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, Kwiecień 2025

Streszczenie

W pracy porównano batjkody Javy i Pythona. Omówiomo 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.

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

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 ninieszej pracy postanowiłem porównać bajtkody i maszyny wirtualne owych języków w celu wskazania szczegółow, którymi się różnią. Porównałem je też pod względem zużycia pamięci.

Praca składa się z dziesięciu rozdziałów. W rozdziale 1 przypomniano podstawowe pojęcia związane z językami Java i Python takie jak bajtkod czy maszyny wirtualne. Omówienie instrukcji bajtkodów zawarto w rozdziale 2. W rozdziale 3 dokonano wstępnego porównania JVM i PVM. W rozdziale 4 omówiono architekturę maszyn wirtualnych Javy i Pythona. W rozdziale 5 porównano JIT Javy i Pythona. W rozdziale 6 przedstawiono zarządzanie pamięcią. W rozdziale 7 wspomniano o środowisku uruchomieniowym. Rozdział 8 zawiera porównanie czasu działania prostych fragmentów kodu. W rozdziale 9 porównano czas działania większych programów. Rozdział 10 zawiera podsumowanie.

0.1. 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.

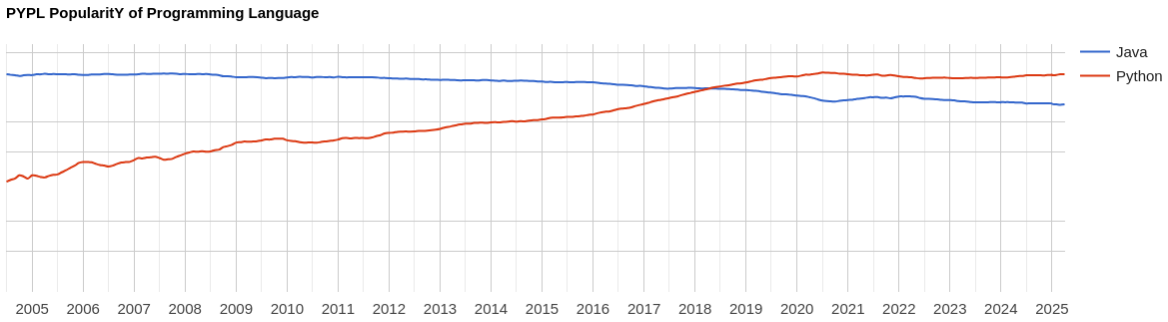
0.2. Historia Pythona

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 3.13.2	Free-threaded mode (experimental), Just-In-Time (JIT) compiler, usprawniony, interaktywny interpreter, colorized tracebacks
2025	Python 3.13.3	Nowy interaktywny shell, free-threading, JIT preview, mimalloc, aktualizacja platformy

0.3. Popularność i znaczenie



Rysunek 1: źródło: https://pypl.github.io/PYPL.html

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, Apr 2025 :

Rank	Change	Language	Share	-year trend
1		Python	30.27 %	+1.4 %
2		Java	15.04 %	-0.6 %
3		JavaScript	7.93 %	-0.7 %
4	↑	C/C++	6.99 %	+0.6 %
5	↓	C#	6.2 %	-0.6 %

Rysunek 2: źródło: https://pypl.github.io/PYPL.html

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 50% przy czym Python jest 2 razy popularniejszy od Javy (stan na kwiecień 2025);


Java
1
python™

Rozdział 1

Podstawowe pojęcia

1.1. 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.

1.1.1. 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.

1.2. 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
2
python™

Rozdział 2

Omówienie instrukcji bajtkodów

2.1. 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.

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

2.2. Struktura bajtkodu Javy

Każda instrukcja ma następującą postać:

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

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 jest 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.

Poprzedza się operand reprezentujący indeks w run-time constant pool znakiem hash i postępuje zgodnie z komentarzem identyfikującym element run-time constant pool, do którego się odwołuje, na przykład:

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

albo:

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

2.3. Instrukcje bajtkodu Pythona

Dokładny zestaw instrukcji zależy od wesji CPythona. W CPython'ie 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. **BINARY_POWER** nie ma argumentu, jako że jego bajtowa reprezentacja wynosi 19.

Instrukcje 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**, **UNARY_NOT**).

- Operacje binarne, które pobierają szczyt stosu (TOS) oraz drugi element ze szczytu stosu (TOS1), dokonują operację a następnie umieszczają wynik na szyczcie stosu (np. **BINARY_POWER**, **BINARY_ADD**).
- Operacje w miejscu, które dają taki sam efekt jak operacje binarne tzn. pobierają element ze szczytu stosu (TOS) oraz drugi element ze szczytu stosu (TOS1), dokonują operację a następnie umieszczają wynik na szyczcie stosu, ale są wykonywane w miejscu o ile TOS1 to obsługuje a wynikowy TOS może być oryginalnym TOS1. Przykłady takich operacji to (np. **INPLACE_POWER**, **INPLACE_ADD**)
- Operacje na listach wymagające do 3 argumentów (np. **SLICE+0**).
- Operacje na listach wymagające nawet dodatkowego argumentu. Nie umieszczają one niczego na stosie (np. **STORE_SLICE)**.

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
values = STACK.pop()
container[start:end] = value
```

- Rozmaite inne operacje, niekiedy wymagające argumentów (np. **PRINT_NEWLINE**, **LOAD_CONST**).

2.4. Struktura bajtkodu Pythona

Analizuję strukturę bajtkodu Pythona na przykładzie funkcji myfunc:

```
def myfunc(alist):
    return len(alist)
```

i jej kod bajtowy:

```
>>> dis.dis(myfunc)
  2           0 RESUME                   0

  3           2 LOAD_GLOBAL              1 (NULL + len)
             12 LOAD_FAST                0 (alist)
             14 CALL                     1
             22 RETURN_VALUE
```

Numer w pierwszej kolumnie oznacza numer linii w kodzie źródowym Pythona. Wiele instrukcji może być przypisanych do tego samego numeru linii. Wartość ta jest obliczana na podstawie informacji z pola co_lnotab obiektu **code object**. Druga kolumna to offset danej instrukcji od początku bajtkodu. Zakładam, że ciąg kodu bajtowego jest zawarty w tablicy, więc wartość ta jest indeksem instrukcji w tablicy. Trzecia kolumna to rzeczywisty, czytelny kod instrukcji. Czwarta kolumna to argument instrukcji.

Instrukcja LOAD_FAST przyjmuje argument 0. Ta wartość jest indeksem tablicy co_varnames. Ostatnia kolumna to wartość argumentu - podana przez funkcję dis. Niektóre opcode'y nie wymagają jawnych argumentów. Zwraca się uwagę, że instrukcja RETURN_VALUE nie przyjmuje jawnego argumentu. Jak również, że maszyna wirtualna Pythona jest oparta na stosie, więc ta instrukcja odczytuje wartość z góry stosu.

Instrukcje kodu bajtowego mają rozmiar dwóch bajtów – jeden bajt dla kodu operacji i drugi

bajt dla argumentu operacji. W przypadku, gdy kod operacji nie przyjmuje argumentu, wówczas drugi bajt argumentu jest wyzerowany. Maszyna wirtualna Pythona używa kodowania bajtów Little-Endian. Kod operacji zajmuje 8 wyższych bitów i argument kodu operacji zajmuje 8 niższych bitów.

Czasami argument kodu operacji może nie zmieścić się w domyślnym pojedynczym bajcie, dlatego maszyna wirtualna Pythona używa kodu operacji EXTENDED_ARG dla tego rodzaju argumentów. Pobiera argument, który jest zbyt duży, aby zmieścił się w pojedynczym bajcie, i dzieli go na dwa (zakłada się, że zmieści się na dwóch bajtach, ale tę logikę można łatwo rozszerzyć powyżej dwóch bajtów) - najbardziej znaczący bajt jest argumentem kodu operacji EXTENDED_ARG, podczas gdy najmniej znaczący bajt jest argumentem jego rzeczywistego kodu operacji. Kody operacji EXTENDED_ARG pojawią się przed rzeczywistym kodem operacji w sekwencji rozkazów, a argument można następnie odbudować poprzez przesunięcie w prawo i połączenie z innymi sekcjami argumentu. Na przykład, jeśli ktoś chce przekazać wartość 321 jako argument do kodu operacji LOAD_CONST, wartość ta nie może zmieścić się w pojedynczym bajcie, dlatego używany jest kod operacji EXTENDED_ARG. Binarna reprezentacja tej wartości to 0b101000001, więc rzeczywisty kod operacji (LOAD_CONST) przyjmuje pierwszy bajt (1000001) jako argument (65 w postaci dziesiętnej), podczas gdy kod operacji EXTENDED_ARG przyjmuje następny bajt (1) jako argument ; zatem mamy (144, 1), (100, 65) jako wyprowadzaną sekwencję instrukcji.

2.5. Porównanie bajtkodów

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 przyapdku Javy jest to **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 pokazuje 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);
```

```
      OPC_INT_BINARY(add, Add, 0);
```

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

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

Poniżej pokazuje 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 zmiennymi a maszyna wirtualna Pythona wywyołuje 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 */
            slotw = NULL;
        }
        x = slotv(v, w);
        if (x != Py_NotImplemented)
            return x;
        Py_DECREF(x); /* can't do it */
    }
    if (slotw) {
        x = slotw(v, w);
        if (x != Py_NotImplemented)
            return x;
        Py_DECREF(x); /* can't do it */
    }
    Py_RETURN_NOTIMPLEMENTED;
}
```

Funkcja binary_op1() pobiera trzy parametry: lewy operand, prawy operand i offset, który identyfikuje slot. Typy obydwu operandów mogą implementować slot. Z tego powodu, binary_op1() sprawdza obydwie implementacje. Aby uzyskąć wynik, woła jedną implementację lub drugą opierając się na następującej logice:

1. Jeśli typ jednego operandu jest podtypem drugiego, wywołaj slot podtypu.

2. Jeśli lewy operand nie ma slotu, wywołaj slot prawego operandu.

3. W przeciwnym przypadku, wywołaj slot lewego operandu.


Java
3
python™

Rozdział 3

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 częścią Java Virtual Machine specification. PVM jest z kolei używana przez CPythona, standardową implementację Pythona. Maszyny te służą podobnym celom, ale jednocześnie zasadniczo się różnią.

3.1. Języki programowania:

- **JVM:** JVM został zaprojektowany 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.

3.2. 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.

3.3. Zarządzanie pamięcią:

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

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

3.4. 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 platformy, ale dostępność PVM dla konkretnej platformy może nie być tak powszechna, jak implementacje JVM. Ponadto PVM zmienia się co wersję Pythona. Uważa się, że przenośność Pythona jest wystarczająco dobra.

3.5. 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
4
python

Rozdział 4

Architektura

4.1. Architektura maszyny wirtualnej Javy

Programiści Javy wiedzą, że kod bajtowy będzie wykonywany przez JRE (Java Runtime Environment). Ale niewielu wie, że JRE to implementacja wirtualnej maszyny Java (JVM), która analizuje kod bajtowy, interpretuje kod 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:

- Class Loader Subsystem
- Runtime Data Area
- Execution Engine

4.2. Class Loader Subsystem

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

4.2.1. Loading

Klasy będą załadowane przez ten komponent. Boot Strap ClassLoader, Extension ClassLoader i Application ClassLoader to trzy klasy modułu ładującego, które pomogą w osiągnięciu tego celu..

- **Boot Strap ClassLoader** – Odpowiedzialny za ładowanie klas ze ścieżki klas bootstrap, nic poza rt.jar. Ten moduł ładujący będzie miał najwyższy priorytet.

- **Extension ClassLoader** – Odpowiedzialny za ładowanie klas znajdujących się w folderze ext.
- **Application ClassLoader** – Odpowiedzialny za ładowanie Application Level Classpath, Environment Variable

4.2.2. Linking

- **Verify** – Weryfikator kodu bajtowego zweryfikuje poprawność kodu. W przypadku niepowodzenia weryfikacji otrzymany zostanie błąd.
- **Prepare** – Dla wszystkich zmiennych statycznych zostanie przydzielona pamięć i wartości domyślne.
- **Resolve** – Wszystkie symboliczne odniesienia do pamięci są zastąpione początkoqymi odniesieniami z Method Area.

4.2.3. Initialization

Jest to ostatnia faza ładowania klas, w konsekwencji wszystkim zmiennym statycznym zostają przypisane początkowe wartości i zostaje wykonany blok statyczny.

4.3. Runtime Data Area

The Runtime Data Area jest podzielona na 5 głównych komponentów:

4.3.1. Method Area

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

4.3.2. Heap Area

Wszystkie obiekty i odpowiadające im zmienne instancji oraz tablice są tutaj przechowywane. Istnieje również jeden obszar sterty na każdą maszynę JVM. Ponieważ method area i heap area współdzielą pamięć dla wielu wątków, przechowywane dane nie są bezpieczne dla wątków.

4.3.3. Stack Area

Dla każdego wątku zostaje utworzony oddzielny runtime stack. Dla każdego wywołania metody zostaje utworzony jeden wpis w pamięci stosu, nazywany stack frame. Wszystkie zmienne lokalne zostają utworzone w pamięci stosu. Obszar stosu jest bezpieczny dla wątków, ponieważ nie jest zasobem współdzielonym. Stack frame jest podzielony na trzy podjednostki:

- **Local Variable Array** – W zależności od metody, ile zmiennych lokalnych jest zaangażowanych i odpowiadające im wartości będą tutaj przechowywane.
- **Operand stack** – Jeśli do wykonania wymagana jest jakakolwiek operacja pośrednia, operand stack pełni rolę obszaru roboczego środowiska wykonawczego dla wykonania operacji.

- **Frame data** – Tutaj przechowywane są wszystkie symbole odpowiadające tej metodzie. W przypadku jakiegokolwiek wyjątku, informacja o bloku catch zachowana jest we frame data.

4.3.4. PC Registers

Każdy wątek ma osobne rejestry PC, aby przechowywać adres aktualnie wykonywanej instrukcji, po jej wykonaniu rejestr PC zostaje zaktualizowany o następną instrukcję.

4.3.5. Native Method stacks

– Native Method Stack przechowuje informacje o metodach natywnych. Dla każdego wątku tworzony jest oddzielny stos metod natywnych.

4.4. Execution Engine

Kod bajtowy przypisany do Runtime Data Area jest wykonany przez Execution Engine, który odczytuje kod bajtowy i wykonuje go fragmentami.

4.4.1. Interpreter

Gdy jedna metoda jest wywoływana wielokrotnie, konieczna jest za każdym razem nowa interpretacja. To jest wada interpretatora.

4.4.2. JIT Compiler

Kompilator JIT neutralizuje wady interpretera. Execution Engine korzysta z interpretera przy konwersji kodu bajtowego, ale kiedy natrafi na powtarzający się kod, używa kompilatora JIT, który kompiluje cały kod bajtowy i zamienia go na kod natywny. Ten natywny kod używany jest bezpośrednio do powtarzających się wywołań metod, co poprawia wydajność systemu.

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

4.4.3. Garbage Collector:

Zbiera i usuwa obiekty, do których nie ma odniesień. Można go uruchomić, wywołując „System.gc()”, ale wykonanie jest niepewne. Garbage collection maszyny wirtualnej Javy gromadzi utworzone obiekty.

Java Native Interface (JNI): JNI współdziała z Native Method Libraries i udostępnia Native Libraries wymagane przez Execution Engine.

Native Method Libraries: Jest to kolekcja Native Libraries wymaganych przez Execution Engine.

4.5. Architektura maszyny wirtualnej Pythona

Wywołanie programu Pythona składa się z trzech etapów:

- **Inicjalizacja**
- **Kompilacja**
- **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 kompilacji. CPython jest interpreterem, a nie kompilatorem w tym sensie, że nie tworzy kodu maszynowego. Interpretery tłumaczą jednak zwykle kod źródłowy do jakiejś postaci pośredniej przed jego wykonaniem. Tak robi również CPython. Etap translacji wykonuje te same czynności 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.

4.5.1. Code object

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 przedstawiam definicję 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
```

```
    Py_ssize_t *co_cell2arg;    /* Maps cell vars which are arguments. */
    PyObject *co_filename;      /* unicode (where it was loaded from) */
    PyObject *co_name;          /* unicode (name, for reference) */
        /* ... more members ... */
};
```

4.5.2. Function object

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 są przechowywane w obiekcie funkcji.

Poniżej przedstawiam definicję struktury obiektu funkcji.

```
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;
```

4.5.3. Frame object

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 przedstawiam definicję obiektu ramki.

```
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.

4.5.4. Thread state

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;
```

4.5.5. Interpreter state

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*).

Przedstawiam poniżej znajduje definicję 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
```

4.5.6. Runtime state

Stan środowiska wykonawczego jest zmienną globalną. Przechowuje dane specyficzne dla procesu. Obejmuje to stan CPythona (np. czy został zainicjowany czy nie) i mechanizm GIL.

4.5.7. Evaluation loop

Wykonanie kodu bajtowego Pythona przez maszynę wirtualną wydaje się skomplikowane. W rzeczywistości jedyne co musi reobić to iterować po instrukcjach i działać zgodnie z nimi. I to robi *_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. Kod bajtowy jest reprezentowany przez tablicę 16-bitowych liczb całkowitych bez znaku, po jednej liczbie całkowitej na instrukcję. 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 można przedstawić ją 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);
        next_instr++;

        switch (opcode) {
            case TARGET(NOP) {
                FAST_DISPATCH(); // more on this later
            }

            case TARGET(LOAD_FAST) {
                // ... code for loading local variable
            }

            // ... 117 more cases for every possible opcode
        }

        // ... error handling
    }

    // ... termination
}
```


Java
5
python™

Rozdział 5

Porównanie JIT Python'a i Javy.

5.1. JIT Javy

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

Kompilacja JIT polega na tłumaczeniu kody bajtowego Java na natywny kod maszynowy w locie tuż przed wykonaniem. Jest zasadniczym elementem modelu wyknawczego Java. W przeciwieństwie do AOT (Ahead Of Time), który kompiluje kod przed uruchomieniem programu, oferując szybkie wykonanie pętli, ale wolniejsze uruchamianie, JIT tłumaczy kod na bieżąco podczas działania programu, zapewnia szybsze uruchamianie i dostosowuje się do aktywnych ścieżek kodu.

JVM obsługuje aplikacje Java i współpracuje z JIT. Podczas wykonywania JVM komunikuje się z kompilatorem JIT. JIT identyfikuje często używane ścieżki kodu i tłumaczy kod bajtowy na wydajny kod natywny, przyspieszając wykonanie. JVM i JIT razem zapewniają, że aplikacje Java działają płynnie i dobrze.

5.1.2. Jak działa JIT:

Kompilator Just-In-Time (JIT) jest ważnym składnikiem środowiska Java Runtime Environment (JRE). Znacznie zwiększa wydajność i szybkość aplikacji Java w czasie wykonywania. Zrozumienie działania JIT zapewnia wgląd w szybkość i wydajność Java. Przedstawię w punktach jak dochodzi do jego zastosowania:

- Java Source to Bytecode:

  Wszystko zaczyna się od kodu źródłowego Java. Tworzymy ten kod skrupulatnie. Ale zanim zacznie działać, musi zostać przekształcony. Kompilator Java, często nazywany „javac", kompiluje kod do kodu bajtowego. Ten kod bajtowy jest neutralną dla platformy wersją naszego kodu źródłowego. Można nazwać go uniwersalnym przepisem naszego programu.

- Loading and Interpretation:

  Po naciśnięciu przycisku „Uruchom" w aplikacji Java zaczyna pracę wirtualna maszyna Java (JVM). Ładuje skompilowane klasy i metody, gotowe do wykonania. Początkowo JVM interpretuje kod bajtowy. To traktujemy jak czytanie tego uniwersalnego przepisu, punkt po punkcie. Działa, ale czasami może działać nieco wolno, szczególnie w przypadku często używanych części kodu.

- JIT Compilation Activation

  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.

- JIT Compilation:

  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.

- Optimized Execution:

  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.

- Performance Enhancement: Chociaż kompilacja JIT potrzebuje trochę czasu i pamięci procesora przy uruchamianiu maszyny i wywoływaniu wielu metod, opłaca się, ponieważ aplikacja działa płynniej i wydajniej. Podczas uruchamiania aplikacji Java kompilator JIT dynamicznie konwertuje „gorące" ścieżki kodu z kodu bajtowego na bardzo wydajny natywny kod maszynowy. Te wspaniałe cechy JIT zapewniają wysoką jakość i niezależność od platformy. Jeszcze raz podkreślę, że dzięki kompilatorze JIT środowisko Java jest tak wyjątkowe, a aplikacje Java zachwycają szybkością i responsywnością.

5.1.3. Optymalizacje stosowane przez komilatory JIT:

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

- Inlining:

  Inlining to proces łączenia mniejszych metod z ich obiektami wywołującymi, zmniejszający obciążenie wywołania metod. Technika ta przyspiesza często wykonywane wywołania metod. Obejmuje optymalizacje, takie jak trivial inlining, call graph inlining, tail recursion elimination i virtual call guard optimizations.

- Local Optimizations:

  Lokalne optymalizacje ulepszają małe fragmenty kodu na raz, wykorzystując techniki takie jak local data flow analyses, register usage optimization i simplifications of Java idioms.

- Control Flow Optimizations:

  Control flow optimizations analizują i reorgaznizują ścieżki kodu poprawiając wydajność. Obejmują one code reordering, loop-related enhancements (reduction, inversion, unrolling), i doskonalsze exception handling.

- Global Optimizations:

  Globalne optymalizacje działają na całą metodę, wymagają więcej czasu kompilacji, ale oferują znaczny wzrost wydajności. Obejmują one global data flow analyses, partial redundancy elimination, escape analysis i optiymalizacje związane z garbage collection i memory allocation.

- Native Code Generation:

  W końcowej fazie drzewa są tłumaczone na instrukcje kodu maszynowego i stosowane są pewne optymalizacje specyficzne dla platformy. Skompilowany kod jest przechowywany w pamięci podręcznej kodu JVM, gotowy do użycia w przyszłości, zapewniając szybkość wykonania.

  Można powiedzieć, że kompilacja JIT jest jak magiczna transformacja, która zamienia zwykły kod Java w superwydajny, błyskawiczny kod maszynowy.

5.1.4. Zalety kompilatora JIT:

- Efficient Memory Usage:

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

- Dynamic Code Optimization:

  Kompilatory JIT optymalizują kod w czasie wykonywania, umożliwiając im dostosowanie się do kontekstu wykonania, co prowadzi do poprawy wydajności.

- Multiple Optimization Levels:

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

- Reduced Page Faults:

  Generując w razie potrzeby natywny kod maszynowy, kompilatory JIT zmniejszają prawdopodobieństwo błędów stron i operacji we/wy dysku, zwiększając czas reakcji aplikacji.

5.1.5. Wady kompilatora JIT:

- Increased Program Complexity:

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

- Limited Benefit for Short Code:

  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.

- Cache Memory Usage:

  Kompilatory JIT zużywają znaczną ilość pamięci podręcznej, potencjalnie wpływając na ogólną wydajność systemu, szczególnie w środowiskach o ograniczonych zasobach..

Wnioski:

Zalety JIT obejmują efektywne wykorzystanie pamięci, dynamiczną optymalizację kodu, wiele poziomów optymalizacji i zmniejszoną liczbę błędów stron, a wszystko to przyczynia się do zwiększonej wydajności. Jednak JIT wprowadza pewną złożoność i może nie zapewniać znacznych korzyści w przypadku bardzo krótkich fragmentów kodu. Warto dokonać takiego kompromisu ze względu na szybkość i możliwości adaptacji, jakie wnosi do ekosystemu Java.

5.2. JIT Pythona

CPython od wersji 3.13 obsługuje JIT. Wykorzystuje on technikę copy-and-patch. Copy-and-patch to prosta technika kompilatora przeznaczona do kompilacji just-in-time (kompilacja JIT), która wykorzystuje pattern matching w celu dopasowania wstępnie wygenerowanych szablonów do części abstrakcyjnego drzewa składni (AST) lub strumienia kodu bajtowego. Emituje odpowiednio wcześniej napisane fragmenty kodu maszynowego, poprawia je i wstawia adresy pamięci, adresy rejestrów, stałych i innych parametrów potrzebnych do wytworzenia kodu wykonywalnego. Kod niepasujący do szablonów można albo zinterpretować w normalny sposób, albo utworzyć kod w celu bezpośredniego wywołania kodu interpretera.

Benchmarki wskazują na 2-9 procentowy przyrost szybkości w stosunku do Pythona bez JIT.


Java
6
python™

Rozdział 6

Zarządzanie pamięcią

Tylko odpowiednie zarządzanie pamięcią maszyn wirtualnych doprowadzi do efektywnego wykorzystania zasobów. Postaram się przedstawić mechanizmy alokacji pamięci, tworzenia obiektów oraz mechanizmy usuwania śmieci stosowane przez wirtualną maszynę Java i wirtualną maszynę Python.

6.1. Memory Allocation and Object Creation:

6.1.1. JVM

JVM wykorzystuje trzy główne obszary do zarządzania pamięcią: obszar metod, stos i stertę. Kiedy obiekty są tworzone przy użyciu słowa kluczowego new, pamięć jest alokowana na stercie, a konstruktor służy do inicjowania obiektu. Sterta jest podzielona na trzy pokolenia:

- **Young Generation:** To tu powstają nowe obiekty
- **Old Generation:** Obiekty, które przetrwały kilka rund zbiórki śmieci w ramach Yotung Generation, ostatecznie awansują do Old Generation, wykorzystywanego do obiektów długowiecznych
- **Metaspace:** Jest to natywny obszar pamięci, który dynamicznie dostosowuje swój rozmiar

6.1.2. PVM

Python wykorzystuje prywatną stertę do zarządzania pamięcią. Globalna blokada interpretera (GIL) zapewnia, że tylko jeden wątek może jednocześnie wykonywać kod bajtowy w czasie zarządzania pamięcią. Python jest typowany dynamicznie, a obiekty tworzone są przy użyciu prostej składni z automatycznie zarządzaną pamięcią.

Python obsługuje różne typy obiektów, takie jak int, dict sklasyfikowane jako obiekty wysokiego poziomu. Podczas schodzenia o jeden poziom obiekty te są definiowane jako typy obiektów Pythona. Alokator obiektów Pythona wywołuje interfejs API alokatora pamięci raw w celu przydzielenia pamięci dla obiektów w systemie operacyjnym. Ten interfejs API wykorzystuje alokatory ogólnego przeznaczenia, wykorzystując funkcję malloc języka C do przypisywania pamięci dla zmiennych

6.2. Garbage Collection:

6.2.1. JVM

Garbage Collection w maszynie JVM działa tylko na obszarze sterty. Pamięć przydzielona na stosie jest automatycznie usuwana po zdjęciu ramki. Istnieją różne algorytmy dla garbage collection.

- **Garbage-First (G1) Collector:** Działa na Young Generation, identyfikując i kolekcjonując obiekty krótkotrwałe. Ocalałe obiekty podlegają dalszej obróbce i mogą zostać przeniesione w inne miejsce w ramach Young Generation.
- **Mark and Sweep (Old Generation):** Identyfikuje i oznacza osiągalne obiekty w Old Geeration. Nieoznaczone obiekty są dealokowane, zwalniając pamięć.
- Nie ma konkretnych algorytmów dla Metaspace

6.2.2. 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 obsługi scenariuszy takich jak cykliczne odniesienia, identyfikowanie i zbieranie obiektów, które nie są już osiągalne.


Java
7
python™

Rozdział 7

Środowisko uruchomieniowe

Do wykonania pomiarów wydajnościowych 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


Java
8
python™

Rozdział 8

Porównanie czasu działania prostych fragmentów kodu

W celu porównania czasu dzialania prostych operacji takich jak dodawanie czy mnożenie stosujemy narzędzie perf_counter w przypadku Pythona i polecenie System.nanoTime() w pętli for w przypadku Javy.

8.1. Pusta pętla

W przypadku Javy do pomiaru czasu wykonania pustej pętli stosuję następujący kod [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(100000000);

        // 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 0.041550506 s. W przypadku wyłączenia JIT (java -Xint Empty) czas wykonania wzrasta do około 0,618245742 s.

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

```
from time import perf_counter_ns

def my_function():
  for i in range(100000000):
    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) / 1000000000, 's')
```

Wykonanie powyższego kodu zajmuje około 1.624114978 s.

Jest to więc 1.624114978/0,041550506 = 39,087730436 raza wolniej niż w przypadku Javy. Po wyłączeniu JIT, Java jest już jedynie 1.624114978/0,618245742=2,626973172 szybsza niż Python.

Warto w tym miejscu zaznaczyć, że gdyby kod funkcji my_function znajdował się poza nią, tak jak na poniższym listingu [Python]:

```
from time import perf_counter_ns

# Start the stopwatch / counter
t1_start = perf_counter_ns()

for i in range(100000000):
  pass

# Stop the stopwatch / counter
t1_stop = perf_counter_ns()

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

to program działałby zdecydowanie wolniej. W tym przypadku byłoby to 3.684830529 s zamiast 1.624114978 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 (3.684830529 s vs 1.624114978 s), wykorzytsanie zmiennych globalnych w pętli znacznie spowalnia program.

8.2. Dodawanie w pętli jednej liczby

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

```java
import java.io.*;

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

        count_function(100000000);

        // 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)
    {
        long j = 0;
        for (long i = 0; i < x; i++)
            j = j + 1;
    }
}
```

Wykonanie powyższego kodu zajmuje około 0.03390587 s. W przypadku wyłączenia JIT (java - Xint Addition) czas wykonania wzrasta do około 0.917541506 s.

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

```python
from time import perf_counter_ns

def my_function():
  sum = 0
  for i in range(100000000):
    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) / 1000000000, 's')
```

Wykonanie powyższego kodu zajmuje około 3.269221091 s.

Jest to więc 3.269221091/0.03390587=90,64205045 raza wolniej niż w przypadku Javy. Po wyłączeniu JIT, Java jest już jedynie 10,915374317/0.917541506=3,563022566 szybsza niż Python.

8.3. Duże liczby

W przypadku korzstania 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)
    {
        // Start measuring execution time
        long startTime = System.nanoTime();

        count_function(100000000);

        // 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)
    {
      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 = 2000;

```

```
37          BigInteger result3 = b2.pow(exponent2);
38
39          BigInteger sum = BigInteger.ZERO;
40
41          for (BigInteger bi = result; bi.compareTo(result2) < 0; bi = bi.add(
    BigInteger.ONE))
42              sum = sum.add(result3);
43      }
44  }
```

Czas jego wykonania wynosi 82.966079934 s.

Odpowiednik tego kodu w Pythonie wykonuje się w czasie 59.438710424 s [Python]:

```
1  from time import perf_counter_ns
2  import sys
3
4  def my_function():
5    sum = 0
6    r = 2000**2000
7    for i in range(100**100, 100**100 + 100000000):
8      sum = sum + r
9
10 # Start the stopwatch / counter
11 t1_start = perf_counter_ns()
12
13 my_function()
14
15 # Stop the stopwatch / counter
16 t1_stop = perf_counter_ns()
17
18 print((t1_stop-t1_start) / 1000000000, '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:

```
1   public static void count_function(long);
2     Code:
3        0: lconst_0
4        1: lstore_2
5        2: lload_2
6        3: lload_0
7        4: lcmp
8        5: ifge          15
9        8: lload_2
10       9: lconst_1
11      10: ladd
12      11: lstore_2
13      12: goto          2
14      15: return
```

i dla BigInteger:

```
1   public static void count_function(long);
2     Code:
3        0: getstatic     #35                 // Field java/math/BigInteger.
    ZERO:Ljava/math/BigInteger;
4        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.

8.4. Kolekcje

W następnych podpunktach mierzymy czas działania kolekcji.

8.4.1. Java Array vs Python List, NumPy Array

Weźmy pod uwagę kod łączenia dwóch tablic [Java].

```java
import java.util.Arrays;

public class Concat {

    public static void main(String[] args) {
        int[] array1 = new int[10000000];
        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;
        int[] result = new int[aLen + bLen];

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

        System.arraycopy(array1, 0, result, 0, aLen);
        System.arraycopy(array2, 0, result, aLen, bLen);

        // 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 0.011578673 s.

W przypadku Pythona skorzystamy z biblioteki NumPy. Gdy korzystamy z danych liczbowych, biblioteka ta pozwala na znaczne przyspieszenie działań. Dla porządku wywołamy również łączenie 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')

# Adding Numpy arrays
t2_start = perf_counter_ns()
result_array = np_array1 + np_array2
t2_stop = perf_counter_ns()

print((t2_stop-t2_start) / 1000000000, 's')
```

Czas działania wynosi 0.416666499 s dla łączenia list i 0.017764834 s dla łącznia dwóch tablic z biblioteki NumPy. Dzięki bibliotece NumPy kod w Pythonie wykonuje się w zbliżonym czasie do kodu w Javie.

8.4.2. Dodawanie String'ów do kolekcji

Dla poniższego programu [Java] mierzącego czas dodawania String'ów do HashSet, TreeSet, HashMap i TreeMap:

```
package com.avenuecode.snippet;

import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;

import java.io.*;

import java.util.*;

@State(Scope.Thread)
public class MyBenchmark {

    int x = 10000000;
    HashSet<String> counts = new HashSet<String>();
    TreeSet<String> counts2 = new TreeSet<String>();
    HashMap<String, String> map = new HashMap<>();
```

```
18      TreeMap<String, String> map2 = new TreeMap<>();
19      ArrayList<String> mylist = new ArrayList<String>();
20      String[] s = new String[x];
21      int y = -1;
22
23      /**
24       * Setup method to initialize data for the benchmark.
25       */
26      @Setup(Level.Trial)
27      public void setup() {
28          for (int i=0; i < x; i++)
29              mylist.add(String.valueOf(i));
30
31          Collections.shuffle(mylist);
32
33          for (int i=0; i < x; i++)
34              s[i] = mylist.get(i);
35      }
36
37      @Fork(value = 1, warmups = 1)
38      @Warmup(iterations = 1)
39      @Benchmark
40      @BenchmarkMode(Mode.AverageTime)
41      public long _0_HashSetAdd() {
42          for (int i=0; i < x; i++)
43              counts.add(s[i]);
44          y = 0;
45          return 0;
46      }
47
48      @Fork(value = 1, warmups = 1)
49      @Warmup(iterations = 1)
50      @Benchmark
51      @BenchmarkMode(Mode.AverageTime)
52      public long _1_TreeSetAdd() {
53          for (int i=0; i < x; i++)
54              counts2.add(s[i]);
55          y = 1;
56          return 0;
57      }
58      @Fork(value = 1, warmups = 1)
59      @Warmup(iterations = 1)
60      @Benchmark
61      @BenchmarkMode(Mode.AverageTime)
62      public long _2_HashMapPut() {
63          for (int i=0; i < x; i++)
64              map.put(s[i], s[i]);
65          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 }
```

uzyskujemy następujące czasy dodawania:

HashSet:

```
1  # Benchmark: com.avenuecode.snippet.MyBenchmark._0_HashSetAdd
2
3  # Run progress: 0,00% complete, ETA 00:08:00
4  # Warmup Fork: 1 of 1
5  # Warmup Iteration   1: 1,333 s/op
6  Iteration   1: 0,952 s/op
7  Iteration   2: 0,906 s/op
8  Iteration   3: 1,027 s/op
9  Iteration   4: 1,045 s/op
10 Iteration   5: 1,048 s/op
11
12 # Run progress: 12,50% complete, ETA 00:07:29
13 # Fork: 1 of 1
14 # Warmup Iteration   1: 1,496 s/op
15 Iteration   1: 1,084 s/op
16 Iteration   2: 1,049 s/op
17 Iteration   3: 1,011 s/op
18 Iteration   4: 0,949 s/op
19 Iteration   5: 0,942 s/op
```

TreeSet:

```
1  # Benchmark: com.avenuecode.snippet.MyBenchmark._1_TreeSetAdd
2
3  # 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
```

W przypadku Pythona dla programu [Python]:

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

def my_function():
  s = list(range(0,10000000))

  for i in range(0, 10000000):
    s[i] = str(s[i])

  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, 10000000):
      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, 10000000):
      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, 10000000):
      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, 10000000):
      dict2[s[i]] = s[i]

    # Stop the stopwatch / counter
    t1_stop = perf_counter()

    print("SortedDict: ", t1_stop-t1_start)

my_function()
```

uzyskujemy następujące czasy wstawiania do kolekcji:

```
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:

```
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.

```
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)
```

8.4.3. Sprawdzanie czy kolekcja zawiera Stringi

Korzystając z programu podobnego do używanego do szacowania czasu dodawania String'ów do kolekcji [Java] [Python], mierzę czas sprawdzania czy String należy do kolekcji.

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)
```

8.4.4. Usuwanie String'ów z kolekcji

Korzystając z programu podobnego do używanego do szacowania czasu dodawania String'ów do kolekcji [Java] [Python], mierzę 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)
```


Java
9
python™

Rozdział 9

Porównanie czasu działania programów

9.1. 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 porównuję czas działania wybranych programów w Javie i Pythonie pochodzących ze strony [CLBG].

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 przyjąłem liczbę 10000. Dokładniejszy opis: [8]

Java

W przypadku Javy walltime programu [Java] przy wykorzystaniu polecenia runexec wynosi 5.927161733000048s. 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

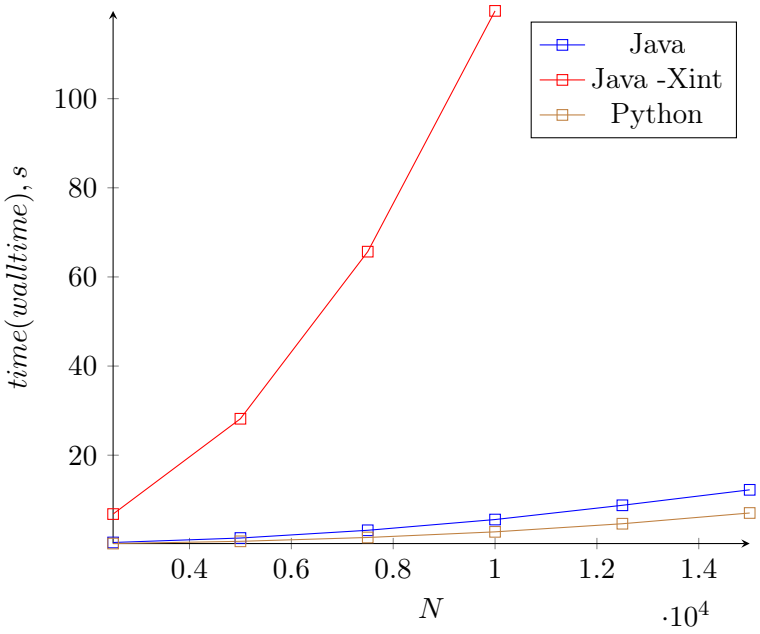
W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wynosi 2.8576857300001848s. Komplet wyników:

```
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
```

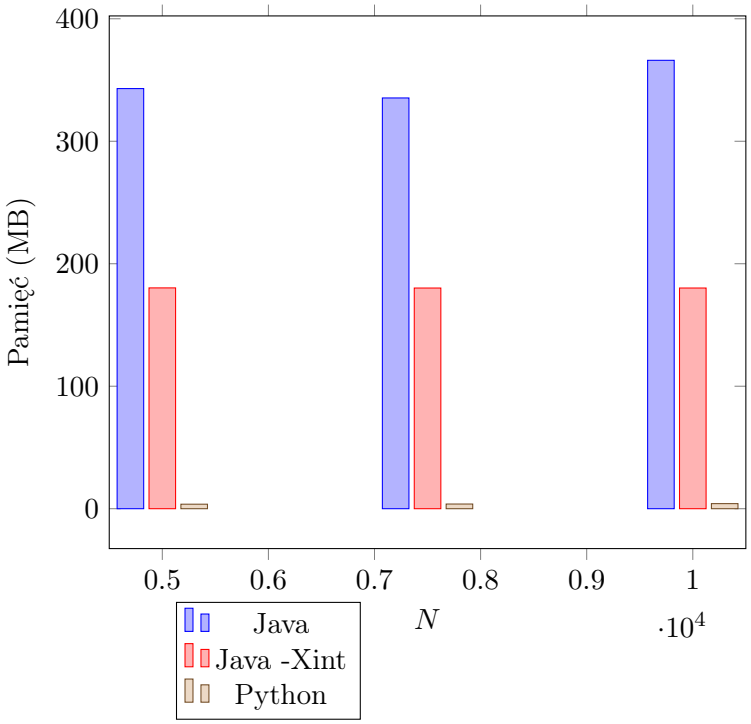
Jak widać w przypadku tego programu Java jest wolniejsza od Pythona. Może to wynikać z wykorzystania do obliczeń liczb typu BigInteger, które jak pokazaliśmy działają wolniej niż duże liczby w Pythonie.

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

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.



Jak widać Java potrzebuje do wykonania programu znacznie wiekszej ilości pamięci niż Python niezależnie od wejściowej wartości 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 wykona-

nia polega na alokowaniu i dealokowaniu wielu drzew przy jednoczesnym zliczaniu liczby ich wierzchołków.

Podstawową liczbę dla pomiarów stanowi 21. Dokładniejszy opis: [9].

Java

W przypadku Javy walltime programu [Java #7] przy wykorzystaniu polecenia runexec wynosi 2.2797591759999705s. 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
```

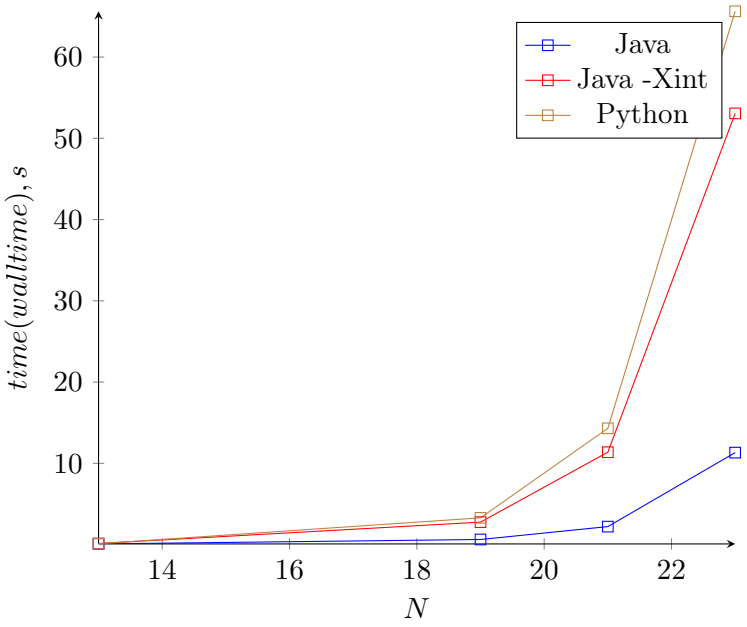
Python

W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wynosi 15.58560308199958s. Komplet wyników:

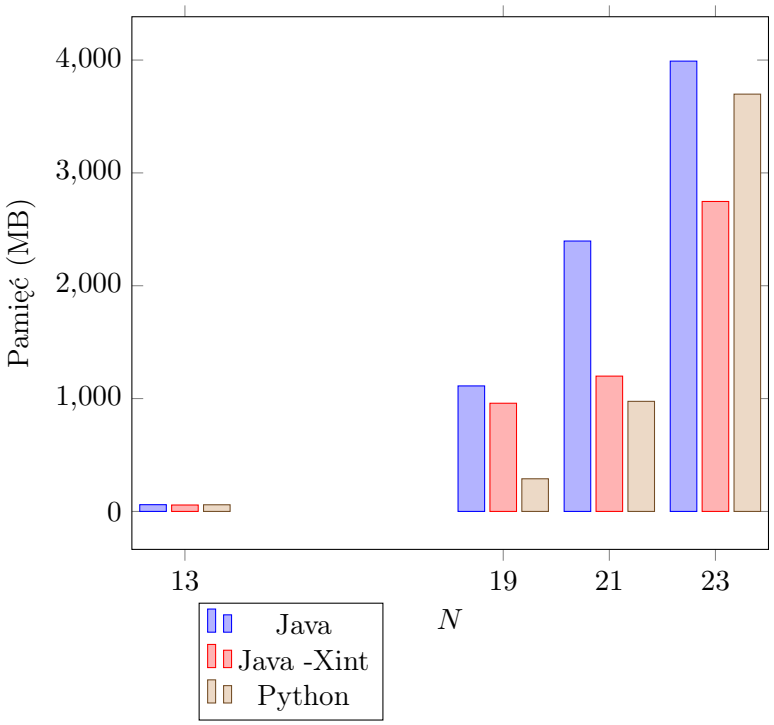
```
starttime=2024-12-20T19:59:42.184918+01:00
returnvalue=0
walltime=15.58560308199958s
cputime=142.584223976s
cputime-cpu0=11.524329568s
cputime-cpu1=11.418168383s
cputime-cpu10=11.543249594s
cputime-cpu11=11.336778209s
cputime-cpu2=11.776000717s
cputime-cpu3=11.450827422s
cputime-cpu4=14.548266545s
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 15.58560308199958s/2.2797591759999705s=6,836512929 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.



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 przyjąłem liczbę 12. Dokładniejszy opis: [10].

Java

W przypadku Javy walltime programu [Java] przy wykorzystaniu polecenia runexec wynosi 3.559818760999974s. 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

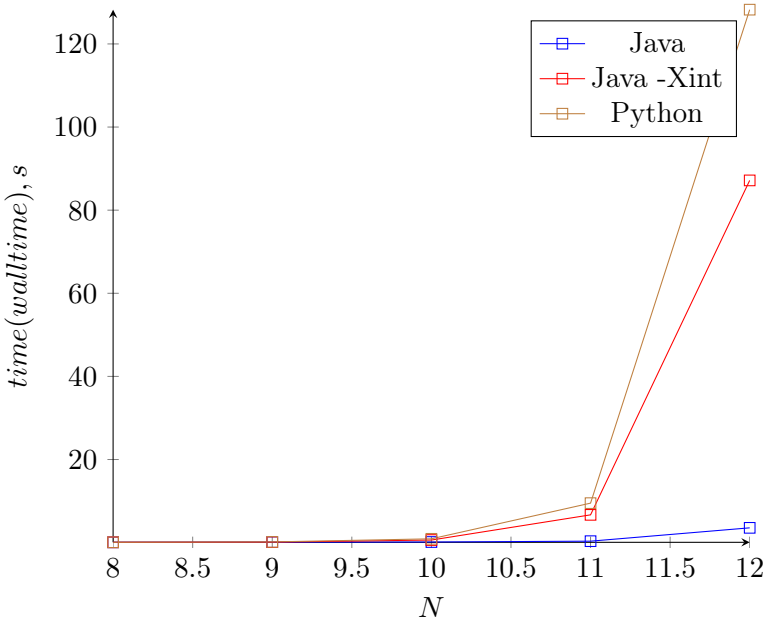
W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wynosi 128.24714689400002s. Komplet wyników:

```
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
```

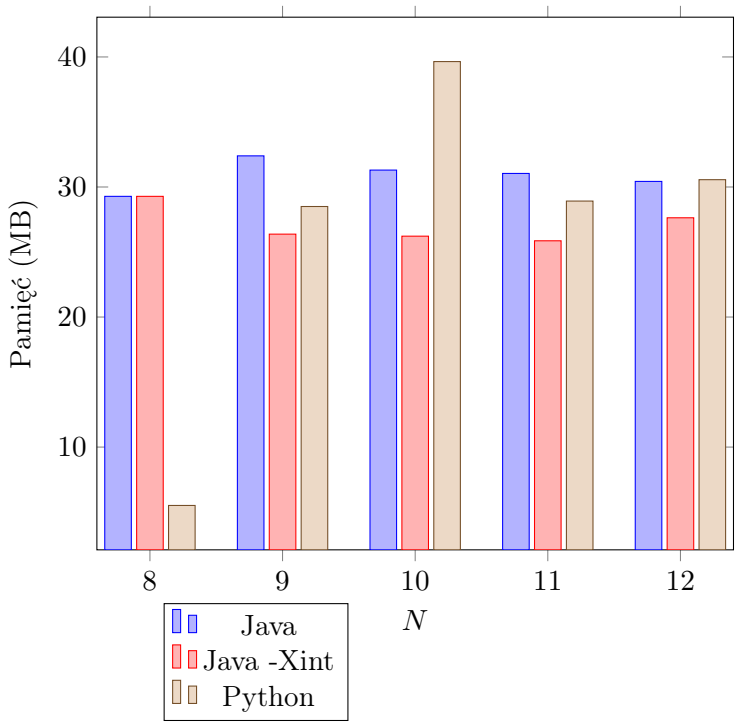
```
memory=32047104B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla N = 12 jest to 128.24714689400002s/3.559818760999974s=36,026313558 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.



9.2.4. N-body

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

Dla pomiarów za bazowe n przyjąłem liczbę 50000000. Dokładniejszy opis: [11].

Java

W przypadku Javy walltime programu [Java #4] przy wykorzystaniu polecenia runexec wynosi 4.217370437999989s. 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
```

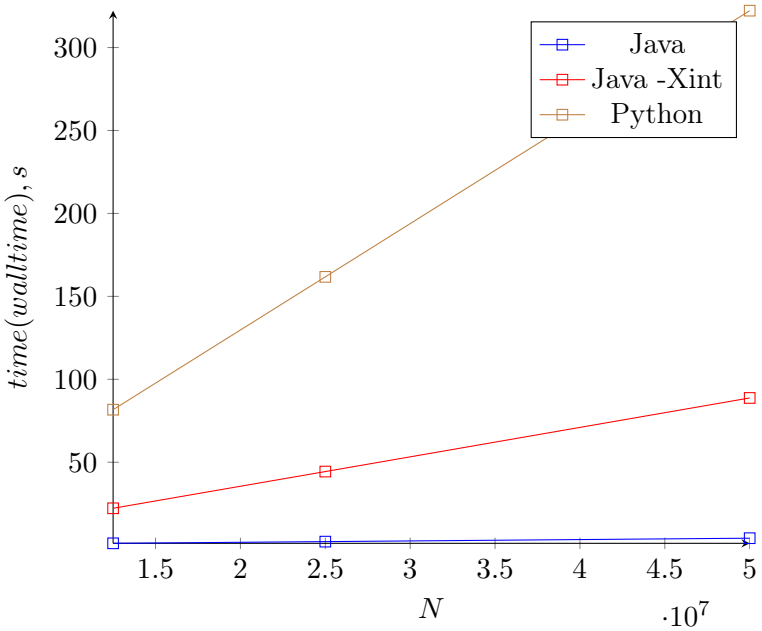
Python

W przypadku Pythona walltime programu [Python 3] przy wykorzystaniu polecenia runexec wynosi 322.2353108489999s. Komplet wyników:

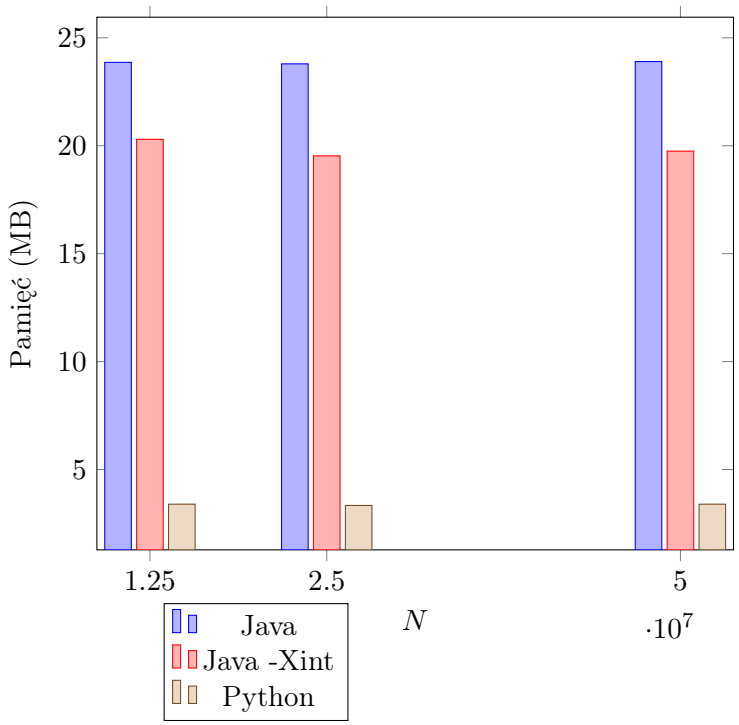
```
starttime=2024-12-25T11:57:15.773324+01:00
returnvalue=0
walltime=322.2353108489999s
cputime=322.226826032s
cputime-cpu0=97.789804078s
cputime-cpu10=0.012211654s
cputime-cpu11=72.973624307s
cputime-cpu3=1.512004426s
cputime-cpu5=22.195560776s
cputime-cpu6=127.743620791s
memory=3559424B
blkio-read=53248B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n = 50000000 jest to 322.2353108489999s/4.217370437999989s=76,406688857 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.



9.2.5. fasta

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

Dla pomiarów za argument wejściowy przyjąłem 25000000. Dokładniejszy opis: [12].

Java

W przypadku Javy walltime programu [Java #6] przy wykorzystaniu polecenia runexec wynosi 1.1648875109999608s. 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
```

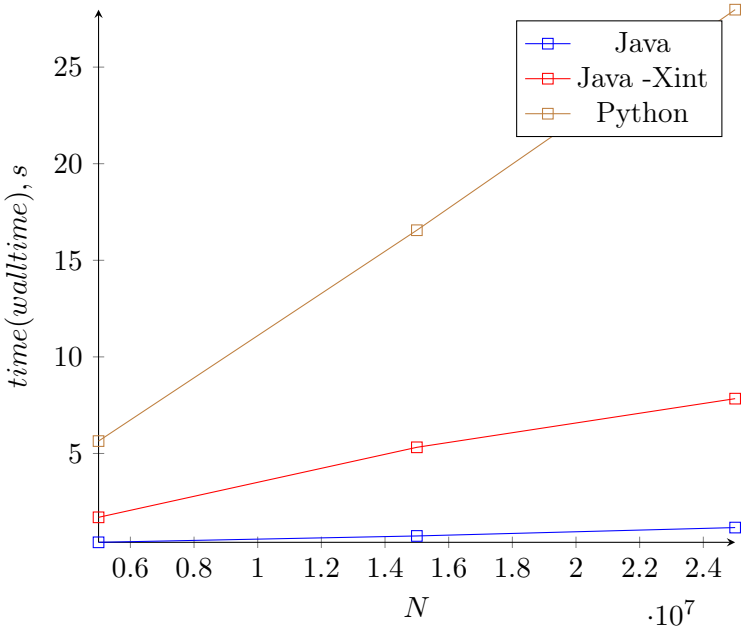
Python

W przypadku Pythona walltime programu [Python 3 #5] przy wykorzystaniu polecenia runexec wynosi 27.973746784000014s. Komplet wyników:

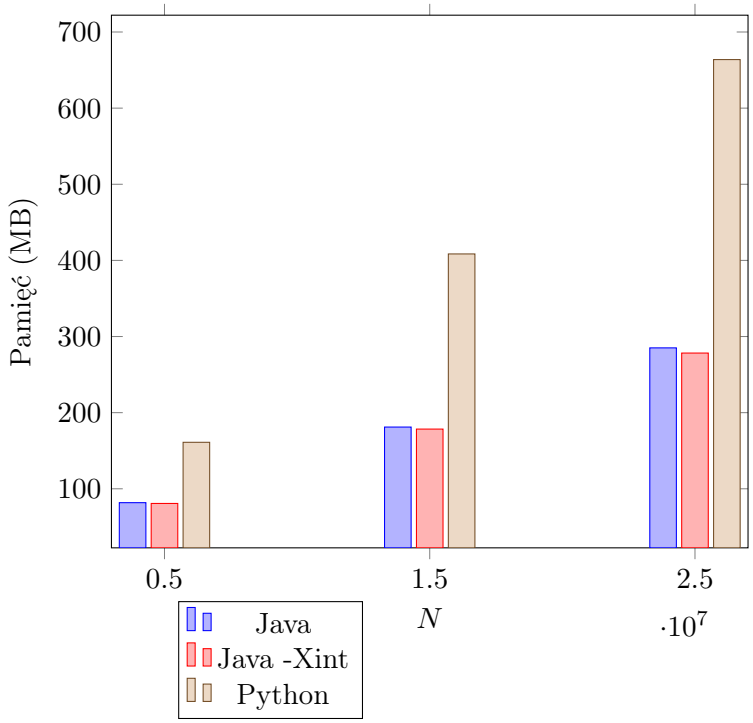
```
starttime=2024-12-25T19:56:00.767456+01:00
returnvalue=0
walltime=27.973746784000014s
cputime=49.00618006s
cputime-cpu0=2.295515155s
cputime-cpu1=11.375523946s
cputime-cpu10=5.328212206s
cputime-cpu11=13.194317161s
cputime-cpu2=2.877192826s
cputime-cpu3=1.743911116s
cputime-cpu4=0.995536218s
cputime-cpu5=4.384794556s
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 27.973746784000014s/1.1648875109999608s=24,014118548 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.



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].

Java

W przypadku Javy walltime programu [Java #6] przy wykorzystaniu polecenia runexec wynosi 2.366201705000094s. 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
```

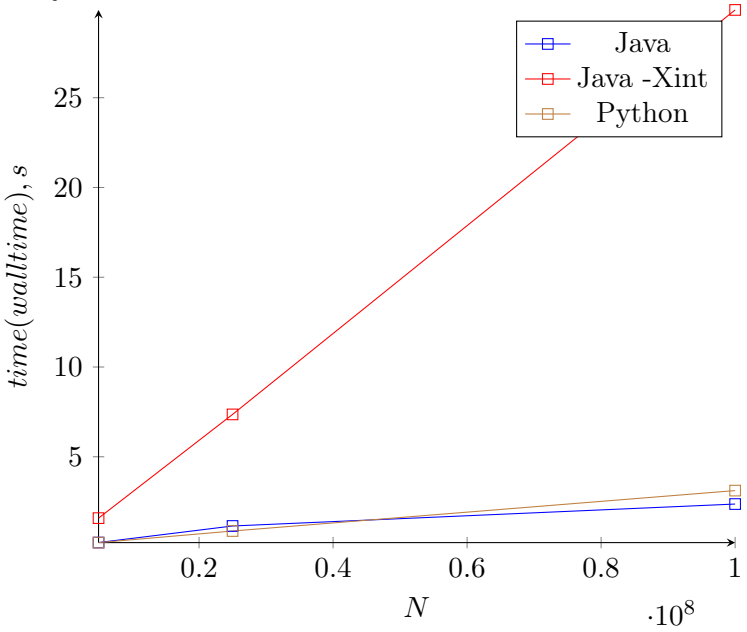
Python

W przypadku Pythona walltime programu [Python 3 #5] przy wykorzystaniu polecenia runexec wynosi 3.1158195779999005s. 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
```

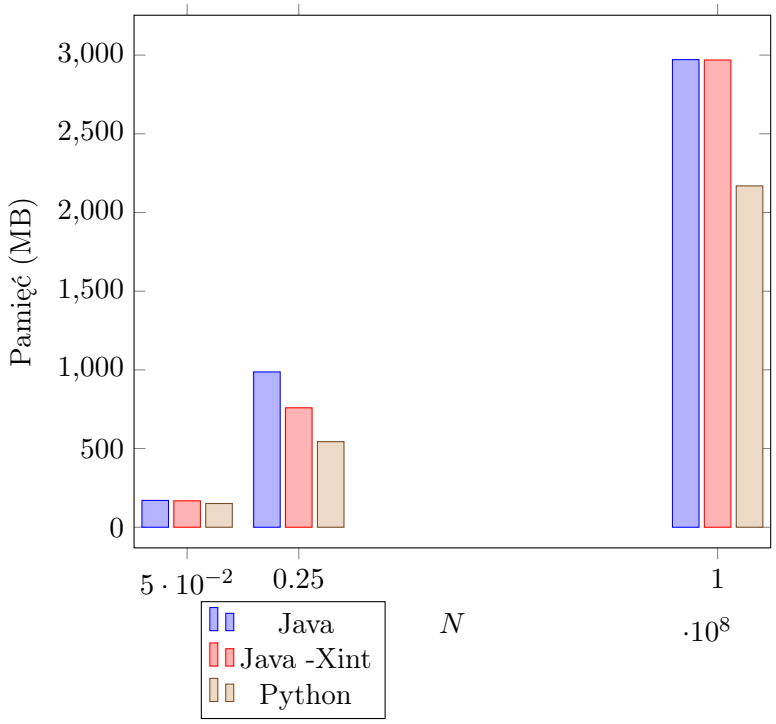
Jak widać w przypadku tego programu Java jest stosunkowo niewiele szybsza od Pythona. Dla N = 100000001 jest to 3.1158195779999005s/2.366201705000094s=1,316802186 raza szybciej.

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.



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 przyjąłem 16000. Dokładniejszy opis: [14].

Java

W przypadku Javy walltime programu [Java #2] przy wykorzystaniu polecenia runexec wynosi 1.1342887529999643s. 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
```

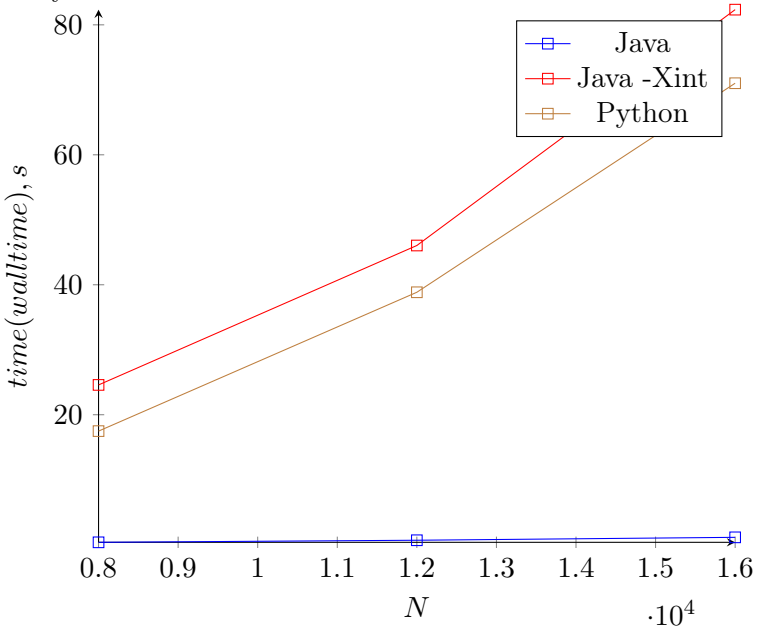
Python

W przypadku Pythona walltime programu [Python 3 #7] przy wykorzystaniu polecenia runexec wynosi 71.01652564599999s. Komplet wyników:

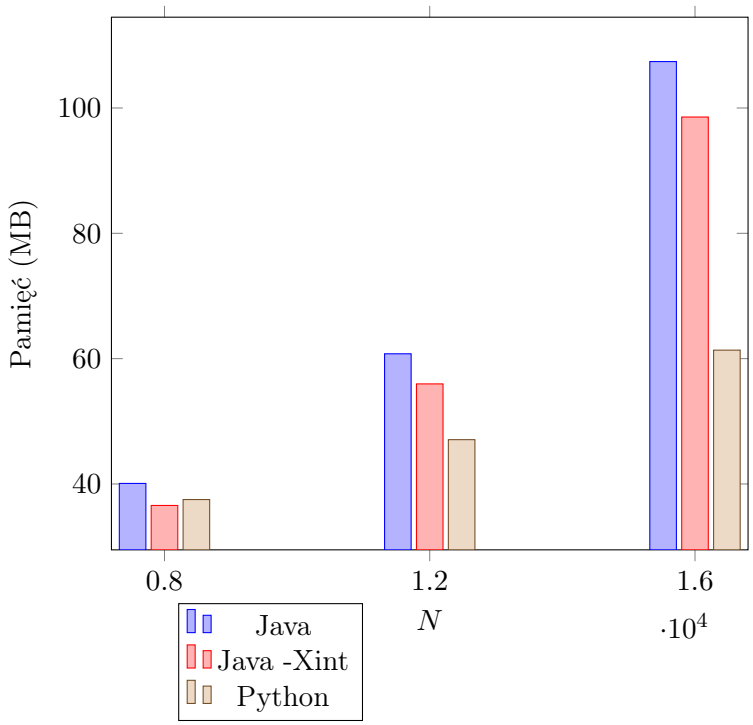
```
starttime=2024-12-25T20:12:15.167569+01:00
returnvalue=0
walltime=71.01652564599999s
cputime=843.977451867s
cputime-cpu0=70.294594650s
cputime-cpu1=70.500735403s
cputime-cpu10=69.902568873s
cputime-cpu11=70.183822083s
cputime-cpu2=70.520069691s
cputime-cpu3=70.074994223s
cputime-cpu4=70.495964229s
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 agumentu 16000 jest to 71.01652564599999s/1.1342887529999643s=62,608859921 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.



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 przyjąłem 5500. Dokładniejszy opis: [15].

Java

W przypadku Javy walltime programu [Java #3] przy wykorzystaniu polecenia runexec wynosi 0.6107691810002507s. 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
```

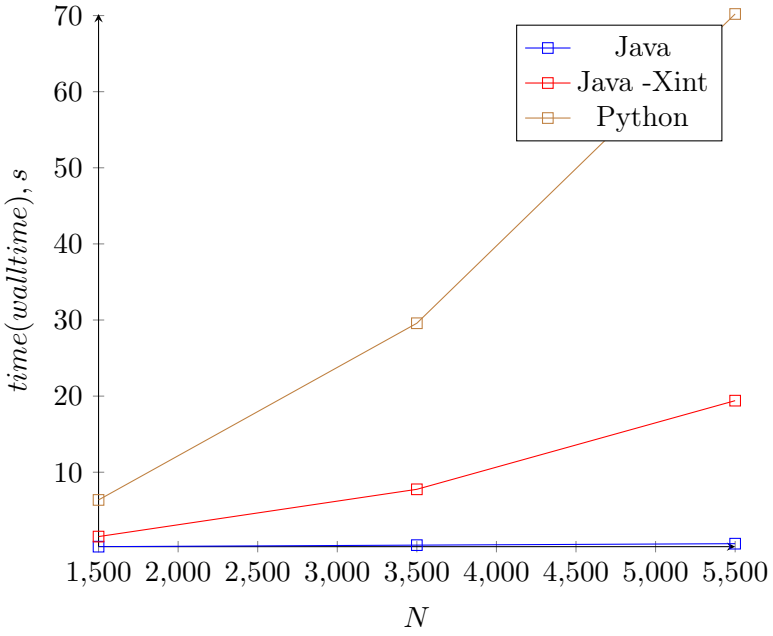
Python

W przypadku Pythona walltime programu [Python 3 #4] przy wykorzystaniu polecenia runexec wynosi 70.20847665300016s. Komplet wyników:

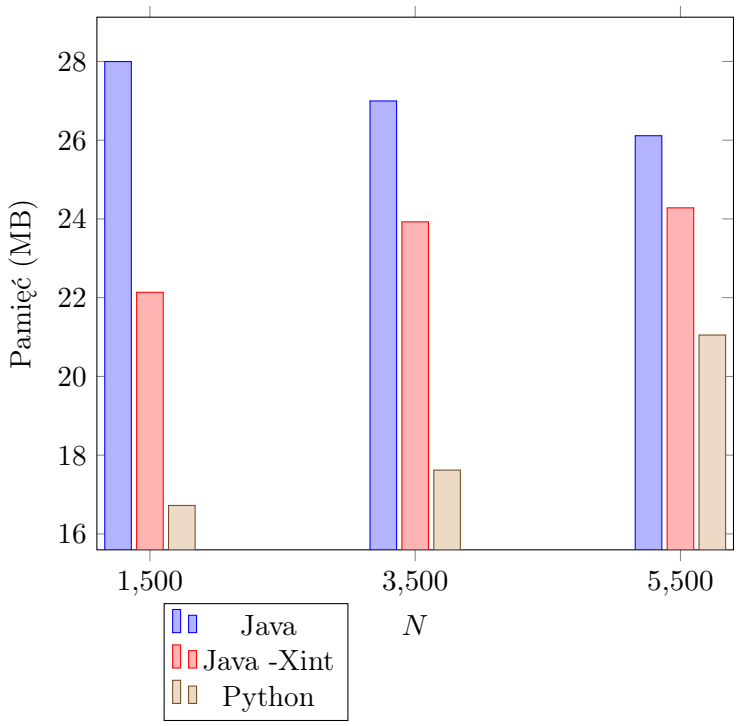
```
starttime=2024-12-25T20:24:30.374322+01:00
returnvalue=0
walltime=70.20847665300016s
cputime=277.682246331s
cputime-cpu0=23.436026862s
cputime-cpu1=33.131612600s
cputime-cpu10=14.584188744s
cputime-cpu11=12.294572871s
cputime-cpu2=16.686515146s
cputime-cpu3=39.313063969s
cputime-cpu4=46.249951552s
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 5500 jest to 70.20847665300016s/0.6107691810002507s=114,950915726 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.



9.3. Przerobione programy ze strony geeksforgeeks.org

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

9.3.1. Mergesort

Program Mergesort sortuje przy pomocy algorytmu sortowania mergesort liste/tablicę, w której znajdują się liczby uporządkowane malejąco: od n do 1. W pomiarach za bazowe n przyjąłem 1000000.

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);
        }
```

```
75      }
76
77      // Driver code
78      public static void main(String args[])
79      {
80          int n = Integer.parseInt(args[0]);
81
82          int arr[] = new int[n];
83          for (int i = 0; i < n; i++)
84            arr[i] = n - i;
85
86          sort(arr, 0, n - 1);
87      }
88  }
```

W przypadku Javy walltime programu dls n=1000000 przy wykorzystaniu polecenia runexec wynosi 0.11144095099962215s. 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]:

```
1  from sys import argv
2
3  def merge(arr, left, mid, right):
4      n1 = mid - left + 1
5      n2 =e j < n2:
6          arr[k] = R[j]
7          j += 1
8          k += 1
9
10 def merge_sort(arr, left, right):
11     if left < right:
12         mid = (left + right) // 2
13
14         merge_sort(arr, left, mid)
15         merge_sort(arr, mid + 1, right)
16         merge(arr, left, mid, right)
17
18 def main():
19     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

    # Create temp arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]

    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = left  # Initial index of merged subarray

    # Merge the temp arrays back
    # into arr[left..right]
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[],
    # if there are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[],
    # if there are any
    while 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
```

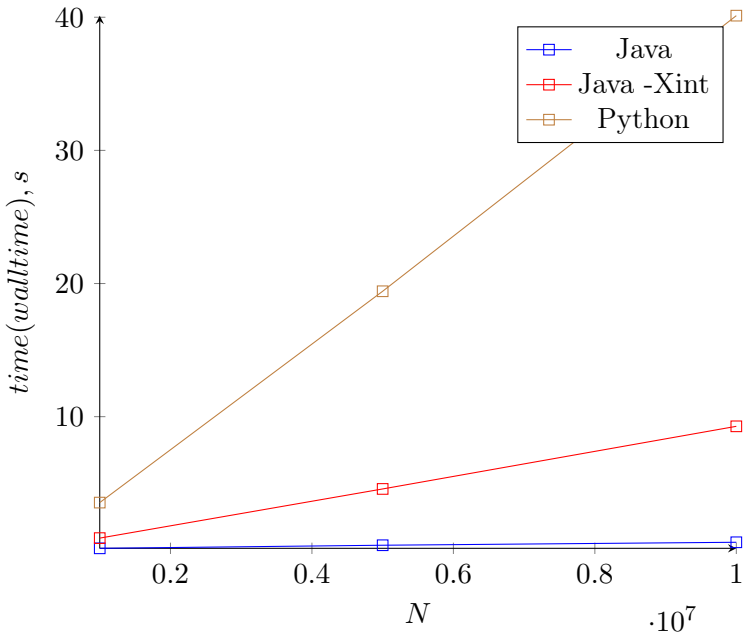
```
80
81      merge_sort(arr, 0, n - 1)
82
83  main()
```

W przypadku Pythona walltime programu dla n=1000000 przy wykorzystaniu polecenia runexec wynosi 3.5459886850003386s. Komplet wyników:

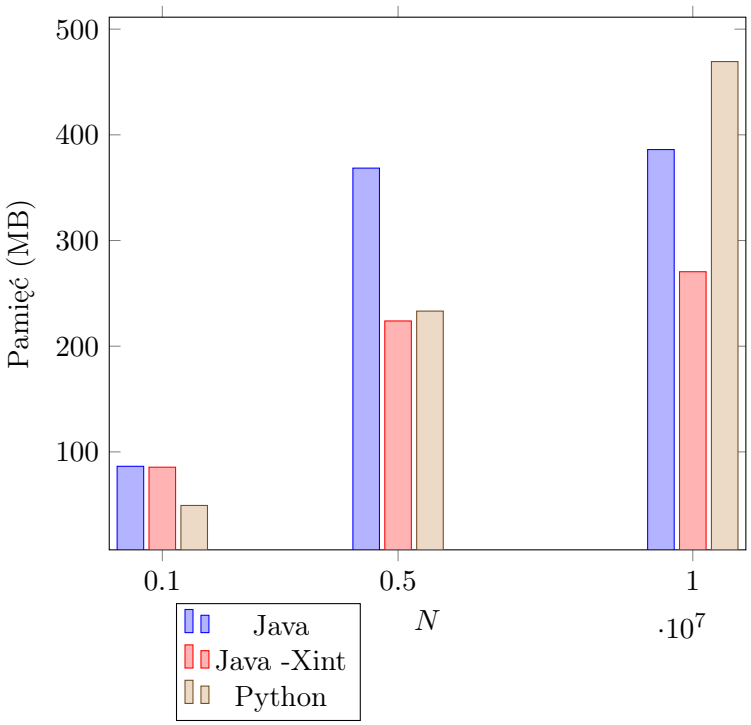
```
starttime=2024-12-23T18:26:32.246044+01:00
returnvalue=0
walltime=3.5459886850003386s
cputime=3.545602191s
cputime-cpu11=3.540892358s
cputime-cpu5=0.004709833s
memory=51707904B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n=1000000 jest to 3.5459886850003386s/0.11144095099962215=31,819440279 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.



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("[");
        //    for (int i = 0; i < res.size(); i++) {
        //        System.out.print(res.get(i));
        //       if (i != res.size() - 1)
        //            System.out.print(", ");
        //    }
        //    System.out.println("]");
        //}
    }
}
```

W przypadku Javy walltime programu dls n=13 przy wykorzystaniu polecenia runexec wynosi 0.4590756580000743s. 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]:

```
from sys import argv

# Utility function for solving the N-Queens
# problem using backtracking.
def nQueenUtil(j, n, board, result,
               rows, diag1, diag2):
    if j > n:

        # A solution is found
        result.append(board.copy())
        return
    for i in range(1, n +1):
        if not rows[i] and not diag1[i + j] and \
           not diag2[i - j + n]:

            # Place queen
            rows[i] = diag1[i + j] = \
                      diag2[i - j + n] = True
            board.append(i)

            # Recurse to next column
            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

```

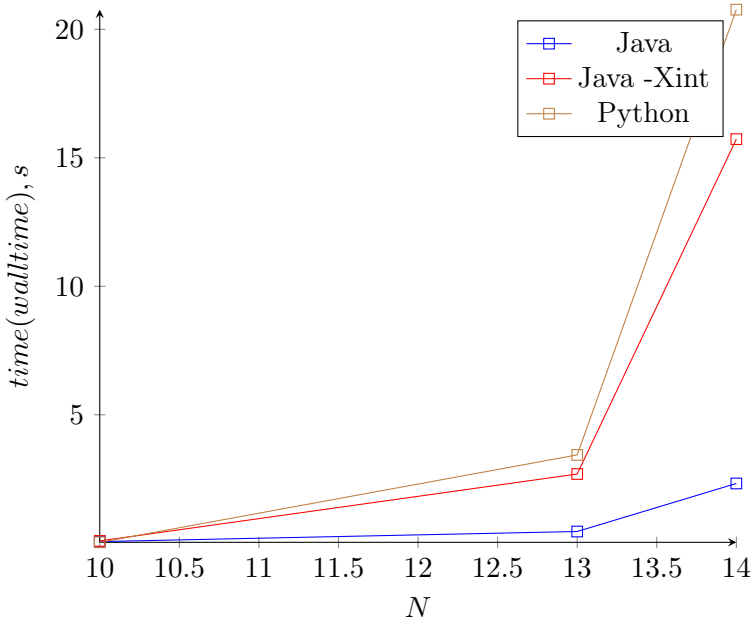
```
44 def main():
45     n=int(argv[1])
46     result = nQueen(n)
47 #    for res in result:
48 #        print("[", end="")
49 #        for i in range(len(res)):
50 #            print(res[i], end="")
51 #            if i != len(res)-1:
52 #                print(", ", end="")
53 #        print("]")
54
55 main()
```

W przypadku Pythona walltime programu dla n=13 przy wykorzystaniu polecenia runexec wynosi 3.439188954999736s. Komplet wyników:

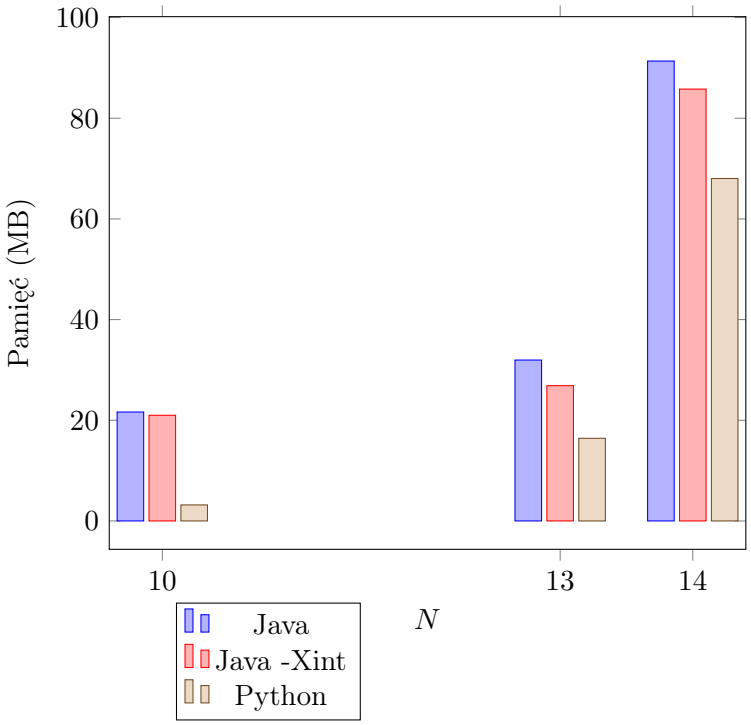
```
starttime=2024-12-23T18:45:24.910125+01:00
returnvalue=0
walltime=3.439188954999736s
cputime=3.401129201s
cputime-cpu0=1.877275741s
cputime-cpu6=1.523853460s
memory=17215488B
blkio-read=0B
blkio-write=0B
```

Jak widać w przypadku tego programu Java jest wyraźnie szybsza od Pythona. Dla n=13 jest to 3.439188954999736s/0.4590756580000743s=7,491551545 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.


100
80
60
40
20
0
Pamięć (MB)
10
13
14
N
Java
Java -Xint
Python


Java
10
python™

Rozdział 10

Podsumowanie

W pracy porównałem bajtkody i maszyny wirtualne Javy i Pythona. Omówiłem architekturę JVM i PVM, JIT Javy i stosowane przez niego optymalizacje oraz dopiero co dodany JIT do Pythona 3.13. Dokonałem analizy porównawczej kolekcji Javy i Pythona. Porównałem czas działania i zużycie pamięci dwóch programów (Mergesort, problem n hetmanów) ze strony geeksforgeeks [6] w Javie i Pythonie. Zbadałem 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.

Doszedłem do następujących wniosków:

- Piętą achillesową Javy jest obsługa dużych liczb. Klasa BigInteger Javy działa wolniej niż duże liczby w Pythonie co sprawia, że programy na niej oparte są również wolniejsze od analogicznych pogramów w Pythonie. Z pomocą przychodzą tu Javie 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 złożonoć 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.

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.