213103547999708
SortedSet:  14.19680624900002
SortedSet:  14.200566351999896
SortedSet:  14.25538980600004
SortedSet:  14.142081440999846
Dictionary:  4.246978401999968
Dictionary:  4.244396434999999
Dictionary:  4.245342547999826
Dictionary:  4.243117217999952
Dictionary:  4.247132589999637
SortedDict:  16.82463326800007
SortedDict:  16.839339327999824
SortedDict:  16.89218398200046
SortedDict:  17.11353439100003
SortedDict:  16.469016367999757
```

Powyższe czasy usuwania elementów z kolekcji zgodne są ze złożonością czasową tych operacji.

```
collection  operation   complexity
HashSet     remove      O(1)
TreeSet     remove      O(log N)
HashMap     remove      O(1)
TreeMap     remove      O(log N)
```

```
set          discard      O(1) (average case)
SortedSet    discard      O(log N)
Dictionary   pop          O(1)
SortedDict   pop          O(log N)
```

JC 11-02: `j.w. brakuje podsumowania` Operacja usuwania ma w rozpatrywanych kolekcjach Javy i ich odpowiednikach w Pythonie taką samą złożoność czasową. W związku z tym różnice w czasie wykonania różnią się jedynie o stałą. Java jest szybsza.


Java
9
python

# Rozdział 9

# Porównanie czasu działania programów

## 9.1. Narzędzia

W tym rozdziale korzystamy dodatkowo z narzędzia:

- runexec - program dostarczany wraz z frameworkiem BenchExec, który pozwala na mierzenie zużycia zasobów, podobnie jak polecenie time, ale z dokładniejszym pomiarem czasu działania i z pomiarem zużycia pamięci

## 9.2. The Computer Language Benchmarks Game

W tej sekcji porównujemy czas działania wybranych programów w Javie i Pythonie pochodzących ze strony [CLBG]. Są to najszybsze wersje programów w Javie i Pythonie nie korzystające (z jednym wyjątkiem) z zewnętrznych bibliotek.

### 9.2.1. PIDIGITS

Program PIDIGITS oblicza n pierwszych cyfr liczby Pi. Następnie wypisuje 10 kolejnych cyfr liczby Pi w każdej linii. Dla pomiarów za bazowe n przyjęto liczbę 10 000. Dokładniejszy opis: [8]

#### Java

W przypadku Javy walltime programu [Java] przy wykorzystaniu polecenia runexec wyniósł 5,93 s. Komplet wyników:

```
starttime=2024-12-20T18:01:42.080860+01:00
returnvalue=0
walltime=5.927161733000048s
cputime=6.890361028s
cputime-cpu0=0.079110021s
cputime-cpu1=0.032669473s
cputime-cpu10=0.063455014s
cputime-cpu11=5.658510060s
cputime-cpu2=0.260540423s
cputime-cpu3=0.042436937s
cputime-cpu4=0.050863610s
```

```
cputime-cpu5=0.151805614s
cputime-cpu6=0.057804142s
cputime-cpu7=0.403615176s
cputime-cpu8=0.037124300s
cputime-cpu9=0.052426258s
memory=383864832B
blkio-read=0B
blkio-write=0B
```

### Python

W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wyniósł 2,86 s. Komplet wyników:

```
starttime=2024-12-20T18:03:53.441213+01:00
returnvalue=0
walltime=2.8576857300001848s
cputime=2.857317687s
cputime-cpu3=2.102434451s
cputime-cpu9=0.754883236s
memory=4349952B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wolniejsza od Pythona. Może to wynikać z wykorzystania do obliczeń liczb typu BigInteger, które jak pokazaliśmy działają wolniej niż duże liczby w Pythonie.

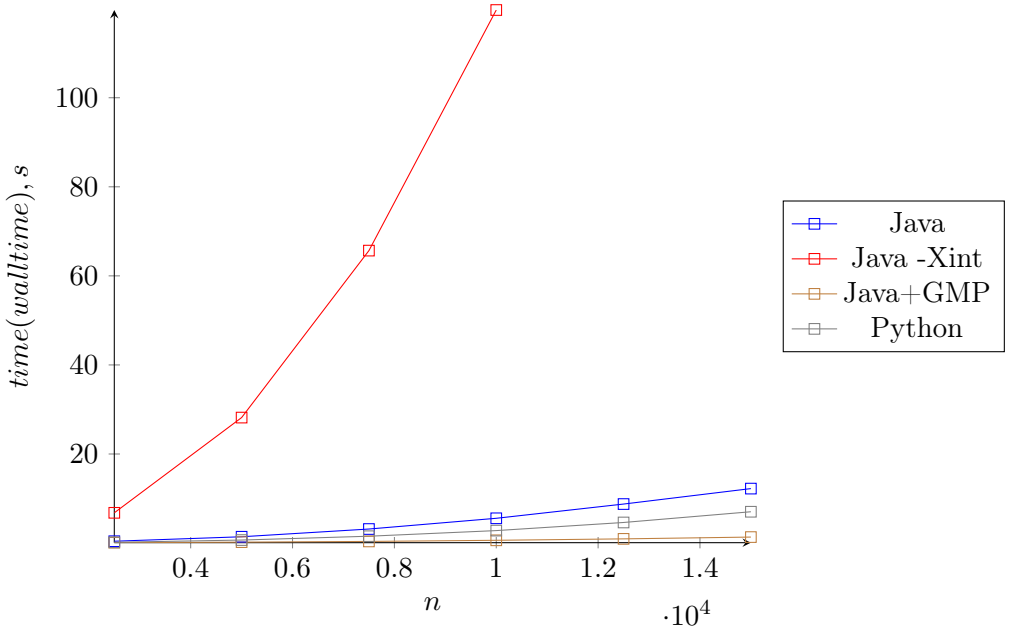
Rozwiązaniem tego problemu może być wykorzystanie bibioteki GMP, która znacząco przyspiesza obliczenia arytmetyczne [Java #3].

### Java+GMP

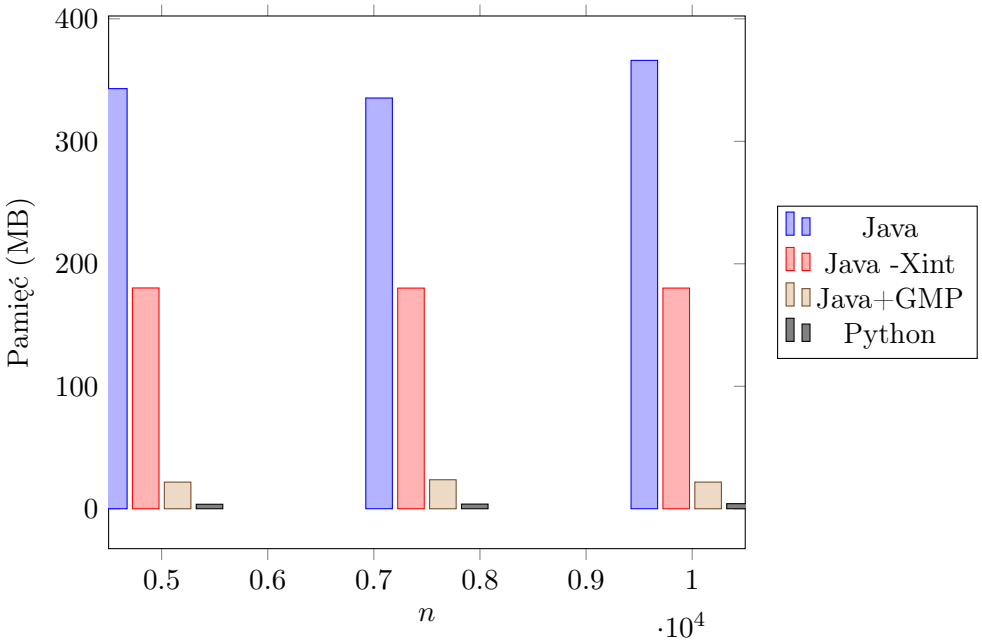
W przypadku użycia Java+GMP walltime programu przy wykorzystaniu polecenia runexec wyniósł 0,601 s. Komplet wyników:

```
starttime=2025-11-04T19:08:31.946039+01:00
returnvalue=0
walltime=0.6010265519998939s
cputime=0.598652273s
cputime-cpu0=0.000061606s
cputime-cpu10=0.000102853s
cputime-cpu11=0.000037550s
cputime-cpu2=0.006865986s
cputime-cpu3=0.565805454s
cputime-cpu4=0.014374581s
cputime-cpu5=0.011007058s
cputime-cpu6=0.000073868s
cputime-cpu7=0.000139712s
cputime-cpu8=0.000059411s
cputime-cpu9=0.000124194s
memory=22831104B
blkio-read=0B
blkio-write=0B
```

Poniżej znajduje się wykres czasu działania algorytmu w Javie i Pythonie dla różnych wartości n.




Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




Jak widać Java potrzebuje do wykonania programu znacznie większej ilości pamięci niż Python niezależnie od wejściowej wartości n.

JC 11-02: Z ciekawości, jeśli to nie jest duży problem, można by dołączyć wyniki dla Java+GMP, zwłaszcza jeśli w innych miejscach używamy np. Python + numpy... Dobrze byłoby dać podsumowanie całego testu i "spekulacje"na temat przyczyn takich wyników po wszystkich pomiarach (szybkość i pamięć) - może być zamiast tych po pomiarach szybkości.

#### Podsumowanie

Podsumujmy co kryje się za poszczególnymi elementami legendy.

- Java — program w Javie, korzystający z domyślnego kompilatora JIT (Just-In-Time).
- Java -Xint — wymuszenie, aby JVM interpretował kod bajtowy bez użycia JIT. Powoduje to znaczne spowolnienie wykonania, ale jest przydatne do debugowania lub porównania wydajności. („-Xint" = tryb interpretowany).
- Java+GMP — integracja z GMP (GNU Multiple Precision Arithmetic Library) w Javie poprzez wrapper typu Java-GMP), co umożliwia szybsze operacje na dużych liczbach niż przy użyciu wbudowanej klasy BigInteger.
- Python — uruchomienie kodu Pythona w trybie interpretowanym

Jak widać Java jest szybsza od Pythona jedynie w przypadku integracji z GMP.

Zużycie pamięci jest zbliżone do stałego (nie ma znaczenia wartość n).

### 9.2.2. binary-trees

Ten program alokuje wpierw głębokie drzewo binarne aby rozciągnąć pamięć. Następnie alokuje drzewo binarne, które istnieje aż do końca działania programu. Pozostała część wykonania polega na alokowaniu i dealokowaniu wielu drzew przy jednoczesnym zliczaniu liczby ich wierzchołków.

Podstawową liczbę dla pomiarów stanowi 21. Liczba ta jest głębokością długotrwale istniejącego drzewa binarna, które pozostaje w pamięci, podczas gdy inne drzewa są tworzone i usuwane. JC 11-02: Jednak należałoby choć trochę napisać czym jest ta "podstawowa liczba"- rozmiarem drzew? liczbą? głębokością? Dokładniejszy opis: [9].

#### Java

W przypadku Javy walltime programu [Java #7] przy wykorzystaniu polecenia runexec wyniósł 2,28 s. Komplet wyników:

```
starttime=2024-12-20T19:58:59.059904+01:00
returnvalue=0
walltime=2.2797591759999705s
cputime=16.777071931s
cputime-cpu0=1.853680797s
cputime-cpu1=1.892253950s
cputime-cpu10=1.848870294s
cputime-cpu11=2.032843361s
cputime-cpu2=0.928579037s
cputime-cpu3=0.677434905s
cputime-cpu4=1.476493191s
cputime-cpu5=1.654540192s
cputime-cpu6=0.403547661s
cputime-cpu7=0.814800178s
cputime-cpu8=1.478957829s
cputime-cpu9=1.715070536s
memory=2565365760B
blkio-read=94208B
blkio-write=0B
```

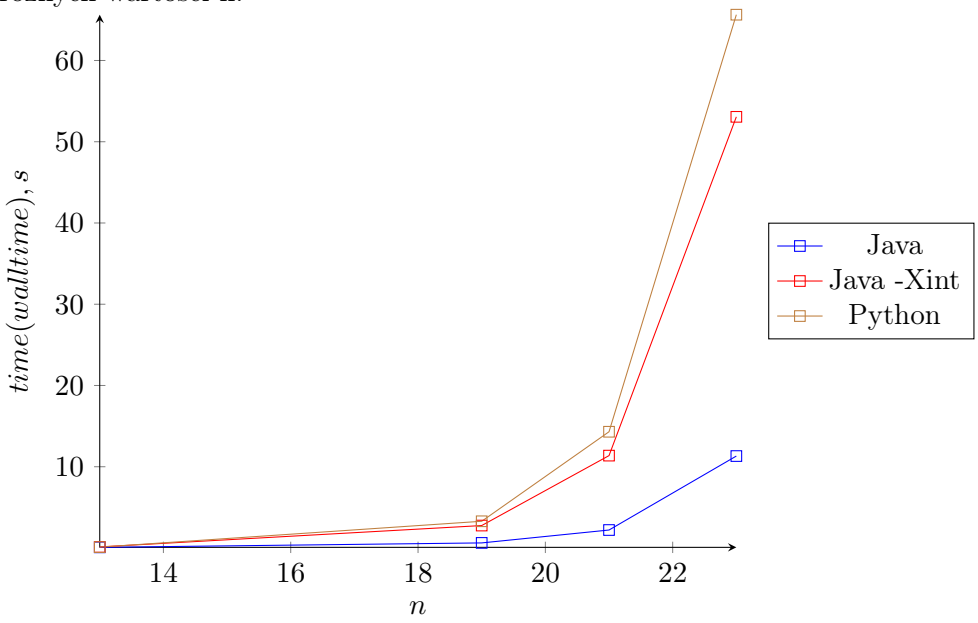
### Python

W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wyniósł 15,6 s. Komplet wyników:

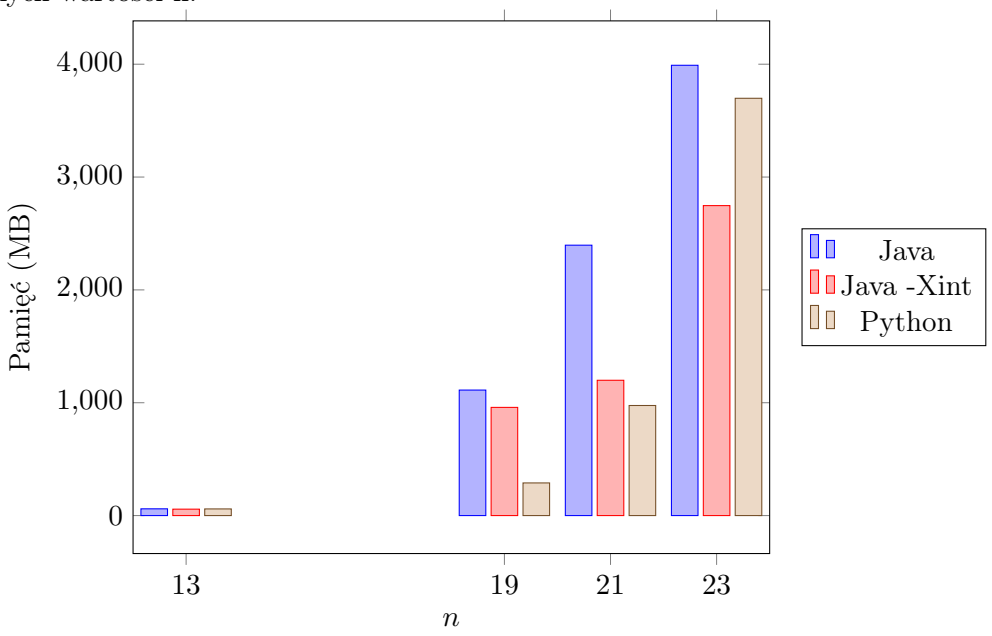
```
starttime=2024-12-20T19:59:42.184918+01:00
returnvalue=0
walltime=15.58560308199958s
cputime=142.584223976s
cputime-cpu0=11.524329568s
cputime-cpu1=11.418168383s
cputime-cpu10=11.543249594s
cputime-cpu11=11.336778209s
cputime-cpu2=11.776000717s
cputime-cpu3=11.450827422s
cputime-cpu4=14.548266545s
cputime-cpu5=11.834461877s
cputime-cpu6=11.801861248s
cputime-cpu7=11.703029944s
cputime-cpu8=11.800562881s
cputime-cpu9=11.846687588s
memory=1025269760B
blkio-read=4157440B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n = 21 jest to 15,6 s/2,28 s = 6,84 raza szybciej.

Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości n.



Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




JC 11-02: Potrzebne podsumowania i próba wytłumaczenia wyników...

**Podsumowanie**

Spróbujmy wytłumaczyć czemu Java jest szybsza od Pythona w tym teście.

- Java może skompilować i zoptymalizować gorące ścieżki kodu (JIT), co daje dużą przewagę, gdy operujemy wielokrotnie na strukturze danych.
- Java ma lepszą wydajność alokacji małych obiektów i zarządzania pamięcią przy intensywnym tworzeniu i niszczeniu struktur a w tym benchmarku jest tworzonych i niszczonych wiele drzew, również małych.
- Python ponosi koszt dynamiczności: typów, interpretacji, alokacji, co kumuluje się w przypadku benchmarku z wieloma węzłami drzewa.

Python zużywa wyraźnie mniej pamięci dla wartości n równej podstawowej liczby dla pomiarów (21) lub trochę mniejszej (19). Dla większych sytuacja poprawia się na korzyść Javy.

### 9.2.3. fannkuch-redux

W przypadku tego programu rozpatrywane są wszystkie permutacje liczb 1,..,n dla danego n. Dla każdej permutacji rozpatrywany jest jej pierwszy element, nazwijmy go x. Następnie porządek x pierwszych elementów jest odwracany. Procedura ta jest powtarzana aż pierwszym elementem stanie się 1. Zapamiętywana jest maksymalna liczba odwróceń spośród wszystkich permutacji. Obliczana jest również wartość checksum aby zagwarantować poprawność implementacji.

Dla pomiarów za bazowe n przyjęto liczbę 12.

JC 11-02: Czy to N to n z pierwszego zdania? Dokładniejszy opis: [10].

### Java

W przypadku Javy walltime programu [Java] przy wykorzystaniu polecenia runexec wyniósł 3,56 s. Komplet wyników:

```
starttime=2024-12-24T12:38:48.923297+01:00
returnvalue=0
walltime=3.559818760999974s
cputime=40.68653424s
cputime-cpu0=3.422547344s
cputime-cpu1=3.466155625s
cputime-cpu10=3.316182076s
cputime-cpu11=3.447459427s
cputime-cpu2=3.412224327s
cputime-cpu3=3.411831401s
cputime-cpu4=3.314061044s
cputime-cpu5=3.302599848s
cputime-cpu6=3.352211472s
cputime-cpu7=3.460125386s
cputime-cpu8=3.293854854s
cputime-cpu9=3.487281436s
memory=31907840B
blkio-read=0B
blkio-write=0B
```

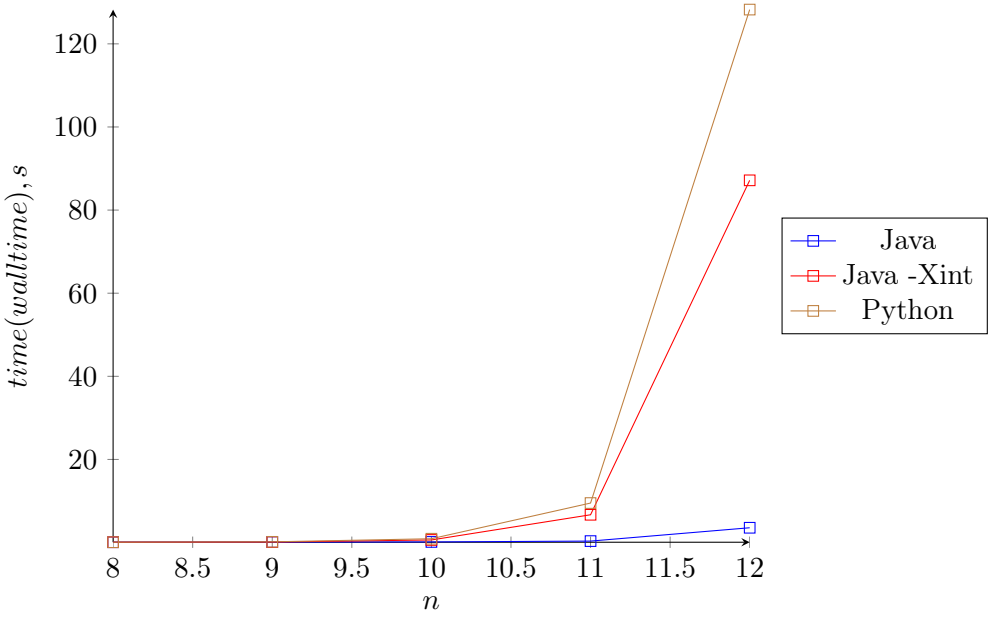
### Python

W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wyniósł 128 s. Komplet wyników:

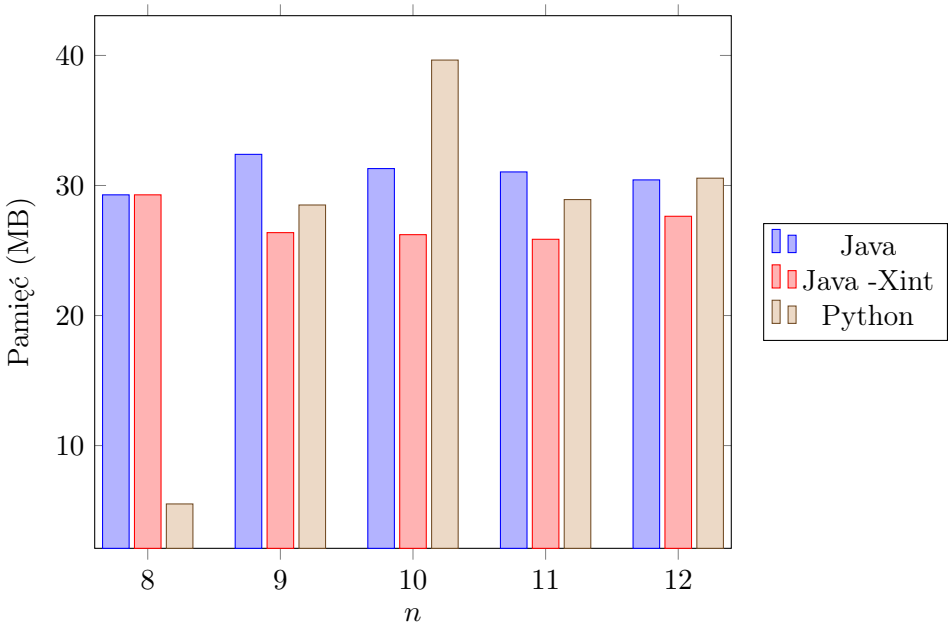
```
starttime=2024-12-24T12:41:12.741818+01:00
returnvalue=0
walltime=128.24714689400002s
cputime=1487.220224683s
cputime-cpu0=120.576718880s
cputime-cpu1=125.762417764s
cputime-cpu10=122.406405365s
cputime-cpu11=124.644218967s
cputime-cpu2=124.072275975s
cputime-cpu3=124.788649866s
cputime-cpu4=122.481056674s
cputime-cpu5=124.992362780s
cputime-cpu6=121.394777709s
cputime-cpu7=126.898999067s
cputime-cpu8=123.979731513s
cputime-cpu9=125.222610123s
memory=32047104B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n = 12 jest to 128 s/3,56 s = 36 raza szybciej.

Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości n.




Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




### Podsumowanie

Test *fannkuch-redux* jest typowym zadaniem **CPU-bound** — większość czasu wykonania spędza na intensywnych obliczeniach i operacjach na tablicach. Python jest w tym teście znacznie wolniejszy niż Java lub C, ponieważ każda operacja wymaga obsługi dynamicznych typów i obiektów. Java i C wykorzystują prymitywne typy oraz optymalizacje JIT/kompilatora, co pozwala znacząco przyspieszyć wykonanie. W efekcie Python może być kilkadziesiąt

razy wolniejszy w tego typu zadaniach. Widać, że problem mjest przez Javę z wyłączonym JIT rozwiązywany niewiele szybciej niż przez Pythona. Java z włączonym JIT rozwiązuje zaś problem bardzo szybko, z ogromną przewagą w stosunku do Pythona. Stąd wniosek, że decydującą rolę w szybkości rozwiązywania tego problemu w przypadku Javy odgrywa JIT.

Zużycie pamięci w tym teście jest zbliżone do stałego.

### 9.2.4. N-body

Modeluje orbity planet z grupy Jowian (Jowisz, Saturn, Uran i Neptun), używając tego samego prostego integratora symplektycznego.

Dla pomiarów za bazowe n przyjęto liczbę 50 000 000. Liczba n oznacza liczbę kroków symulacji. `JC 11-02: Tym razem zupełnie nie wiadomo czym jest n...` `JC 11-02: Proszę stałe liczbowe w tekście uczytelnić j.w. Podobnie w Pythonie - tam też można używać podkreśleń: 10_000` Dokładniejszy opis: [11].

#### Java

W przypadku Javy walltime programu [Java #4] przy wykorzystaniu polecenia runexec wyniósł 4,22 s. Komplet wyników:

```
starttime=2024-12-25T11:53:57.697005+01:00
returnvalue=0
walltime=4.217370437999989s
cputime=4.218599014s
cputime-cpu10=0.008972176s
cputime-cpu11=0.005400429s
cputime-cpu2=0.022193033s
cputime-cpu3=4.148934257s
cputime-cpu4=0.003043800s
cputime-cpu5=0.016407312s
cputime-cpu6=0.009972404s
cputime-cpu7=0.000178814s
cputime-cpu8=0.000066114s
cputime-cpu9=0.003430675s
memory=25059328B
blkio-read=544768B
blkio-write=0B
```

#### Python

W przypadku Pythona walltime programu [Python 3] przy wykorzystaniu polecenia runexec wyniósł 322 s. Komplet wyników:

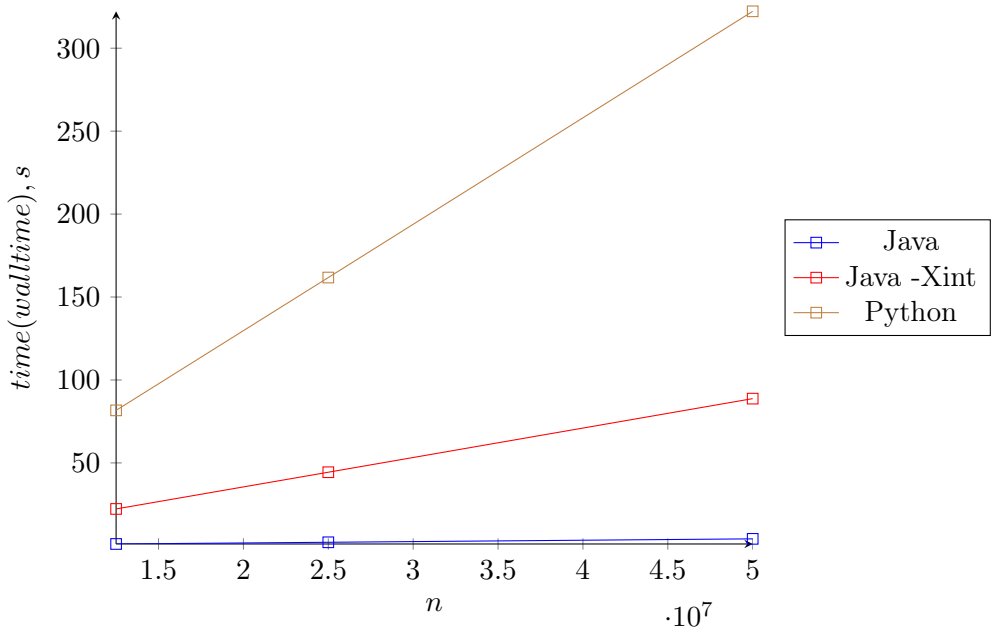
```
starttime=2024-12-25T11:57:15.773324+01:00
returnvalue=0
walltime=322.2353108489999s
cputime=322.226826032s
cputime-cpu0=97.789804078s
cputime-cpu10=0.012211654s
cputime-cpu11=72.973624307s
cputime-cpu3=1.512004426s
cputime-cpu5=22.195560776s
cputime-cpu6=127.743620791s
```

```
memory=3559424B
blkio-read=53248B
blkio-write=0B
```

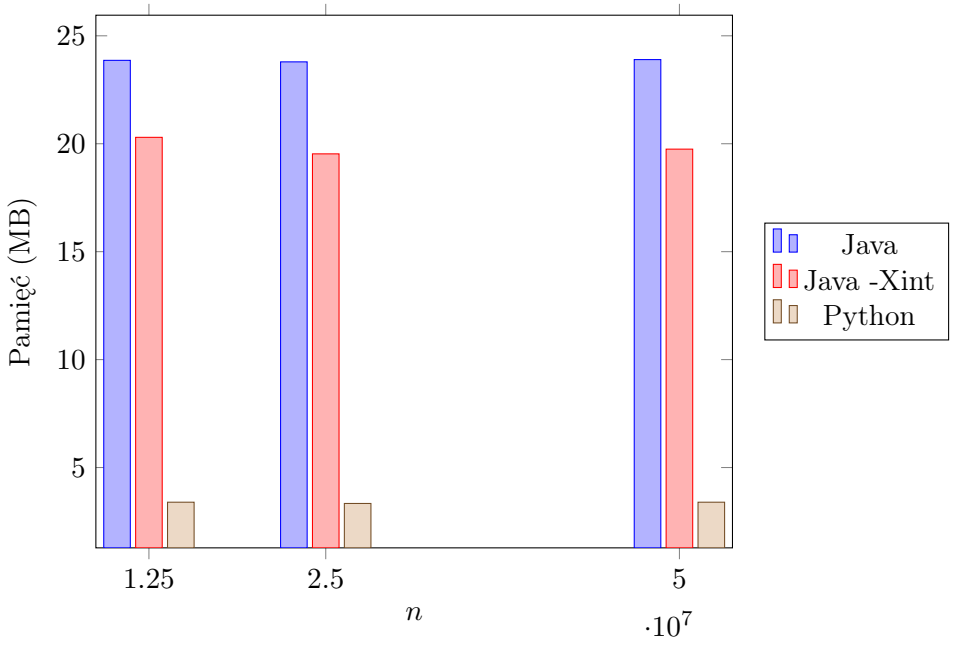
Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n = 50 000 000 jest to 322 s/4,22 s = 76,3 raza szybciej.

Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości n.

JC 11-02: Znowu: raz duże N, raz małe n. Proszę to ujednolicić.




Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.



#### Podsumowanie

W teście N-body Python jest znacznie wolniejszy niż Java, ponieważ zadanie jest bardzo obliczeniowo intensywne — dużo pętli i operacji na liczbach. Python wykonuje je z narzutem dynamicznych typów i obiektów, podczas gdy Java używa prymitywów i JIT, co daje ogromną przewagę wydajnościową.

Zużycie pamięci w tym teście jest stałe. Liczba n oznacza ilość kroków symulacji, nie wpływa ona na zużycie pamięci.

### 9.2.5. fasta

- generuje sekwencje DNA, kopiując z danej sekwencji
- generuje sekwencje DNA poprzez ważony losowy wybór z 2 alfabetów

| **Typ sekwencji** | **Charakter** | **Długość** |
|---|---|---|
| ALU repeat | powtarzający się wzorzec DNA | $2 \times n$ |
| IUB random | losowa sekwencja z rozkładem wag (IUB) | $3 \times n$ |
| Homo sapiens random | losowa sekwencja DNA z innym rozkładem wag | $5 \times n$ |
| **Łącznie** | - | $10 \times n$ |

Tabela 9.1: Podsumowanie typów sekwencji w programie fasta (Benchmarksgame).

`JC 11-02: Nie bardzo rozumiem co tu jest do roboty? Po prostu generuje losowy ciąg na wyjście? To pewnie większość czasu zajmuje IO, a nie losowanie... Rozumiem, że 250..0 to rozmiar wygenerowanego DNA? Proszę to napisać!`

Dla pomiarów za argument wejściowy n przyjęto 25 000 000, czyli rozmiar wygenerowanego DNA to $10 \times n = 250\,000\,000$. Dokładniejszy opis: [12].

#### Java

W przypadku Javy walltime programu [Java #6] przy wykorzystaniu polecenia runexec wyniósł 1.16 s. Komplet wyników:

```
starttime=2024-12-25T19:50:16.432118+01:00
returnvalue=0
walltime=1.1648875109999608s
cputime=4.734007739s
cputime-cpu0=0.547533909s
cputime-cpu1=0.477176959s
cputime-cpu10=0.005104631s
cputime-cpu11=0.128299400s
cputime-cpu2=0.377512691s
cputime-cpu3=0.578542008s
cputime-cpu4=0.240216090s
cputime-cpu5=0.852080439s
cputime-cpu6=0.441369131s
cputime-cpu7=0.478429609s
cputime-cpu8=0.533055693s
cputime-cpu9=0.074687179s
memory=298942464B
blkio-read=0B
blkio-write=0B
```

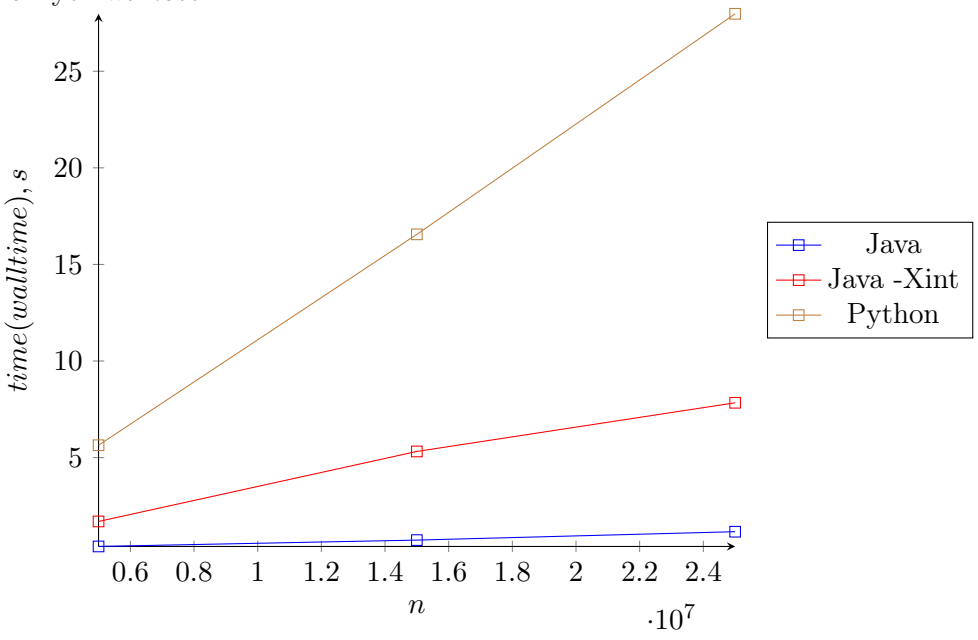
### Python

W przypadku Pythona walltime programu [Python 3 #5] przy wykorzystaniu polecenia runexec wyniósł 28 s. Komplet wyników:

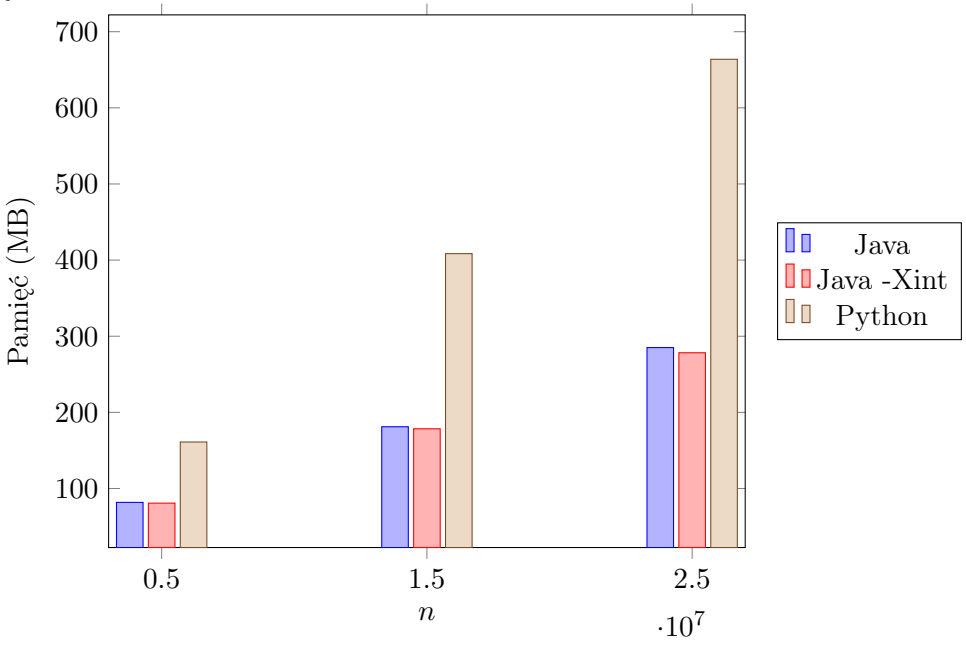
```
starttime=2024-12-25T19:56:00.767456+01:00
returnvalue=0
walltime=27.973746784000014s
cputime=49.00618006s
cputime-cpu0=2.295515155s
cputime-cpu1=11.375523946s
cputime-cpu10=5.328212206s
cputime-cpu11=13.194317161s
cputime-cpu2=2.877192826s
cputime-cpu3=1.743911116s
cputime-cpu4=0.995536218s
cputime-cpu5=4.384794556s
cputime-cpu6=2.489412690s
cputime-cpu7=2.882490619s
cputime-cpu8=1.439273567s
memory=696020992B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla argumentu 100000001 jest to 28 s/1,16 s = 24,1 raza szybciej.

Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości n.



Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




**Podsumowanie**

Python w tym teście odstaje, ponieważ zadanie generuje i zapisuje bardzo dużą ilość danych w pętli — co wymaga bardzo wydajnego dostępu do pamięci, buforowania i minimalnych narzutów języka. Java jest w tym lepsza, stąd duża różnica.

W benchmarku **fasta** Python zużywa więcej pamięci niż Java, ponieważ:

- implementacja Pythona używa bardziej pamięciochłonnych struktur (listy, kopiowanie danych),
- Java operuje na tablicach prymitywnych i minimalizuje dodatkowe alokacje,
- przy dużych danych narzut języka staje się mniej istotny, a główną rolę odgrywają struktury danych.

### 9.2.6. reverse-complement

Znając sekwencję zasad jednej nici DNA, od razu znamy sekwencję nici DNA, która się z nią zwiąże, nić ta nazywa się odwrotnym dopełnieniem.

Program czyta ze standardowego wejścia plik w formacie FASTA i wypisuje na standardowe wyjście id, opis i odwrotne dopełnienie w formacie FASTA.

Dane wejściowe zostały wygenerowane za pomocą programu fasta. Długość sekwencji wynosi 100000001. Dokładniejszy opis: [13].

`JC 11-02: No tu to chyba wyłącznie jest IO...`

### Java

W przypadku Javy walltime programu [Java #6] przy wykorzystaniu polecenia runexec wyniósł 2,37 s. Komplet wyników:

```
starttime=2024-12-25T13:36:16.654139+01:00
returnvalue=0
walltime=2.366201705000094s
cputime=2.680236005s
cputime-cpu0=0.000092554s
cputime-cpu1=0.000087434s
cputime-cpu10=0.001456222s
cputime-cpu11=0.001197352s
cputime-cpu2=0.000078177s
cputime-cpu3=2.040724949s
cputime-cpu4=0.016653725s
cputime-cpu5=0.013650926s
cputime-cpu6=0.000079108s
cputime-cpu7=0.601405798s
cputime-cpu8=0.000336001s
cputime-cpu9=0.004473759s
memory=3115700224B
blkio-read=0B
blkio-write=0B
```

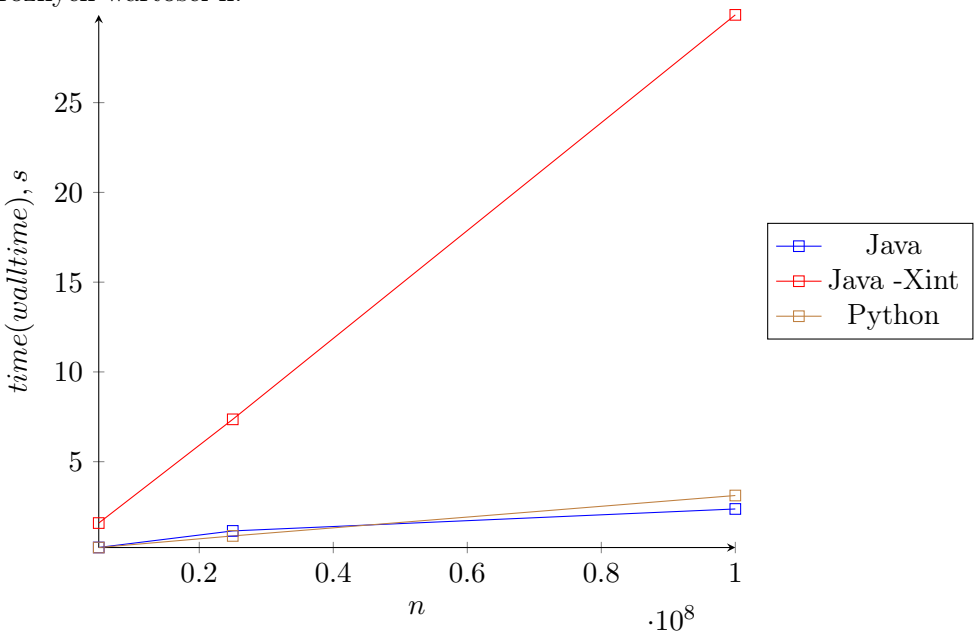
### Python

W przypadku Pythona walltime programu [Python 3 #5] przy wykorzystaniu polecenia runexec wyniósł 3,11 s. Komplet wyników:

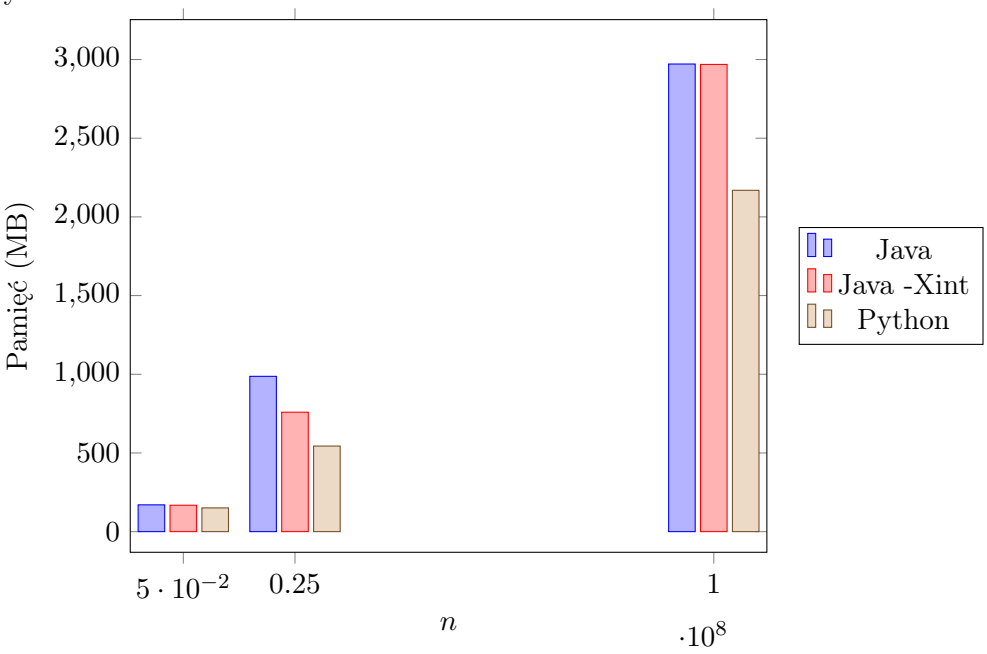
```
starttime=2024-12-25T13:38:31.845847+01:00
returnvalue=0
walltime=3.1158195779999005s
cputime=5.790129265s
cputime-cpu0=1.019473178s
cputime-cpu1=0.567983424s
cputime-cpu10=0.009591560s
cputime-cpu2=0.000162765s
cputime-cpu3=0.007990056s
cputime-cpu5=0.053840911s
cputime-cpu6=1.809747270s
cputime-cpu7=0.583753191s
cputime-cpu8=0.001700389s
cputime-cpu9=1.735886521s
memory=2274304000B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest stosunkowo niewiele szybsza od Pythona. Dla n = 100000001 jest to 3,11 s/2,37 s = 1,31 raza szybciej.

Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości n.




Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




**Podsumowanie:**

W teście *reverse-complement* różnica między Javą a Pythonem jest mniejsza, ponieważ zadanie polega głównie na przetwarzaniu tekstu i operacjach I/O, a nie na intensywnych obliczeniach. Python wykonuje te operacje w szybkim kodzie C, więc mimo że Java (dzięki JIT) pozostaje szybsza, różnica nie jest tak duża jak w testach numerycznych.

W tym programie zużycie pamięci przez Javę i Pythona jest podobne. Dane dominują nad narzutem języka/środowiska, więc różnice w zużyciu pamięci między językami mogą są niewielkie — główną część pamięci stanowią dane same w sobie.

### 9.2.7. mandelbrot

Kreśli zbiór Mandelbrota [-1.5-i,0.5+i] na bitmapie n na n. Zapisuje wynik bajt po bajcie w pliku PBM. Dla pomiarów za argument wejściowy przyjęto 16000. Dokładniejszy opis: [14].

#### Java

W przypadku Javy walltime programu [Java #2] przy wykorzystaniu polecenia runexec wyniósł 1,13 s. Komplet wyników:

```
walltime=1.1342887529999643s
cputime=11.93891101s
cputime-cpu0=1.019560255s
cputime-cpu1=0.989544058s
cputime-cpu10=0.975318721s
cputime-cpu11=1.044600389s
cputime-cpu2=0.983189828s
cputime-cpu3=0.978190282s
cputime-cpu4=0.990735071s
cputime-cpu5=0.997777040s
cputime-cpu6=0.987225069s
cputime-cpu7=1.012641294s
cputime-cpu8=0.981878383s
cputime-cpu9=0.978250620s
memory=112623616B
blkio-read=0B
blkio-write=0B
```

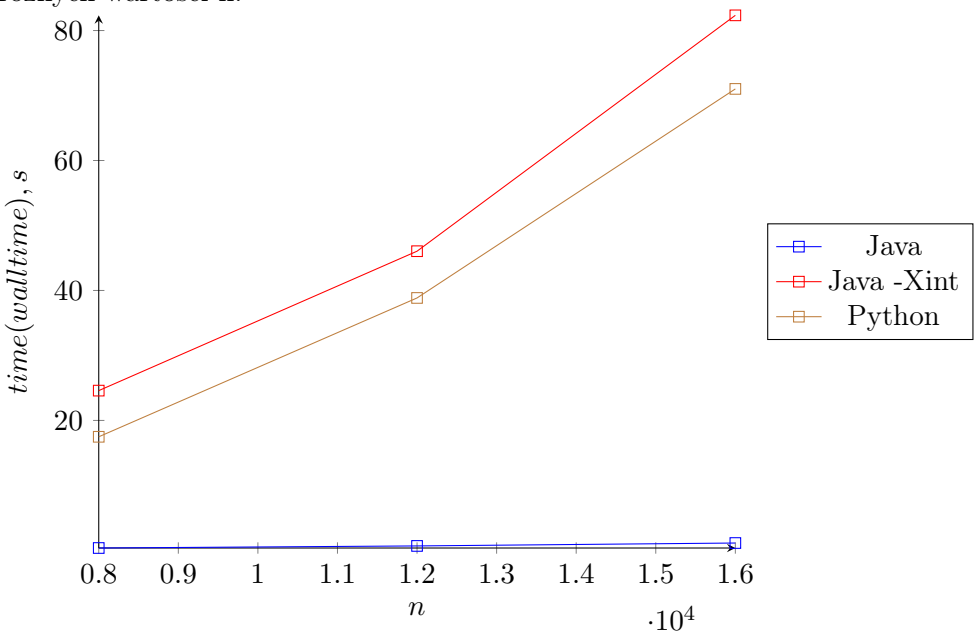
#### Python

W przypadku Pythona walltime programu [Python 3 #7] przy wykorzystaniu polecenia runexec wyniósł 71 s. Komplet wyników:

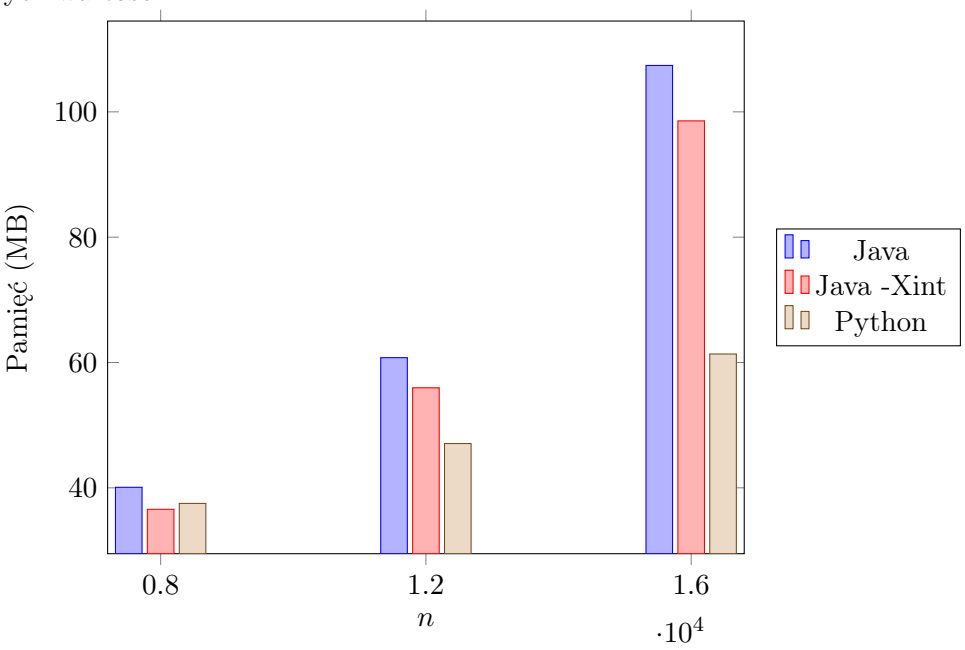
```
starttime=2024-12-25T20:12:15.167569+01:00
returnvalue=0
walltime=71.01652564599999s
cputime=843.977451867s
cputime-cpu0=70.294594650s
cputime-cpu1=70.500735403s
cputime-cpu10=69.902568873s
cputime-cpu11=70.183822083s
cputime-cpu2=70.520069691s
cputime-cpu3=70.074994223s
cputime-cpu4=70.495964229s
cputime-cpu5=70.273095224s
cputime-cpu6=70.291708382s
cputime-cpu7=70.591711177s
cputime-cpu8=70.442063069s
cputime-cpu9=70.406124863s
memory=64335872B
blkio-read=151552B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla argumentu 16000 jest to 71 s/1,13 s = 62,8 raza szybciej.

Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości n.




Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




### Podsumowanie

Widać, że problem mandelbrot jest przez Javę z wyłączonym JIT rozwiązywany niewiele wolniej niż przez Pythona. Java z włączonym JIT rozwiązuje zaś problem bardzo szybko, z ogromną przewagą w stosunku do Pythona. Stąd wniosek, że decydującą rolę w szybkości rozwiązywania tego problemu odgrywa JIT.

Mniejsze zużycie pamięci przez Pythona niz Javę w tym programie wynika z implementacji

algorytmu.

### 9.2.8. spectral-norm

Oblicza normę widmową nieskończonej macierzy A z wpisami $a_{11} = 1, a_{12} = 1/2, a_{21} = 1/3, a_{13} = 1/4, a_{22} = 1/5, a_{31} = 1/6$, itd.

Dla pomiarów za argument wejściowy przyjęto 5 500. Dokładniejszy opis: [15].

#### Java

W przypadku Javy walltime programu [Java #3] przy wykorzystaniu polecenia runexec wyniósł 0,618 s. Komplet wyników:

```
starttime=2024-12-25T20:19:10.698662+01:00
returnvalue=0
walltime=0.6107691810002507s
cputime=4.24365897s
cputime-cpu0=0.502503172s
cputime-cpu1=0.299692284s
cputime-cpu10=0.303250014s
cputime-cpu11=0.292081007s
cputime-cpu2=0.415043783s
cputime-cpu3=0.294426456s
cputime-cpu4=0.338604593s
cputime-cpu5=0.284607222s
cputime-cpu6=0.511081268s
cputime-cpu7=0.284779923s
cputime-cpu8=0.411520472s
cputime-cpu9=0.306068776s
memory=27381760B
blkio-read=0B
blkio-write=0B
```

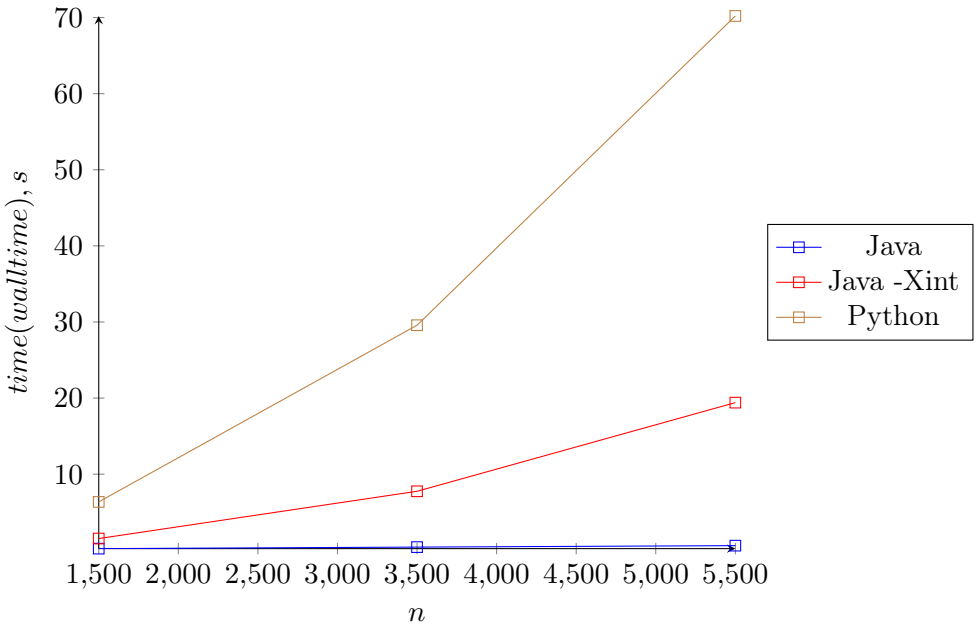
#### Python

W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wyniósł 70,2 s. Komplet wyników:

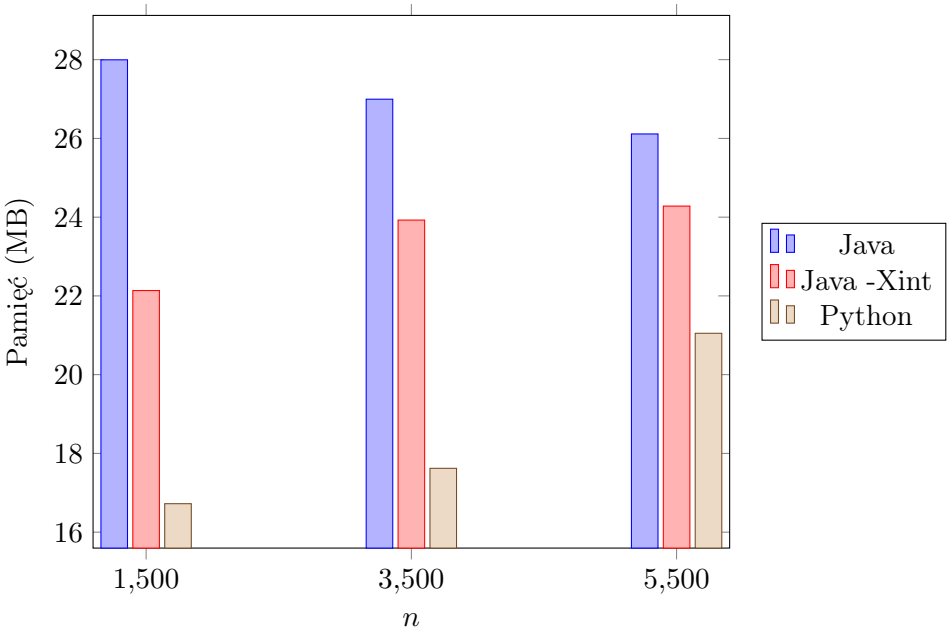
```
starttime=2024-12-25T20:24:30.374322+01:00
returnvalue=0
walltime=70.20847665300016s
cputime=277.682246331s
cputime-cpu0=23.436026862s
cputime-cpu1=33.131612600s
cputime-cpu10=14.584188744s
cputime-cpu11=12.294572871s
cputime-cpu2=16.686515146s
cputime-cpu3=39.313063969s
cputime-cpu4=46.249951552s
cputime-cpu5=8.991012702s
cputime-cpu6=8.721421147s
cputime-cpu7=31.267817521s
cputime-cpu8=27.581012635s
cputime-cpu9=15.425050582s
```

```
memory=22110208B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla argumentu 5 500 jest to 70.2 s/0.618 s = 114 raza szybciej. Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości N.




Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




### Podsumowanie

Ogromna różnica w szybkości wynika z tego, że test mocno obciąża procesor — zawiera dużo obliczeń numerycznych i pętli. Java (dzięki JVM, kompilacji JIT i zoptymalizowanym

typom prymitywnym) jest znacznie lepiej przystosowana do takiego zadania niż zwykły Python. Tymczasem Python ma duży narzut związany z modelem obiektowym, dynamicznym typowaniem i brakiem automatycznych optymalizacji w czystych pętlach.

Java ma większy niż Python początkowy narzut pamięci. Stąd dla małych wartości n potrzebuje znacznie więcej pamięci niż Python. Dla większych sytuacja poprawia się na korzyść Javy.

## 9.3. Przerobione programy ze strony geeksforgeeks.org

W tej sekcji porównano czas działania przerobionych programów testowych pochodzących ze strony geeksforgeeks.org.

### 9.3.1. Mergesort

Program Mergesort sortuje przy pomocy algorytmu sortowania mergesort listę/tablicę, w której znajdują się liczby uporządkowane malejąco: od n do 1.

W pomiarach za bazowe n przyjąłem 1 000 000.

#### Java

Kod źródłowy programu w Javie [Java]:

```
// Java program for Merge Sort
import java.io.*;

class GfG {

    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    static void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        // Create temp arrays
        int L[] = new int[n1];
        int R[] = new int[n2];

        // Copy data to temp arrays
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];

        // Merge the temp arrays

        // Initial indices of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = l;
```

```
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of L[] if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy remaining elements of R[] if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Main function that sorts arr[l..r] using
    // merge()
    static void sort(int arr[], int l, int r)
    {
        if (l < r) {

            // Find the middle point
            int m = l + (r - l) / 2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr, m + 1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    // Driver code
    public static void main(String args[])
    {
        int n = Integer.parseInt(args[0]);

        int arr[] = new int[n];
          for (int i = 0; i < n; i++)
              arr[i] = n - i;

        sort(arr, 0, n - 1);
```

```
    }
}
```

W przypadku Javy walltime programu dls n=1 000 000 przy wykorzystaniu polecenia runexec wyniósł 0,111 s. Komplet wyników:

```
starttime=2024-12-23T18:23:52.973959+01:00
returnvalue=0
walltime=0.11144095099962215s
cputime=0.11972095s
cputime-cpu0=0.020661110s
cputime-cpu1=0.003487471s
cputime-cpu10=0.005665164s
cputime-cpu11=0.001796297s
cputime-cpu2=0.000269566s
cputime-cpu3=0.001961156s
cputime-cpu4=0.004080414s
cputime-cpu5=0.001882721s
cputime-cpu6=0.000039865s
cputime-cpu8=0.078099802s
cputime-cpu9=0.001777384s
memory=90558464B
blkio-read=0B
blkio-write=0B
```

### Python

Kod źródłowy programu w Pythonie [Python]:

```
from sys import argv

def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 =e j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2

        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)

def main():
    n=int(argv[1])
    arr = [0] * n
    for i in range(0, n):
      arr[i] = n - i

    merge_sort(arr, 0, n - 1)

main() right - mid
```

```
27
28      # Create temp arrays
29      L = [0] * n1
30      R = [0] * n2
31
32      # Copy data to temp arrays L[] and R[]
33      for i in range(n1):
34          L[i] = arr[left + i]
35      for j in range(n2):
36          R[j] = arr[mid + 1 + j]
37
38      i = 0  # Initial index of first subarray
39      j = 0  # Initial index of second subarray
40      k = left  # Initial index of merged subarray
41
42      # Merge the temp arrays back
43      # into arr[left..right]
44      while i < n1 and j < n2:
45          if L[i] <= R[j]:
46              arr[k] = L[i]
47              i += 1
48          else:
49              arr[k] = R[j]
50              j += 1
51          k += 1
52
53      # Copy the remaining elements of L[],
54      # if there are any
55      while i < n1:
56          arr[k] = L[i]
57          i += 1
58          k += 1
59
60      # Copy the remaining elements of R[],
61      # if there are any
62      while j < n2:
63          arr[k] = R[j]
64          j += 1
65          k += 1
66
67  def merge_sort(arr, left, right):
68      if left < right:
69          mid = (left + right) // 2
70
71          merge_sort(arr, left, mid)
72          merge_sort(arr, mid + 1, right)
73          merge(arr, left, mid, right)
74
75  def main():
76      n=int(argv[1])
77      arr = [0] * n
78  z tym, że złożoność tej metody w Javie wynosi O(log N) a w Pythonie
79
80      for i in range(0, n):
81        arr[i] = n - i
```

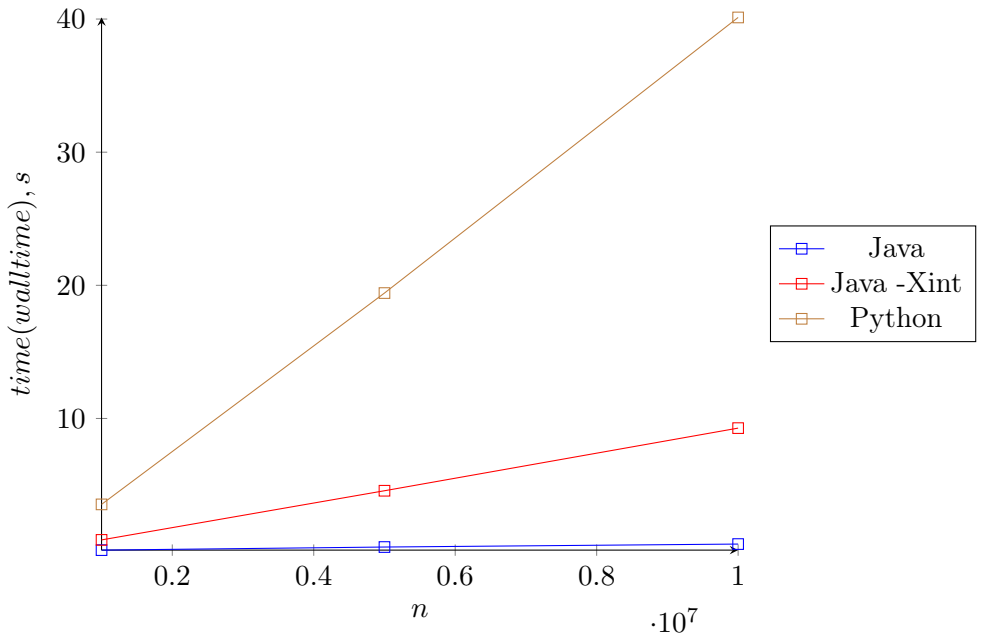
```
82
83      merge_sort(arr, 0, n - 1)
84
85  main()
```

W przypadku Pythona walltime programu dla n=1 000 000 przy wykorzystaniu polecenia runexec wyniósł 3,55 s. Komplet wyników:

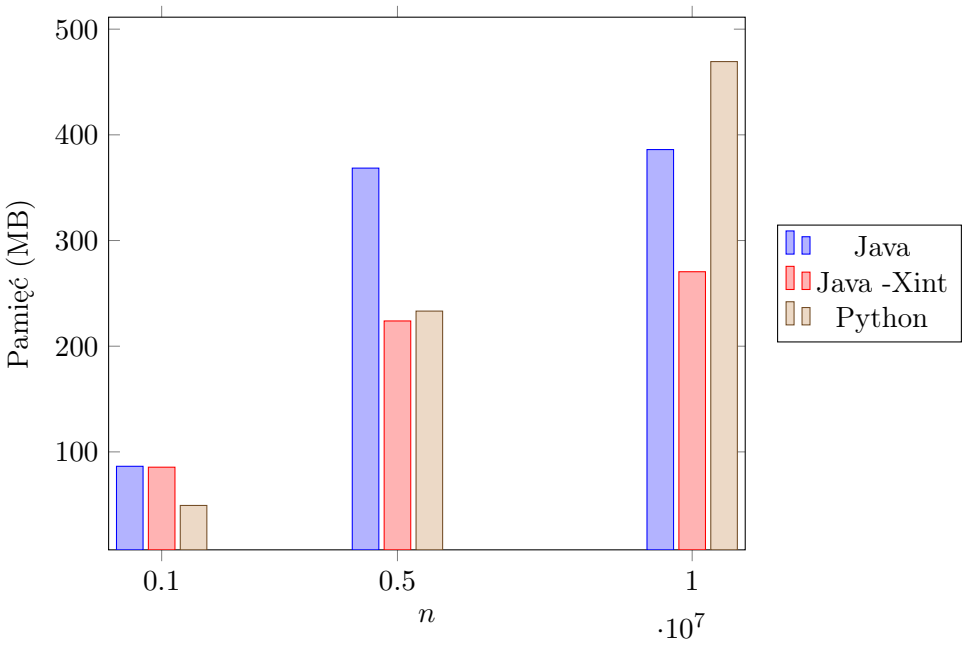
```
starttime=2024-12-23T18:26:32.246044+01:00
returnvalue=0
walltime=3.5459886850003386s
cputime=3.545602191s
cputime-cpu11=3.540892358s
cputime-cpu5=0.004709833s
memory=51707904B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n=1 000 000 jest to 3,55 s/0.111 s = 32 raza szybciej.

Poniżej znajduje się wykres czasu działania podstawowego algorytmu w Javie i Pythonie dla różnych wartości n.




Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




**Podsumowanie**

Python jest wolniejszy w tym teście głównie z powodu pracy na obiektach. Java może używać typów prymitywnych i wydajnych tablic co przekłada się na mniej kosztownych operacji.

Java ma większy niż Python początkowy narzut pamięci. Stąd dla mniejszych tablic zużywa więcej pamięci. Przy większych Mergesort w Pythonie zużywa więcej pamięci niż w Javie, ponieważ:

- Pythonowe listy przechowują obiekty, a nie surowe wartości.
- Interpreter Pythona ma większy narzut pamięci i stosu rekurencyjnego.
- Java używa tablic prymitywnych.

### 9.3.2. Problem n hetmanów

Mając daną liczbę całkowitą n, zadanie polega na znalezieniu wszystkich różnych rozwiązań problemu n-królowych, w którym n hetmanów jest umieszczonych na szachownicy n×n w taki sposób, że żadne dwie hetmany nie mogą się atakować.

Każde rozwiązanie jest unikalną konfiguracją n królowych, reprezentowaną jako permutacja [1,2,3,. . . ,n]. Liczba na i-tym miejscu oznacza rząd królowej w i-tej kolumnie. Na przykład [3,1,4,2] reprezentuje rozwiązanie dla n=4. W pomiarach za bazowe n przyjąłem 13.

**Java**

Kod źródłowy programu w Javie [Java]:

```
import java.util.ArrayList;
import java.util.List;

// Utility class for N-Queens
class GfG {

```

```
    // Utility function for solving the N-Queens
    // problem using backtracking.
    static void nQueenUtil(int j, int n,
                    List<Integer> board,
                    List<List<Integer> > result,
                    boolean[] rows, boolean[] diag1,
                    boolean[] diag2){
        if (j > n) {

            // A solution is found
            result.add(new ArrayList<>(board));
            return;
        }
        for (int i = 1; i <= n; i++) {
            if (!rows[i] && !diag1[i + j]
                && !diag2[i - j + n]) {

                // Place queen
                rows[i] = diag1[i + j] = diag2[i - j + n]
                    = true;
                board.add(i);

                // Recurse to next column
                nQueenUtil(j + 1, n, board, result, rows,
                           diag1, diag2);

                // Remove queen (backtrack)
                board.remove(board.size() - 1);
                rows[i] = diag1[i + j] = diag2[i - j + n]
                    = false;
            }
        }
    }

    // Solves the N-Queens problem and returns
    // all valid configurations.
    static List<List<Integer> > nQueen(int n){

        List<List<Integer> > result = new ArrayList<>();
        List<Integer> board = new ArrayList<>();
        boolean[] rows = new boolean[n + 1];
        boolean[] diag1 = new boolean[2 * n + 1];
        boolean[] diag2 = new boolean[2 * n + 1];

        // Start solving from first column
        nQueenUtil(1, n, board, result, rows, diag1, diag2);
        return result;
    }

    public static void main(String[] args){

        int n = Integer.parseInt(args[0]);
        List<List<Integer> > result = nQueen(n);
        //for (List<Integer> res : result) {
        //    System.out.print("[");
```

```
62          //     for (int i = 0; i < res.size(); i++) {
63          //         System.out.print(res.get(i));
64          //        if (i != res.size() - 1)
65          //             System.out.print(", ");
66          //     }
67          //     System.out.println("]");
68          //}
69      }
70  }
```

W przypadku Javy walltime programu dls n=13 przy wykorzystaniu polecenia runexec wyniósł 0,459 s. Komplet wyników:

```
starttime=2024-12-23T18:44:30.970948+01:00
returnvalue=0
walltime=0.4590756580000743s
cputime=0.506433078s
cputime-cpu0=0.005824122s
cputime-cpu1=0.059389618s
cputime-cpu10=0.005215852s
cputime-cpu11=0.002777278s
cputime-cpu2=0.003773366s
cputime-cpu6=0.000202028s
cputime-cpu7=0.000066765s
cputime-cpu8=0.429184049s
memory=33517568B
blkio-read=0B
blkio-write=0B
```

### Python

Kod źródłowy programu w Pythonie [Python]:

```
1  from sys import argv
2
3  # Utility function for solving the N-Queens
4  # problem using backtracking.
5  def nQueenUtil(j, n, board, result,
6                 rows, diag1, diag2):
7      if j > n:
8
9          # A solution is found
10         result.append(board.copy())
11         return
12     for i in range(1, n +1):
13         if not rows[i] and not diag1[i + j] and \
14            not diag2[i - j + n]:
15
16             # Place queen
17             rows[i] = diag1[i + j] = \
18                       diag2[i - j + n] = True
19             board.append(i)
20
21             # Recurse to next column
22             nQueenUtil(j +1, n, board,
```

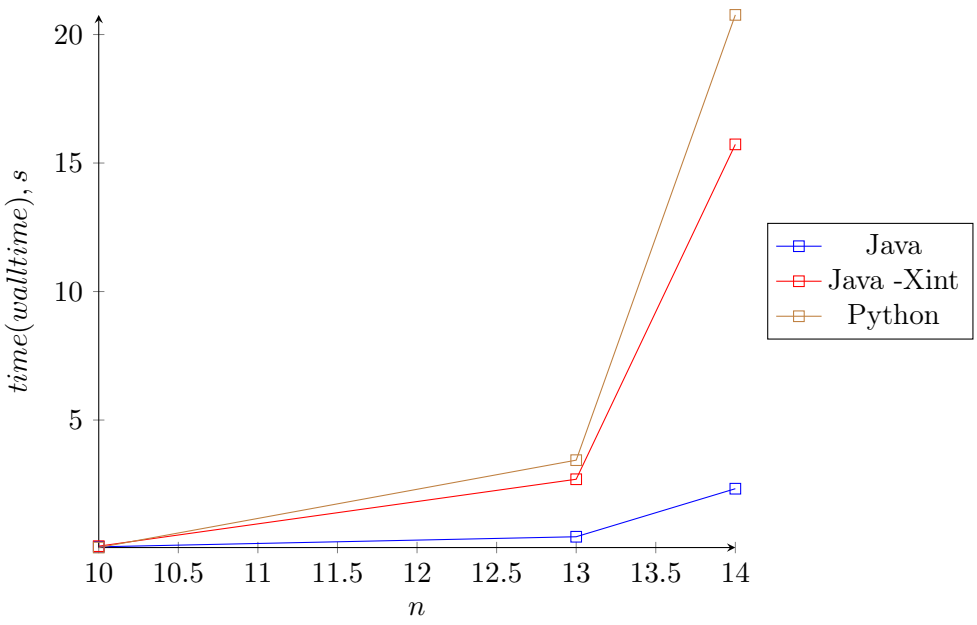
```
                        result, rows, diag1, diag2)

            # Remove queen (backtrack)
            board.pop()
            rows[i] = diag1[i + j] = \
                      diag2[i - j + n] = False

# Solves the N-Queens problem and returns
# all valid configurations.
def nQueen(n):
    result = []
    board = []
    rows = [False] * (n +1)
    diag1 = [False] * (2 * n +1)
    diag2 = [False] * (2 * n +1)

    # Start solving from first column
    nQueenUtil(1, n, board, result,
               rows, diag1, diag2)
    return result

def main():
    n=int(argv[1])
    result = nQueen(n)
#    for res in result:
#        print("[", end="")
#        for i in range(len(res)):
#            print(res[i], end="")
#            if i != len(res)-1:
#                print(", ", end="")
#        print("]")

main()
```

W przypadku Pythona walltime programu dla n=13 przy wykorzystaniu polecenia runexec wyniósł 3,44 s. Komplet wyników:

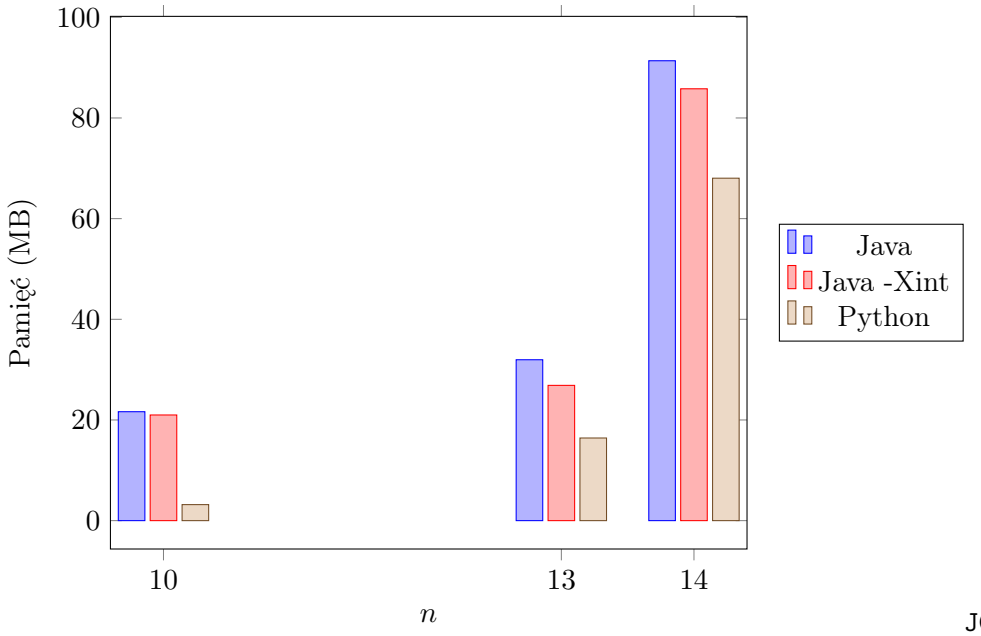
```
starttime=2024-12-23T18:45:24.910125+01:00
returnvalue=0
walltime=3.439188954999736s
cputime=3.401129201s
cputime-cpu0=1.877275741s
cputime-cpu6=1.523853460s
memory=17215488B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n=13 jest to 3,44 s/0,459 s = 7,49 raza szybciej.

Poniżej znajduje się wykres czasu działania programu w Javie i Pythonie dla różnych wartości n.


20
15
10
5
$time(walltime), s$
10
10.5
11
11.5
12
12.5
13
13.5
14
$n$
Java
Java -Xint
Python

Poniżej znajduje się wykres zapotrzebowania na pamięć w przypadku Javy i Pythona dla różnych wartości n.




JC 11-02: Brakuje podsumowań po każdym teście z Pana interpretacją wyników (dlaczego wg Pana są jakie są). No i legendę może dałoby się przenieść na prawo? Wtedy całość byłaby zgrabniejsza (chociaż może wcale nie, bo nic już by nie tłumaczyło tylko dwóch obrazków na stronie. A trzy by pewnie i tak nie weszły...

### Podsumowanie

Można zaobserwować, że problem n-hetmanów jest przez Javę z wyłączonym JIT rozwiązywany niewiele szybciej niż przez Pythona. Stąd wniosek, że dużą rolę w szybkości rozwiązywania tego problemu odgrywa JIT.

Java ma większy niż Python początkowy narzut pamięci. Stąd dla mniejszej liczby hetmanów zużywa znacznie więcej pamięci. Dla większej liczby sytuacja się wyrównuje. Java ma lepsze zarządzanie pamięcią, bo tablice prymitywów są zwarte, a odśmiecacz pamięci jest bardziej przewidywalny.


Java
10
python™

# Rozdział 10

# Podsumowanie

W pracy porównano bajtkody i maszyny wirtualne Javy i Pythona. Omówiono architekturę JVM i PVM, JIT Javy i stosowane przez niego optymalizacje oraz dopiero co dodany JIT do Pythona 3.13. Dokonano analizy porównawczej kolekcji Javy i Pythona. Porównano czas działania i zużycie pamięci dwóch programów (Mergesort, problem n hetmanów) ze strony geeksforgeeks [6] w Javie i Pythonie. Zbadano też czas działania oraz zużycie pamięci programów rozwiązujących te same problemy w Javie i Pythonie ze strony CLBG [7]: binary-trees, fannkuch-redux, n-body, fasta, reverse-complment, mandelbrot, spectral-norm.

**Uzyskano następujące wnioski:**

- Piętą achillesową Javy jest obsługa dużych liczb całkowitych. Klasa BigInteger Javy działa wolniej niż duże liczby w Pythonie co sprawia, że programy na niej oparte są również wolniejsze od analogicznych programów w Pythonie. Z pomocą przychodzą tu zewnętrzne biblioteki takie jak GMP, ale komplikują one z kolei kod programu.
- Z pomocą Pythonowi przychodzą zewnętrzne biblioteki takie jak np. NumPy do obliczeń numerycznych, ale nie zawsze ich zastosowanie przyspiesza działanie programów.
- Okazuje się, że w niektórych przypadkach kolekcje Pythona są szybsze od ich odpowiedników w Javie. Ma to miejsce np. w przypadku metody containsKey klasy TreeSet (Java) i in klasy SortedSet (Python). Jest to związane z tym, że złożoność tej metody w Javie wynosi O(log N) a w Pythonie O(1). W przypadku, gdy złożoność metody jest taka sama w obu językach, Java okazuje się szybsza.
- Kod Pythonowy działa szybciej w funkcji niż poza nią. Jest to odczuwalne, gdy elementów, po których iterujemy jest dużo.
- Java — silnie typowany bajtkod i zaawansowana JVM zapewniają wysoką wydajność, optymalizacje i skalowalność.
- Python — dynamiczny, lekki bajtkod wykonywany przez PVM jest elastyczny i prosty, ale trudniejszy do optymalizacji.

# Bibliografia


[1] *The Java® Virtual Machine Specification*, https://docs.oracle.com/javase/specs/jvms/se13/html/index.html, 2019-08-21

[2] Sanket Bhosale, *Memory Management in JVM and PVM: A Comparative Overview*, https://www.linkedin.com/posts/sanket-bhosale-3108751a7_java-python-core2web-activity-7134199599744843776-rjUK/, 2024

[3] Sakshee Agrawal, *Understanding Just-In-Time (JIT) Compilation in Java*, https://medium.com/@sakshee_agrawal/understanding-just-in-time-jit-compilation-in-java-ae2a6b9fa931, 2023-10-17

[4] *JVM vs PVM*, http://simplealgo.com/jvm-vs.pvm/, 2024

[5] Obi Ike-Nwosu, *Inside The Python Virtual Machine*, https://leanpub.com/insidethepythonvirtualmachine/read, 2020-08-07

[6] Gouy, Isaac. *geeksforgeeks*, https://www.geeksforgeeks.org/ (odwiedzone 31/12/2024).

[7] Gouy, Isaac. *The Computer Language Benchmarks Game*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html (odwiedzone 31/12/2024).

[8] Gouy, Isaac. *The Computer Language Benchmarks Game: pidigits description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/pidigits.html#pidigits (odwiedzone 31/12/2024).

[9] Gouy, Isaac. *The Computer Language Benchmarks Game: binarytrees description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/binarytrees.html#binarytrees (odwiedzone 31/12/2024).

[10] Gouy, Isaac. *The Computer Language Benchmarks Game: fannkuchredux description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html#fannkuchredux (odwiedzone 31/12/2024).

[11] Gouy, Isaac. *The Computer Language Benchmarks Game: nbody description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/nbody.html#nbody (odwiedzone 31/12/2024).

[12] Gouy, Isaac. *The Computer Language Benchmarks Game: fasta description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fasta.html#fasta (odwiedzone 31/12/2024).

[13] Gouy, Isaac. *The Computer Language Benchmarks Game: revcomp description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/revcomp.html#revcomp (odwiedzone 31/12/2024).

[14] Gouy, Isaac. *The Computer Language Benchmarks Game: mandelbrot description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/mandelbrot.html#mandelbrot (odwiedzone 31/12/2024).

[15] Gouy, Isaac. *The Computer Language Benchmarks Game: spectralnorm description.*, https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/spectralnorm.html#spectralnorm (odwiedzone 31/12/2024).

[16] Liquid Web *Python version history*, https://www.liquidweb.com/blog/latest-python-version/.

[17] Hitesh Umaletiya *History of various Java versions*, https://www.brilworks.com/blog/java-versions-new-features-and-deprecation/.

[18] Admin 3 *Java Version History*, https://manaschool.in/history-of-java/.

[19] *Maszyna wirtualna*, https://pl.wikipedia.org/wiki/Maszyna_wirtualna.

[20] *Bajtkod*, https://pl.wikipedia.org/wiki/Kod_bajtowy.

[21] Anthony Shaw, "Python 3.13 gets a JIT", Blog, 9 Jan 2024, https://tonybaloney.github.io/posts/python-gets-a-jit.html.

[22] Sanket Bhosale, "Java / Python – core2web activity ...", LinkedIn post, https://www.linkedin.com/posts/sanket-bhosale-3108751a7_java-python-core2web-activity-7134199599744843776-rjUK?trk=public_profile.

[23] Srikanth Dannarapu, "JVM Architecture", Medium, 7 Mar 2023, https://medium.com/javarevisited/jvm-architecture-32def70b6de.

[24] Victor (author), "Python behind the scenes #1: how the CPython VM works", TenThousandMeters.com blog, https://tenthousandmeters.com/blog/python-behind-the-scenes-1-how-the-cpython-vm-works/.

[25] Python Dev Guide, https://devguide-pt-br.readthedocs.io/pt-br/latest/internals/interpreter/.

[26] Społeczność deweloperów LLVM, „LLVM — Infrastruktura kompilatora LLVM", https://llvm.org/, dostęp: 3 listopada 2025.

[27] P. Kumar, M. Kurup, Zarządzanie pamięcią w Javie, DigitalOcean, 2025, https://www.digitalocean.com/community/tutorials/java-jvm-memory-model-memory-management-in-java.