AKADEMIA TARNOWSKA
Wydział Politechniczny

Kierunek: ***Informatyka***

Specjalność: Informatyka Stosowana

Specjalizacja: Inżynieria Oprogramowania

2023/2024

Bartosz Kmiecik

PRACA INŻYNIERSKA

Analiza wydajności wybranych najpopularniejszych języków programowania

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, 2024

Spis treści

1. Wstęp **3**

1.1. Motywacja 3

1.2. Cel pracy 3

1.3. Zakres pracy 3

2. Języki programowania **5**

2.1. C++ 6

2.2. Java 7

2.3. Python 8

3. Środowiska programistyczne **11**

3.1. Visual Studio 11

3.2. IntelliJ IDEA 12

3.3. PyCharm 13

4. Algorytmy **14**

4.1. Wyszukiwanie n-tej liczby ciągu Fibonacciego 14

4.2. Algorytm sortowania szybkiego 15

4.3. Algorytm Blowfish 15

4.4. Algorytm kompresji RLE 16

4.5. Algorytm kompresji LZW 17

5. Specyfikacja sprzętowa **18**

6. Wyniki badań **19**

6.1. Wyniki dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej 20

6.1.1 C++ 20

6.1.2 Python 22

6.1.3 Java 24

6.2. Wyniki dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej 27

6.2.1 C++ 27
6.2.2 Python 29
6.2.3 Java 31
6.3. Wyniki dla algorytm sortowania szybkiego 34
6.3.1 C++ 34
6.3.2 Python 36
6.3.3 Java 38
6.4. Wyniki dla algorytmu Blowfish 41
6.4.1 C++ 41
6.4.2 Python 43
6.4.3 Java 45
6.5. Wyniki dla algorytmu kompresji metodą RLE 48
6.5.1 C++ 48
6.5.2 Python 50
6.5.3 Java 52
6.6. Wyniki dla algorytmu kompresji metodą LZW 55
6.6.1 C++ 55
6.6.2 Python 57
6.6.3 Java 59
7. Podsumowanie i wnioski **62**
Bibliografia
Spis rysunków
Spis tabel

1. Wstęp

W dzisiejszych czasach charakteryzujących się niezwykle dynamicznym postępem w dziedzinach informatycznych jednym z kluczowych aspektów, które towarzyszą tworzeniu oprogramowanie, jest wybór języka. Jest to podstawa, na której budowany jest projekt mająca wpływ na efektywność i skalowalność.

1.1. Motywacja

W kontekście problematyki związanej z wyborem języka programowania praca ta została poświęcona porównaniu wydajności wybranych najpopularniejszych języków programowania. Kryteria, jakimi kierowano się przy wyborze, nie są przypadkowe i zostały dokładnie opisane w kolejnym rozdziale.

1.2. Cel pracy

Celem pracy jest przeprowadzenie kompleksowego porównania wydajności trzech wybranych najpopularniejszych języków programowania na przykładzie różnych algorytmów. Praca ta pozwoli wyłonić język, który najbardziej będzie się nadawać do zastosowania w sytuacjach, w których liczy się szybkość wykonywania kodu oraz zużycie zasobów komputera. Dzięki tej pracy poszerzona zostanie również wiedza na temat każdego z tych języków oraz tego, jaką rolę odgrywa on w dzisiejszych warunkach branży tworzenia oprogramowania.

1.3. Zakres pracy

Praca została rozpoczęta od wyboru trzech języków programowania, które będą poddane testom. Metoda ich wyboru została objaśniona w rozdziale *2. Języki programowania.* Dla każdego z języków wybrane zostało zintegrowane środowisko programistyczne (IDE), które było najbardziej odpowiadającym specyfice pisania kodu w danym języku. Następnie wybranych zostało pięć algorytmów charakteryzujących się różnym zastosowaniem oraz stopniem skomplikowania. Dla każdego z nich została napisana implementacja w trzech wcześniej wybranych językach programowania. Dalszym krokiem

pracy było przetestowanie każdej z implementacji przy pomocy narzędzi systemu *Windows 10* pod kątem zużycia zasobów komputera i czasu wykonania programu. Na koniec otrzymane wyniki zostały poddane analizie.

2. Języki programowania

Pierwszym problemem, jaki przyszło rozwiązać w czasie pisania pracy inżynierskiej był wybór języków programowania, których wydajność będzie później testowana i porównywana na podstawie różnych algorytmów. Tu posłużono się ankietą przeprowadzoną na witrynie Stack Overflow w 2022 roku[10]. Została ona przeprowadzona na grupie 73 268 osób związanych z programowaniem. W rozdziale *Technology* w podrozdziale *Most popular technologies* w sekcji *Programming, scripting, and markup languages* możemy znaleźć następujące informacje:

- Wśród profesjonalnych programistów najczęściej wskazywanymi językami były:
    1. JavaScript - 67,9%
    2. HTML/CSS - 54,93%
    3. SQL - 52,64%
    4. Python - 43,51%
    5. TypeScript - 40,08%
    6. Java - 33,4%
    7. C# - 29,72%
    8. Bash/Shell - 29,47%
    9. PHP - 21,42%
    10. C++ - 20,17%
- Wśród osób uczących się programować najczęściej wskazywanymi językami były:
    1. HTML/CSS - 62,59%
    2. JavaScript - 59,79%
    3. Python - 58,38%
    4. Java - 38,67%
    5. SQL - 37,8%
    6. C++ - 34,69%
    7. C - 31,94%
    8. C# - 20,45%
    9. Bash/Shell - 19,37%
    10. PHP - 18,82%
- Wśród wszystkich respondentów najczęściej wskazywanymi językami były:
    1. JavaScript - 65,36%
    2. HTML/CSS - 55,08%

3. SQL - 49,43%
4. Python - 48,07%
5. TypeScript - 34,83%
6. Java - 33,27%
7. Bash/Shell - 29,07%
8. C# - 27,98%
9. C++ - 22,55%
10. PHP - 20,87%

Jak widać wiele języków powtarza się we wszystkich kategoriach. Z tej grupy zdecydowano się wybrać technologie użyte przy pisaniu pracy inżynierskiej. Od razu można wyeliminować język HTML, ponieważ nie jest on językiem programowania, a językiem znaczników[11]. Taka sama sytuacja ma miejsce w przypadku SQL, ponieważ jest to język zapytań[12] oraz Bash będącego językiem komend w systemach UNIX[15]. Nie zostały wybrane również JavaScript, ponieważ, jak sama nazwa wskazuje, jest on językiem skryptowym[13] oraz TypeScript będący nadzbiorem języka JavaScript[14]. Wybór padł na języki Python, Java i C++ (jako bardziej rozbudowany od C i lepiej nadający się do aplikacji konsolowych od C#[3]).

2.1. C++

Twórcą języka C++ jest duński informatyk Bjarne Stroustrup. Pracę rozpoczął w 1979 roku i pierwotnie język nosił nazwę *C z Klasami.* Motywacją dla rozpoczęcia pracy nad językiem była chęć pisania wydajnych programów w stylu zalecanym przez Simula67. Aby to osiągnąć, do języka C dodane zostały funkcję dające możliwość lepszego sprawdzania danych, programowanie w paradygmacie obiektowym i abstrakcję w ramach tego paradygmatu.

Nazwę, którą znamy obecnie nadano pod koniec 1983 roku, kiedy technologia ta była wykorzystywana przez AT&T. Firma ta wspierała rozwój języka C++, ponieważ w tamtym okresie zajęta była rozwojem systemów o większej złożoności i mających większe wymagania dotyczące niezawodności od większości organizacji. Nazwa C++ została nadana przez Ricka Mascittiego i oznacza ewolucyjny charakter zmian wprowadzonych w C++ w stosunku do języka C.

W roku 1985 została opublikowana pierwsza komercyjna wersja języka C++. W latach osiemdziesiątych dodano szablony i obsługę wyjątków. Pierwszym standardem ISO

języka C++ był C++ 98. Od tamtego czasu powstało jeszcze pięć standardów: C++03, C++11, C++14, C++17, a najbardziej aktualnym jest C++ 20 opublikowany w grudniu 2020 roku.

Z początku najważniejszą cechą różniącą język C++ od języka C była możliwość stosowania klas i wprowadzenie pojęcia szeroko pojętej obiektowości. W późniejszych etapach rozwoju wprowadzono wiele innych modyfikacji i ulepszeń. Ich celem było sprawienie, żeby C++ przeistoczył się w język wygodniejszy do użytkowania dla programistów oraz bardziej elastyczny w jego wykorzystaniu od swojego pierwowzoru w postaci języka C. Pomimo wszystkich zmian i ponad czterdziestu lat rozwoju w technologii C++ w dalszym ciągu jednak kładzie się duży nacisk na zachowanie możliwie jak największej zgodności z językiem C, czego świadectwem może być możliwość wykorzystywania bibliotek napisanych z myślą o C w programach opartych na C++.

Jedną z ważnych właściwości języka C++ jest możliwość pisania kodu z wykorzystaniem wielu paradygmatów programowania, takich jak paradygmat proceduralny, obiektowy, generyczny, funkcyjny i modularny. W języku C++ można zatem stosować jednocześnie kilka różnych stylów programowania. Możliwe jest również programowanie na poziomie asemblera[3][4].

2.2. Java

Język Java nierozerwalnie związany jest z językiem C, z którego czerpie składnie, oraz C++, od którego zaczerpnął większość elementów powiązanych z obiektowością. Język ten posiada również kilka aspektów charakterystycznych powstałych w odpowiedzi na jej poprzedników oraz po wyciągnięciu wniosków z doświadczeń z innymi technologiami.

Historia Javy rozpoczyna się w roku 1991, kiedy to Mike Sheridan, Ed Frank, Chris Warth, Patrick Naughton i James Gosling opracowali opisywany w tym rozdziale język, który do 1995 roku nosił nazwę Oak. Głównym założeniem towarzyszącym powstawaniu tej technologii było wprowadzenie do użytku języka, który mógłby działać niezależnie od platformy sprzętowej, a jego głównym celem było oprogramowanie elektroniki użytkowej, takiej jak piloty zdalnego sterowania, czy kuchenki mikrofalowe[1].

Drugim i mogłoby się wydawać, że ważniejszym czynnikiem rozwoju języka Java są strony WWW. Początki dynamicznego rozwoju internetu przypadają właśnie na okres, w którym wprowadzono do użytku omawianą w tym rozdziale technologię. Internet nie

rozróżnia rodzajów procesorów i systemów operacyjnych, więc Java idealnie sprawdzała się w tworzeniu stron internetowych.

Od 1993 roku dla inżynierów pracujących nad projektem języka Java oczywistym stał się fakt, że problemy związane z przenośnością kontrolerów urządzeń elektronicznych będzie również występował w przypadku kodu programów dostępnych z pomocą internetu. Przy niewielkim nakładzie środków przeniesiono główny nacisk na rozwój języka z kierunku elektroniki użytkowej na Internet.

Dziedziczenie przez język Java elementów z języków C i C++ jest działaniem celowym. Inżynierowie pracujący nad rozwojem tej technologii od początku wiedzieli, że zapożyczenie składni z C oraz zasad obiektowości z C++ sprawi, że Java stanie się bardziej przystępny dla programistów, którzy zdobyli już doświadczenie z technologiami C i C++. Właśnie z myślą o doświadczonych pracownikach branży IT rozwijany był język Java. Wynika z tego podejścia jednak poważna wada, którą Herbert Schildt podsumował zdaniem: *Java nie jest językiem do nauki programowania.*

Niektóre osoby nieobyte z tematem z racji podobieństwa między językami Java i C++ sądzą, że Java to swego rodzaju interpretacja języka C++. Jest to poważny błąd, o którym świadczy choćby to, że Java w żadnym stopniu nie jest zgodna z językiem C++[1].

Język programowania Java miał zapełnić lukę związaną z pisaniem kodu działającego niezależnie od platformy sprzętowej, który jest udostępniany za pomocą Internetu. Udało się osiągnąć to i wiele więcej, ponieważ rozszerzono i udoskonalono paradygmat obiektowy. Znany jest on z języka C++. Udostępniono również bibliotekę pozwalającą na dostęp do Internetu i wprowadzono zintegrowaną obsługę przetwarzania wielowątkowego[1].

2.3. Python

Prace nad językiem Python rozpoczęły się w latach 90. XX wieku przez holenderskiego programistę Guido van Rossuma w czasie prac w Instytucie badawczym Matematyki i Informatyki CWI znajdującym się w Królestwie Niderlandów. Python miał z założenia zastąpić język programowania znany jako ABC i charakteryzujący się budową utrudniającą stosowanie rozszerzeń. Nazwa Python jest hołdem oddanym brytyjskiej grupie komediowej Monty Python i odzwierciedla filozofię twórcy, który chciał sprawić, aby użytkowanie tego języka dawało programistom radość. Od 1995 roku Guido van Rossum rozwijał projekt języka Python w Korporacji Narodowych Inicjatyw Badawczych CNRI z siedzibą mieszczącą się w Stanach Zjednoczonych Ameryki w mieście Resto w stanie

Wirginia. Zostało tam wydanych kilka wersji oprogramowania. W maju 2000 roku główny zespół programistów języka Python z Guido van Rossumem na czele przenieśli się do BeOpen, a następnie założyli BeOpen PythonLabs. W październiku tego samego roku odbyła się kolejna przeprowadzka, tym razem do Digital Creations (w dniu dzisiejszym Zope Corporation). W roku 2001 powstała organizacja pożytku publicznego, której celem było posiadanie własności intelektualnej związanej z językiem Python nosząca nazwę Python Software Foundation. Fundacja ta jest wspierana materialnie przez Zope Corporation.

Język Python, podobnie jak C++, obsługuje wiele paradygmatów programowania. W pełni obsługiwane są paradygmaty obiektowy i strukturalny. Istnieją jednak również rozwiązania wspierające programowanie funkcyjne i aspektowe. Inne paradygmaty są obsługiwane poprzez rozszerzenie.

Python używa typowania dynamicznego, czyli przypisanie typów danych do wartości przechowywanych w zmiennych zachodzi w czasie działania programu. Dzięki temu programiści są zwolnieni z obowiązku deklarowania typów zmiennych. Wadą tego rozwiązania jest utrudniona kontrola nad integralnością programu. Zmienna w różnych momentach działania programu może przechowywać wartości różnych typów. Innymi językami, w których występuje typowanie dynamiczne są np. MATLAB, R, PHP, Ruby czy JavaScript.

Język Python został zaprojektowany z myślą o zapewnieniu dużej rozszerzalności z wykorzystaniem modułów. Dzięki temu rozwiązaniu język ten zyskał popularność jako sposób na dodawanie programowalnych interfejsów do już istniejących aplikacji. Python był rozwijany z myślą o zapewnieniu prostej składni i gramatyki. Ułatwia to naukę języka i czyni go popularnym wśród osób z małym doświadczeniem w programowaniu.

Jednym z głównych założeń towarzyszących pracom nad językiem Python było uczynienie go czytelnym. Ułatwia to wizualnie przejrzyste formatowanie oraz użycie angielskich słów kluczowych. W przeciwieństwie do wielu innych języków programowania Python nie używa nawiasów klamrowych do oddzielenia bloków kodu. Zamiast tego stosowana jest tabulacja.

Dużych rozmiarów biblioteka standardowa języka Python jest przez wielu programistów wymieniana jako jeden z jego największych zalet, ponieważ jest to narzędzie dostosowane do wykonywania wielu różnorodnych zadań. Obsługiwanych jest wiele standardowych protokołów i formatów używanych w aplikacjach internetowych, takich jak MIME i HTTP. Biblioteka zawiera też moduły do tworzenia graficznych interfejsów

użytkownika, generowania liczb pseudolosowych, łączenia się z relacyjnymi bazami danych, testowania jednostkowego i manipulowania wyrażeniami regularnymi[5].

3. Środowiska programistyczne

W czasie opracowywania pracy inżynierskiej niezwykle ważnym aspektem był wybór odpowiedniego środowiska programistycznego. Nie zapadła decyzja na korzystanie z jednego edytora, który później za pomocą różnorodnych rozszerzeń i dodatków zostałby dostosowany do pisania kodu w trzech wybranych wcześniej językach programowania. Taką możliwość daje na przykład Visual Studio Code[6]. Zamiast tego dla każdej z technologii wybrano osobny edytor, który zawiera wszystko, czego programista może potrzebować przy pracy nad projektami wykonywanymi w danym języku programowania. Wybór padł na Visual Studio dla C++, IntelliJ IDEA dla Javy i PyCharm dla Pythona.

3.1. Visual Studio

Visual Studio to zintegrowane środowisko programistyczne (ang. IDE), które zostało opracowane i wprowadzone na rynek przez firmę Microsoft. Jego zadanie to tworzenie programów komputerowych, takich jak aplikacje, usługi i strony internetowe, czy aplikacje mobilne. Visual studio wykorzystuje platformy programistyczne, które zostały wykonane przez firmę Microsoft, takie jak Windows API, Windows Presentation Foundation, Windows Store, Windows Forms czy Microsoft Silverlight.

Edytor kodu znajdujący się w Visual Studio obsługuje technologię IntelliSense oraz zapewnia refaktoryzację kodu i zintegrowany debugger. Do innych wbudowanych narzędzi można zaliczyć na przykład projektant do tworzenia aplikacji GUI, projektant stron internetowych, projektant schematów baz danych czy profiler kodu. Visual Studio obsługuje również wtyczki, przykładowo dające możliwość obsługi systemu kontroli wersji.

Visual Studio obsługuje 36 języków programowania, jednak nie wszystkie są wbudowane. Bez instalacji różnego rodzaju wtyczek i rozszerzeń w ramach tego IDE mamy możliwość pisania programów w językach C, C++, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, JavaScript, TypeScript, HTML i CSS. Dodatki mogą zapewnić odpowiednie środowisko do pracy na przykład w Pythonie, Ruby czy Node.js. Nie jest obsługiwany język Java, chociaż taka możliwość istniała w przeszłości.

Dostępne są trzy wersje Visual Studio: Community, Professional i Enterprise. Edycja Community jest dedykowana użytkownikom prywatnym oraz celom edukacyjnym. Wersja Professional ma te same funkcjonalności co Community, tyle że przeznaczona jest do użytku komercyjnego. Edycja Enterprise wprowadza wiele dodatkowych opcji w zakresie

zintegrowanego środowiska programistycznego (np. diagramy warstw architektonicznych, walidacja architektury), debugowania i diagnostyki (np. IntelliTrace, analiza kopii zapasowej pamięci platformy .NET) i narzędzi do testowania (np. Live Unit Testing, IntelliTest, Microsoft Fakes). Czyni to z wersji Enterprise najbardziej rozbudowaną pod względem zawartości edycję Visual Studio[7].

3.2. IntelliJ IDEA

IntelliJ IDEA to zintegrowane środowisko programistyczne (ang. IDE) napisane w języku Java. Służy ono do tworzenia oprogramowania komputerowego w językach Java, Kotlin, Groovy i innych opartych na JVM, czyli wirtualnej maszynie Javy. Środowisko to rozwijane jest przez firmę JetBrains (wcześniej znaną jako IntelliJ).

Pierwsza wersja IntelliJ IDEA została udostępniona na początku 2001 roku. Wówczas było to jedno z niewielu dostępnych IDE dla języka Java ze zintegrowanymi zaawansowanymi funkcjami nawigacji po kodzie i jego refaktoryzacji. W 2009 roku firma JetBrains udostępniła kod źródłowy IntelliJ IDEA w ramach licencji Apache 2.0 typu open source. JetBrains rozpoczęło również dystrybucję ograniczonej darmowej wersji IDE pod nazwą Community Edition. Komercyjna edycja Ultimate posiada dodatkowe funkcjonalności i jest dostępna za opłatą.

IntelliJ IDEA zapewnia funkcje takie jak uzupełnianie napisanego kodu poprzez analizę kontekstową, nawigację po kodzie umożliwiającą bezpośrednie przeskoczenie do klasy lub deklaracji w kodzie, refaktoryzację i debugowanie kodu oraz opcje poprawiania niespójności za pomocą sugestii.

Z IntelliJ IDEA zostały zintegrowane narzędzia do budowania kodu, takie jak Grunt lub Gradle. Obsługiwane są również różne systemy kontroli wersji. Nie tylko popularny Git, ale także Mercurial, Subversion czy Perforce. IDE w wersji Ultimate zapewnia również dostęp do baz danych takich jak Microsoft SQL Server, PostgreSQL, SQLite, Oracle i MySQL.

IntelliJ IDEA obsługuje wtyczki i dodatki, dzięki którym możliwe jest dodawanie niewystępujących w podstawowej wersji IDE funkcjonalności. Wtyczki można pobrać i zainstalować ze strony internetowej zawierającej repozytorium ponad 3000 dodatków dla IntelliJ IDEA w wersji Community i Ultimate (dla każdej edycji dostępne są inne wtyczki) lub bezpośrednio z IDE, za pomocą wbudowanych funkcji wyszukiwania i instalacji wtyczek[8].

3.3. PyCharm

PyCharm to zintegrowane środowisko programistyczne (ang. IDE) wykorzystywane do programowania w języku Python. Zapewnia między innymi debugger graficzny, analizę kodu, zintegrowane środowisko do testów jednostkowych, integrację z systemami kontroli wersji (Mercurial, Git, Subversion, Perforce i CVS) oraz wsparcie dla tworzenia stron internetowych przy wykorzystaniu Django. Rozwijaniem PyCharm zajmuje się firma JetBrains.

Jest to wieloplatformowe IDE i działa na systemach Microsoft Windows, macOS i Linux. Dostępna jest wersja Community wydana na licencji Apache oraz bardziej rozbudowana wersja Professional wydana na licencji zastrzeżonej.

PyCharm pojawił się na rynku zintegrowanych środowisk programistycznych w wersji 1.0 w 2010 roku jako bezpośrednia konkurencja dla PyDev - IDE języka Python funkcjonującego w ramach Eclipse oraz Komodo opracowywanego przez firmę ActiveState. Kolejne wersje były udostępniane w 2011 (2.0), 2013 (3.0) i 2014 roku (4.0). Od roku 2013 PyCharm zaczął być udostępniany pod postacią Open Source.

Środowisko to zapewnia wiele pomocnych funkcji w programowaniu w języku Python. Należą do nich na przykład uzupełnianie kodu na podstawie analizy kontekstu oraz podświetlanie błędów składni. Możliwa jest również nawigacja po kodzie przy wykorzystaniu specjalistycznego widoku projektu i struktury plików oraz szybkiego przemieszczania się pomiędzy klasami, plikami i metodami. Szeroko wspierane są również funkcje umożliwiające refaktoryzację kodu. Zintegrowane zostały również narzędzia naukowe, takie jak Matplotlib i NumPy[9].

4. Algorytmy

Kluczowym zagadnieniem związanym z pisaniem pracy inżynierskiej był wybór odpowiednich algorytmów, na przykładzie których testowany w przyszłości zostanie kod napisany w językach C++, Java i Python. Zapadła decyzja na użycie kilku algorytmów o różnym stopniu skomplikowania: od używanych do nauki podstaw pisania w językach programowania do skomplikowanych algorytmów szyfrowania. Proste algorytmy zostały zawarte w pracy, ponieważ zwykle cechują się one wykładniczą złożonością obliczeniową dającą możliwość najlepszej obserwacji różnic w wydajności.

4.1. Wyszukiwanie n-tej liczby ciągu Fibonacciego

Ciąg Fibonacciego to ciąg liczb naturalnych. Określony jest on rekurencyjnie w taki sposób, że pierwszy wyraz to 0, drugi to 1, a każdy następny to suma dwóch poprzednich. Ciąg ten został omówiony w 1202 roku przez włoskiego matematyka Leonarda z Pizy, znanego szerzej jako Fibonacci[16].

Algorytm wyszukiwania n-tej liczby ciągu to doskonały algorytm do testów wydajności, ponieważ bardzo szybko zwiększają się liczby, na których dokonywane są operacje. Można go zapisać w wersji iteracyjnej lub rekurencyjnej. Padła decyzja nad zawarciem w pracy zarówno jednego, jak i drugiego rozwiązania, aby dodatkowo zaistniała możliwość porównania wydajności różnych wersji algorytmu.

```cpp
#include <iostream>
#include <chrono>

uint32_t fibonacci(uint8_t n) {
    if (n <= 1) {
        return n;
    }
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}

int main() {
    uint8_t n = 30;
    uint64_t start = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch()).count();
    uint32_t result = fibonacci(n);
    uint64_t stop = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch()).count();
    std::cout << "Liczba ciagu Fibonacciego o numerze 30 to: " << result << std::endl;
    std::cout << "Czas wykonania programu to " << stop - start << " ms.";
    return 0;
}
```

Rysunek 4.1. Kod algorytmu w wersji rekurencyjnej napisany w języku C++. Użyte zostały funkcje *duration_cast* w celu obliczenia czasu wykonania programu [opracowanie własne]

4.2. Algorytm sortowania szybkiego

Algorytm sortowania szybkiego (ang. quicksort) to popularny algorytm sortowania o stosunkowo wysokiej wydajności rzędu *O(n log n)*.

Działanie algorytmu można przedstawić w następujący sposób: z tablicy danych do posortowania wybiera się tak zwany element rozdzielający (ang. pivot). Następnie tablica dzielona jest na dwa mniejsze fragmenty. Do jednego przenoszone są wszystkie elementy mniejsze od elementu rozdzielającego, a do drugiego wszystkie większe. Obydwa fragmenty są następnie sortowane. Cały proces kończy się w momencie, w którym z podziału uzyskiwane są fragmenty z jednym elementem, ponieważ takie nie wymagają sortowania. Takie działanie algorytmu określane jest mianem zasady *dziel i zwyciężaj*[17].

Złożoność obliczeniowa tego algorytmu sprawia, że traci on na wydajności przy małych zbiorach elementów do posortowania. W takich właśnie warunkach zostaną przeprowadzone testy.

4.3. Algorytm Blowfish

Blowfish to szyfr opracowany w 1993 roku przez amerykańskiego kryptografa Bruce'a Schneiera. Jest to szyfr z kluczem symetrycznym. Zaprojektowany został jako algorytm do przeznaczenia ogólnego jako następca innych starzejących się szyfrów. Blowfish, w przeciwieństwie do innych szyfrów powstających w tamtym czasie, nie został opatentowany. Autor chciał, żeby każdy mógł z niego skorzystać.

Blowfish korzysta z bloku o rozmiarze 64 bitów. Klucz ma zmienny rozmiar: od 32 do 448 bitów.

Żeby algorytm działał poprawnie, musi on zostać zainicjalizowany kluczem. Dobrą praktyką jest przechowywanie tego klucza w postaci zaszyfrowanej.

Blowfish korzysta z pięciu tablic podkluczy: tablica P zawierająca 18 elementów i cztery tablice S zawierające po 256 elementów. Są one niezbędne do poprawnego działania obecnej w algorytmie funkcji Feistela.[18]

```
uint32_t P[18] = {
    0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,
    0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,
    0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,
    0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917,
    0x9216d5d9, 0x8979fb1b
};

uint32_t S[4][256] = {
    { 0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7 /* ... */ },
    { 0x76dc4190, 0x98d220bc, 0xefa8c1d7, 0xf61e2562 /* ... */ },
    { 0x76dc4190, 0x98d220bc, 0xefa8c1d7, 0xf61e2562 /* ... */ },
    { 0x76dc4190, 0x98d220bc, 0xefa8c1d7, 0xf61e2562 /* ... */ }
};
```

Rysunek 4.2. Przykład inicjalizacji tablicy P i czterech tablic S napisany w języku C++ [opracowanie własne]

Podstawą szyfrowania są cztery następujące akcje:

1. Operacja XOR na lewej połowie danych wejściowych i wartością kolejnej komórki w tablicy P,
2. Funkcja Feistela używająca danych wyjściowych z punktu pierwszego jako argumentów,
3. Operacja XOR na danych wyjściowych funkcji Feistela i prawej połowie danych wejściowych,
4. Zamiana lewej i prawej strony danych wejściowych miejscami.

Powtarza się te czynności szesnaście razy. Na koniec wykonuje się operację XOR na lewej połowie danych wejściowych i wartości osiemnastej komórki tablicy P oraz prawej połowie danych wejściowych i wartości siedemnastej komórki tablicy P. Odszyfrowywanie odbywa się w identyczny sposób z tą różnicą, że wartości tablicy P są wykorzystywane w odwrotnej kolejności.[18]

4.4. Algorytm kompresji RLE

RLE, czyli Run-Length Encoding (kodowanie długości serii) jest metodą bezstratnej kompresji danych. W tej metodzie sekwencje danych, w których jedna wartość się powtarza, przechowywane są jako pojedyncza wartość danych oraz liczby kolejnych jej wystąpień. Najlepiej sprawdza się to w przypadku kompresji formatów, które zawierają wiele takich

sekwencji, na przykład mapy bitowe. W przypadku plików, w których nie ma wielu takich sekwencji RLE może wywołać efekt przeciwny, czyli zwiększenie rozmiaru pliku.

Na przykład następujący ciąg wartości: *ppppwwwssz* po poddaniu kompresji algorytmem korzystającym z metody RLE wyglądać będzie w następujący sposób: *p4w3s2z1*. Widać to dobrze, jak bardzo krótkie sekwencje negatywnie wpływają na efekt końcowy.

Kodowanie długości serii było stosowane od 1967 roku w transmisji analogowych sygnałów telewizyjnych. Stało się też dość popularną metodą kompresji obrazów w usługach internetowych.[19]

4.5. Algorytm kompresji LZW

LZW to metoda bezstratnej kompresji danych opublikowana w 1984 roku przez Abrahama Lempel, Jacoba Ziv i Terry'ego Welch. Od nazwisk tych inżynierów metoda wzięła swoją nazwę. Wykorzystywana jest do kompresji formatów takich jak GIF, czy PDF. Przez pewien czas była opatentowana, co przyczyniło się do Powstania formatu PNG.

Pojedynczy krok algorytmu kompresji polega na wyszukaniu w słowniku najdłuższego prefiksu danych, które nie zostały jeszcze zakodowane. Jako dane wyjściowe wypisuje się kod, który jest związany z tym słowem. Potem do słownika dodaje się nową pozycję. Jest to konkatenacja słowa oraz pierwszej litery, która nie jest dopasowana.

Dekompresja jest bardziej skomplikowanym procesem. Dzieje się tak, ponieważ konieczne jest wykrycie przypadków ciągów *scscs* nie znajdujących się w słowniku. Cig *sc* znajduje się w słowniku, natomiast ciąg *c* jest dowolny i może być nawet pusty.

5. Specyfikacja sprzętowa

Ważnym aspektem jest wymienienie specyfikacji komputera, na którym zostały wykonane testy. Był to laptop Acer Aspire 5 z następującymi podzespołami:

- Procesor: 8-rdzeniowy Intel^{R} Core^{TM} i5-10210U CPU, 1.60GHz, 2.11 GHz
- Pamięć RAM: 16 GB
- Typ systemu: 64-bitowy system operacyjny, procesor x64

Pozostałe podzespoły zostały pominięte z uwagi na brak ich wpływu na przebieg badań.

6. Wyniki badań

Badania nad algorytmami zostały przeprowadzone w czasie, kiedy jedynymi programami działającymi na komputerze były IDE, menedżer zadań i monitor zasobów. Każdy algorytm został uruchomiony dziesięć razy, aby umożliwić uśrednienia wyników. Jak później się okazało był to zabieg zbędny, ponieważ różnica między wynikami dla każdego uruchomienia algorytmu były pomijalnie małe.

Wyniki badań prezentowane są na dwóch zrzutach ekranu przedstawiających monitor zasobów oraz menedżer zadań z momentu wykonywania programu. Dane o zużyciu zawarte w podsumowaniach mówią o szczytowych momentach z przeprowadzania testów.

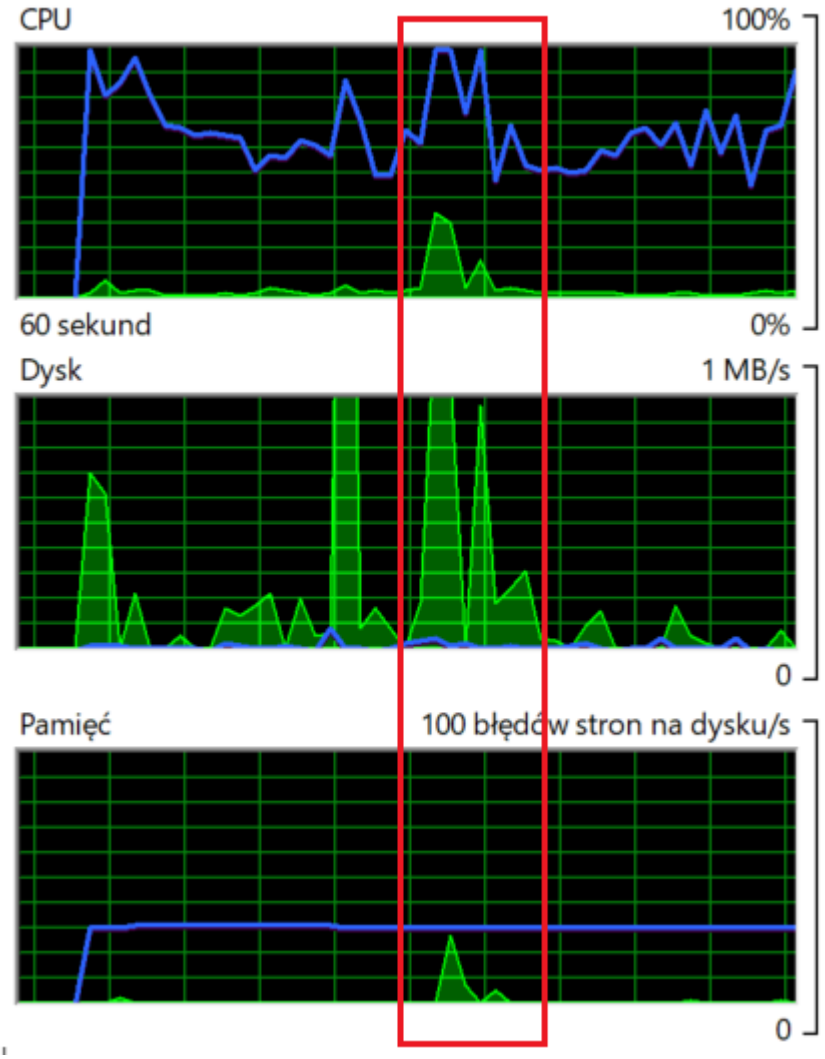
Czas przedstawiany w wynikach badań został zaokrąglony do milisekund.

6.1. Wyniki dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej

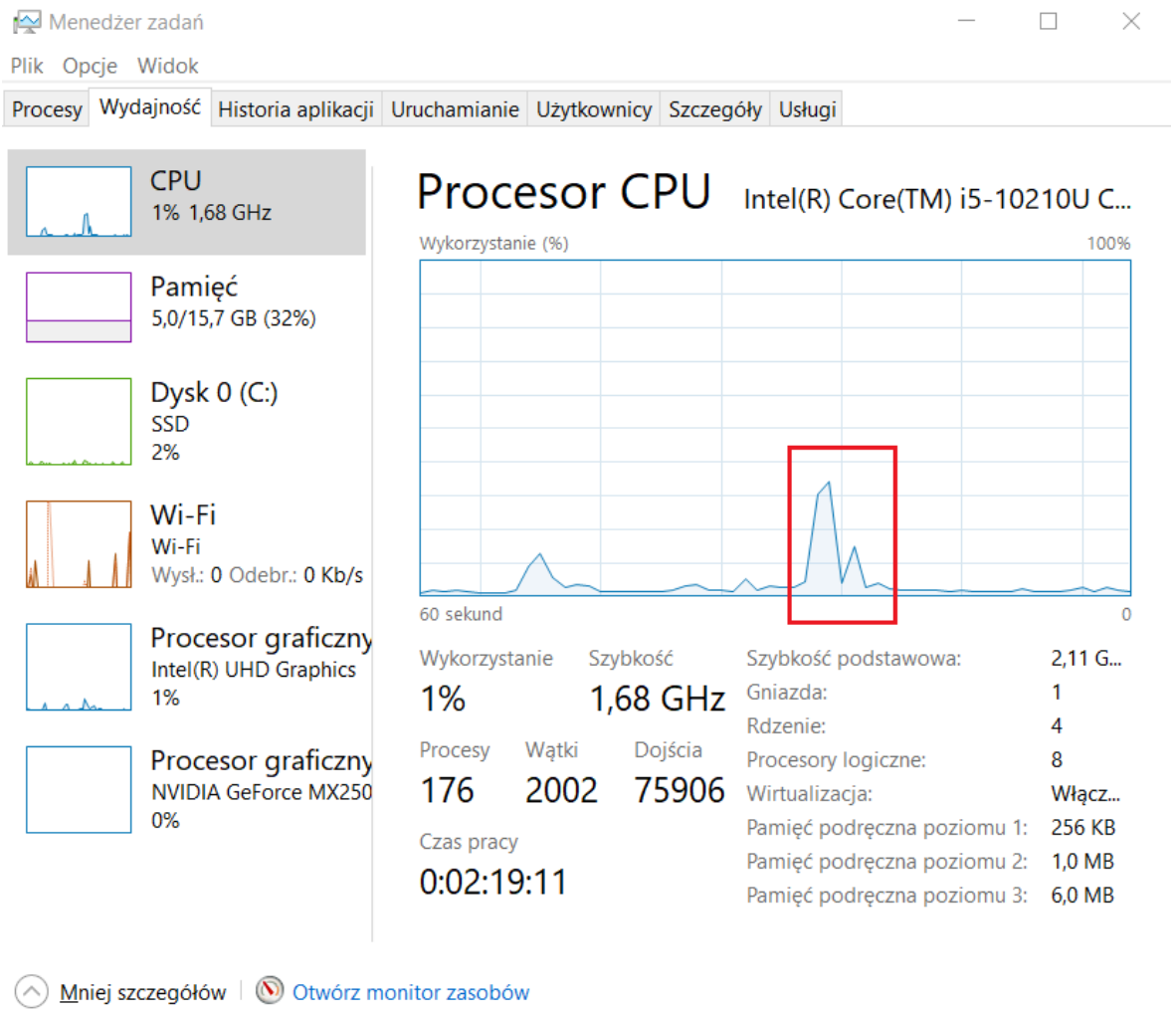
Przedstawione wyniki pochodzą z testów algorytmu wyszukującego trzydziestą liczbę w ciągu Fibonacciego.

6.1.1 C++

Algorytm napisany w języku C++ zużywa sporo zasobów komputera, ale rekompensuje to niski czas wykonania programu.



Rysunek 6.1 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

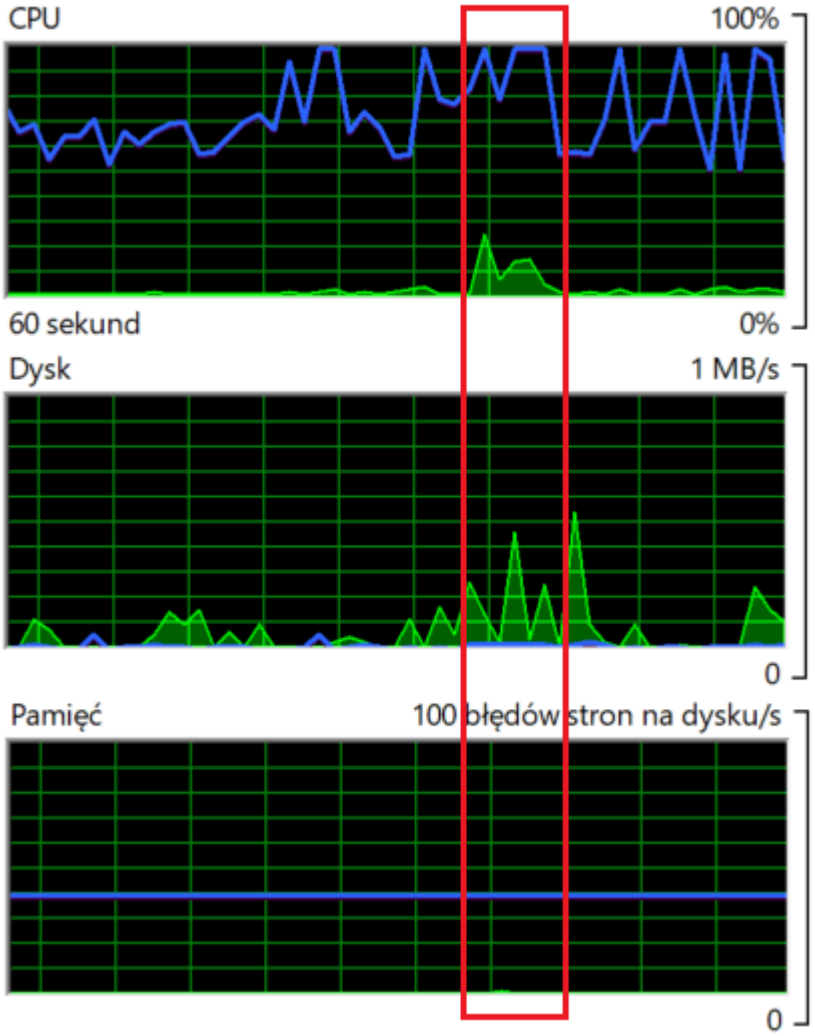


Rysunek 6.2 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

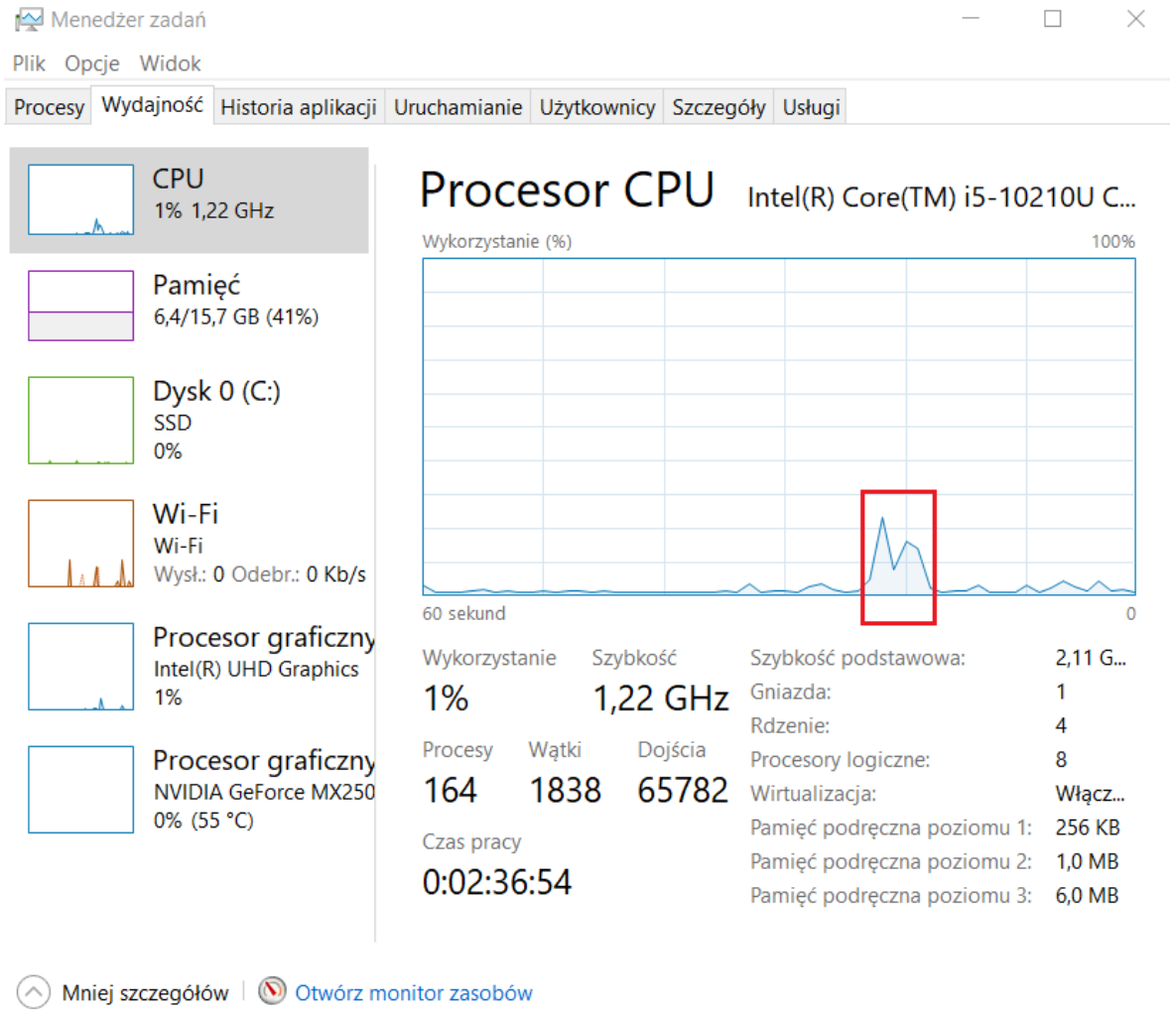
Średni czas wykonania programu to 32 ms. Zużycie procesora wahało się między 30 a 40%. Zużycie dysku wahało się między ok. 0,9 a 1 MB/s. Zużycie pamięci wahało się pomiędzy 20 a 30 błędów stron na dysku/s.

6.1.2 Python

Algorytm napisany w języku Python nie zużywa dużych ilości zasobów komputera, ale czas wykonania programu jest wielokrotnie większy od algorytmów napisanych w innych językach.



Rysunek 6.3 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

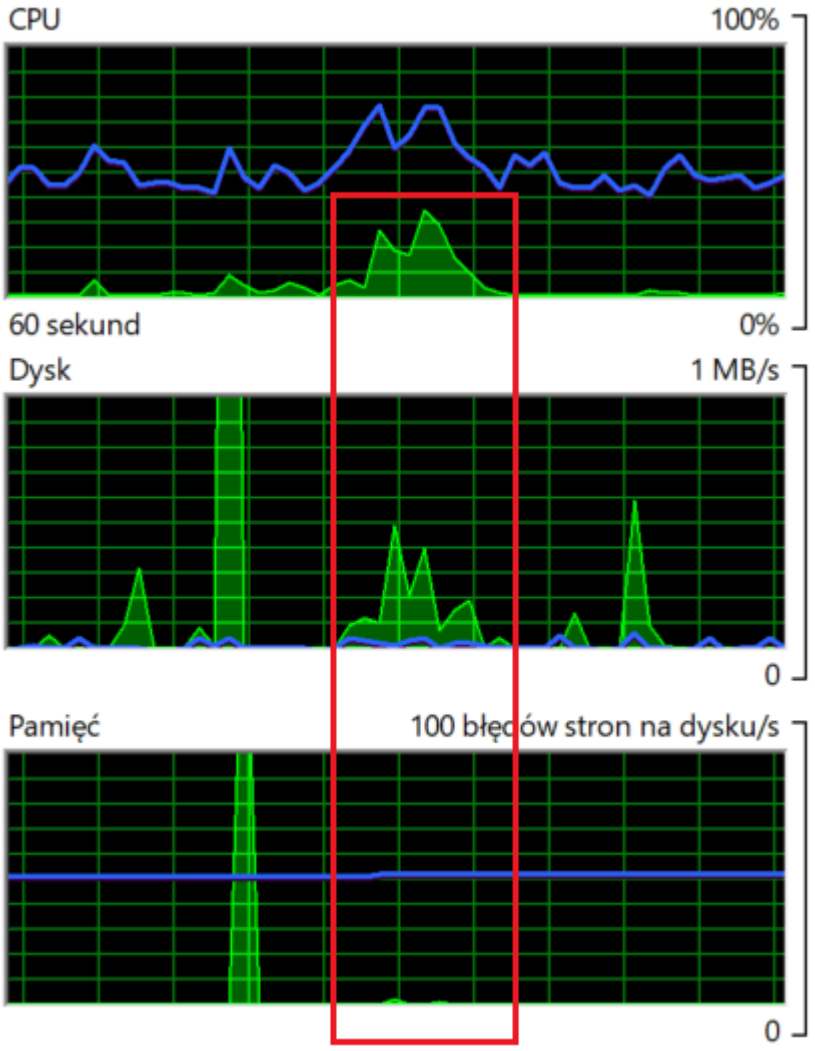


Rysunek 6.4 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

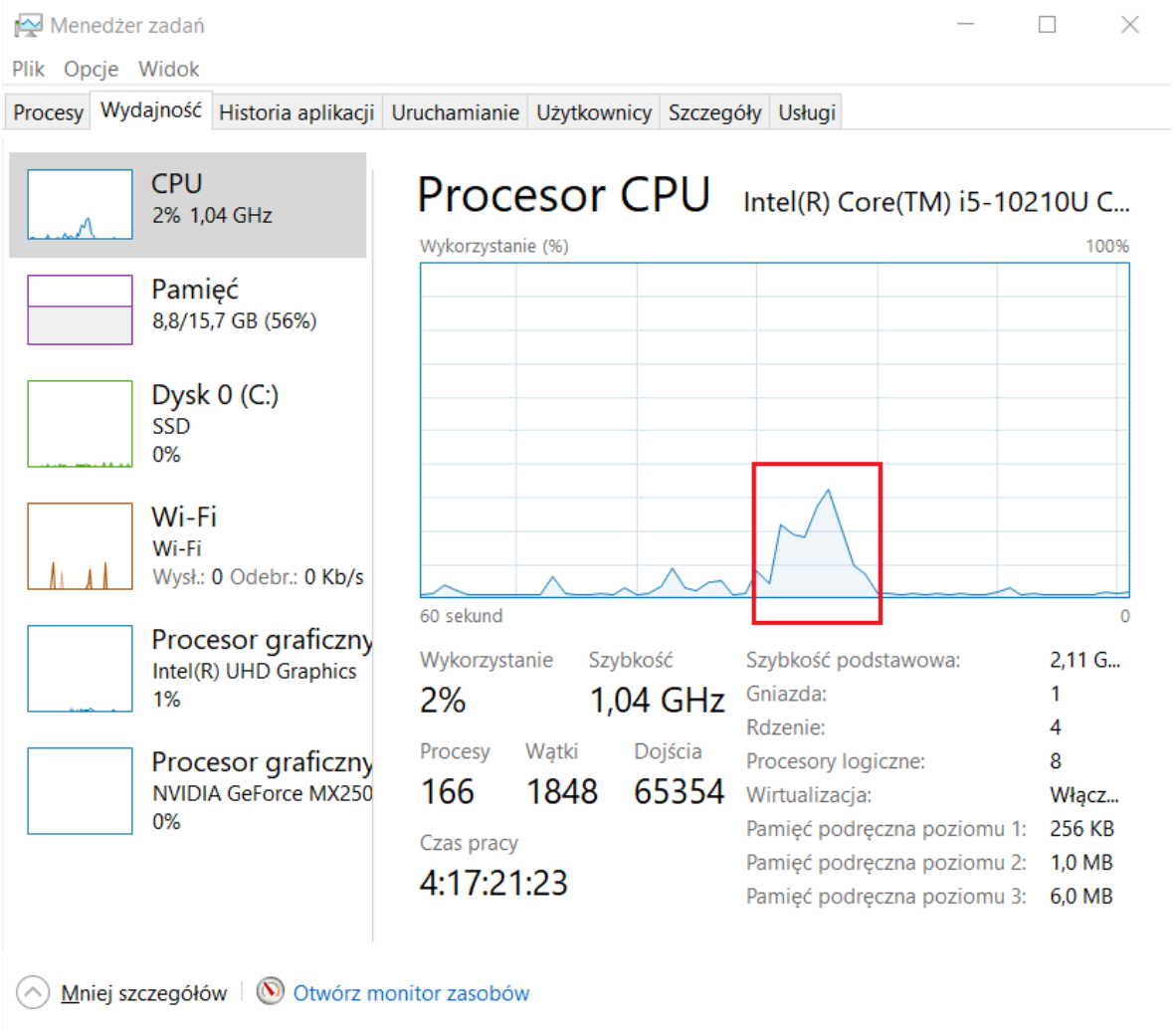
Średni czas wykonania programu to 214 ms. Zużycie procesora wahało się między 20 a 30%. Zużycie dysku wahało się między ok. 0,4 a 0,5 MB/s. Zużycie pamięci wynosiło do 10 błędów stron na dysku/s.

6.1.3 Java

Algorytm napisany w języku Java zużywa mniej zasobów komputera niż ten napisany w C++, a jednocześnie czas wykonania programu jest krótszy.



Rysunek 6.5 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]



Rysunek 6.6 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Średni czas wykonania programu to 8 ms. Zużycie procesora wahało się między 30 a 40%. Zużycie dysku wahało się między ok. 0,4 a 0,5 MB/s. Zużycie pamięci wynosiło do 10 błędów stron na dysku/s.

W tabelach 6.1 i 6.2 przedstawiono czas wykonania programu wyrażony w milisekundach dla każdej implementacji algorytmu w czasie dziesięciu przeprowadzonych badań.

	1	2	3	4	5
C++	32 ms	31 ms	29 ms	33 ms	35 ms
Python	210 ms	218 ms	216 ms	220 ms	215 ms
Java	10 ms	8 ms	7 ms	8 ms	7 ms

Tabela 6.1 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej (testy od 1 do 5) [opracowanie własne]

	6	7	8	9	10
C++	32 ms	33 ms	29 ms	30 ms	36 ms
Python	214 ms	212 ms	211 ms	213 ms	211 ms
Java	9 ms	11 ms	7 ms	6 ms	7 ms

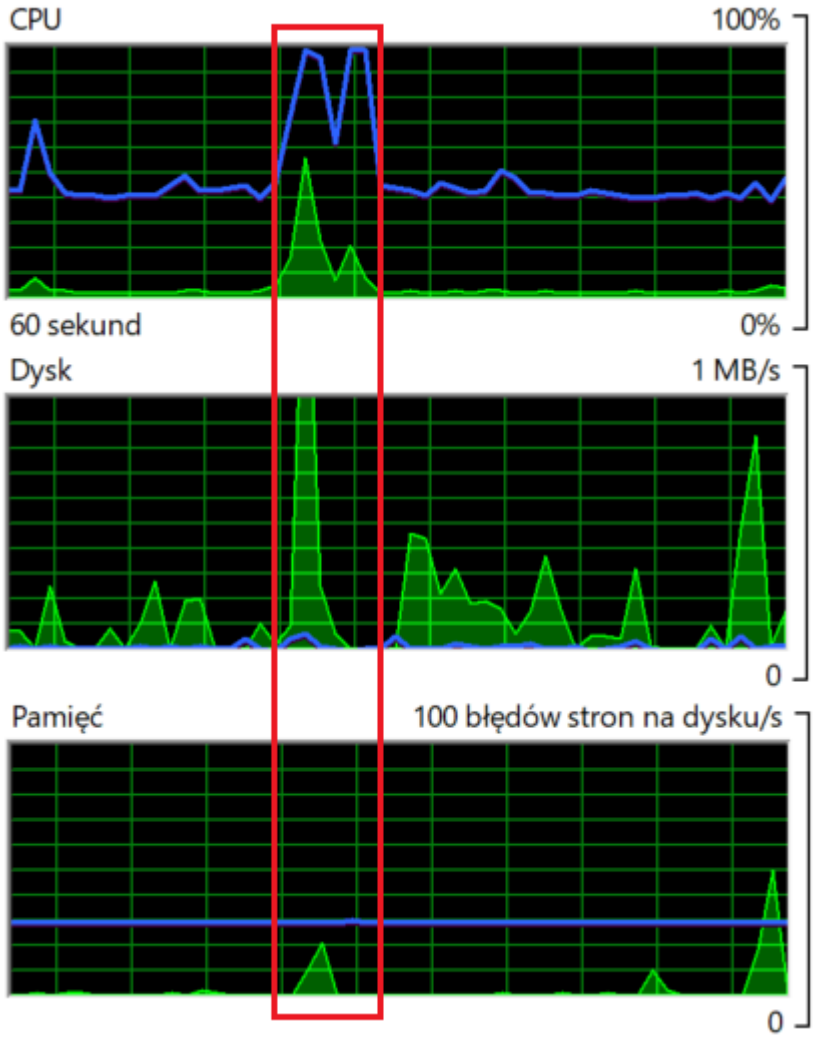
Tabela 6.2 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej (testy od 6 do 10) [opracowanie własne]

6.2. Wyniki dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej

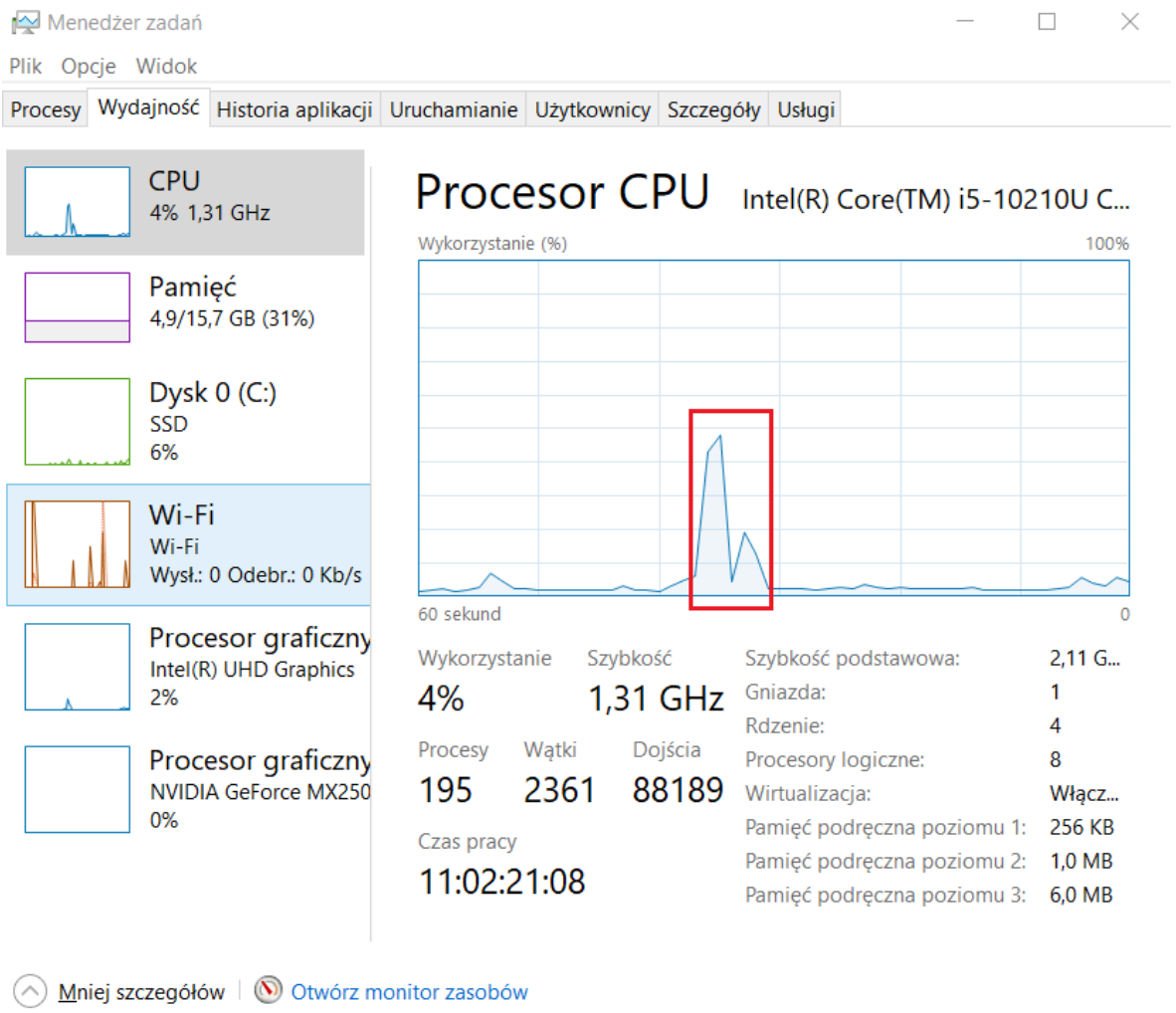
Przedstawione wyniki pochodzą z testów algorytmu wyszukującego 100000 liczbę w ciągu Fibonacciego.

6.2.1 C++

Algorytm napisany w języku C++ charakteryzuje się dużym zużyciem zasobów komputera, jakim jest dysk. Czas wykonania programu jest nieporównywalnie niski.



Rysunek 6.7 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

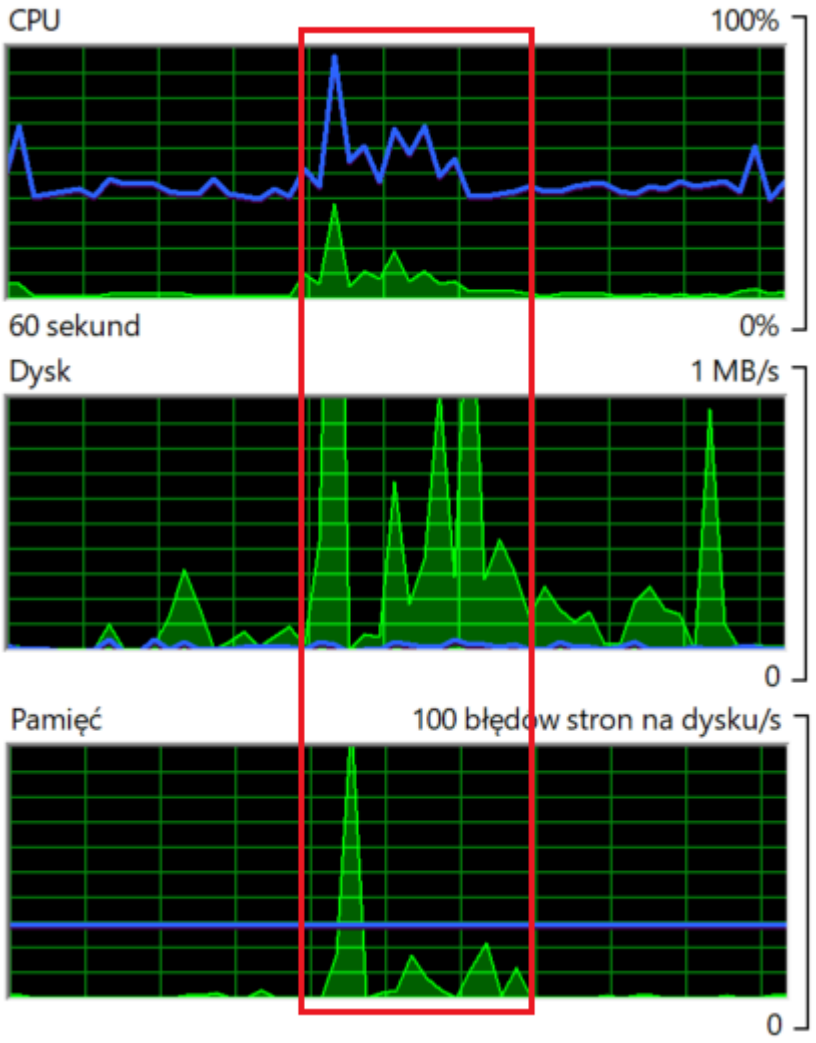


Rysunek 6.8 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

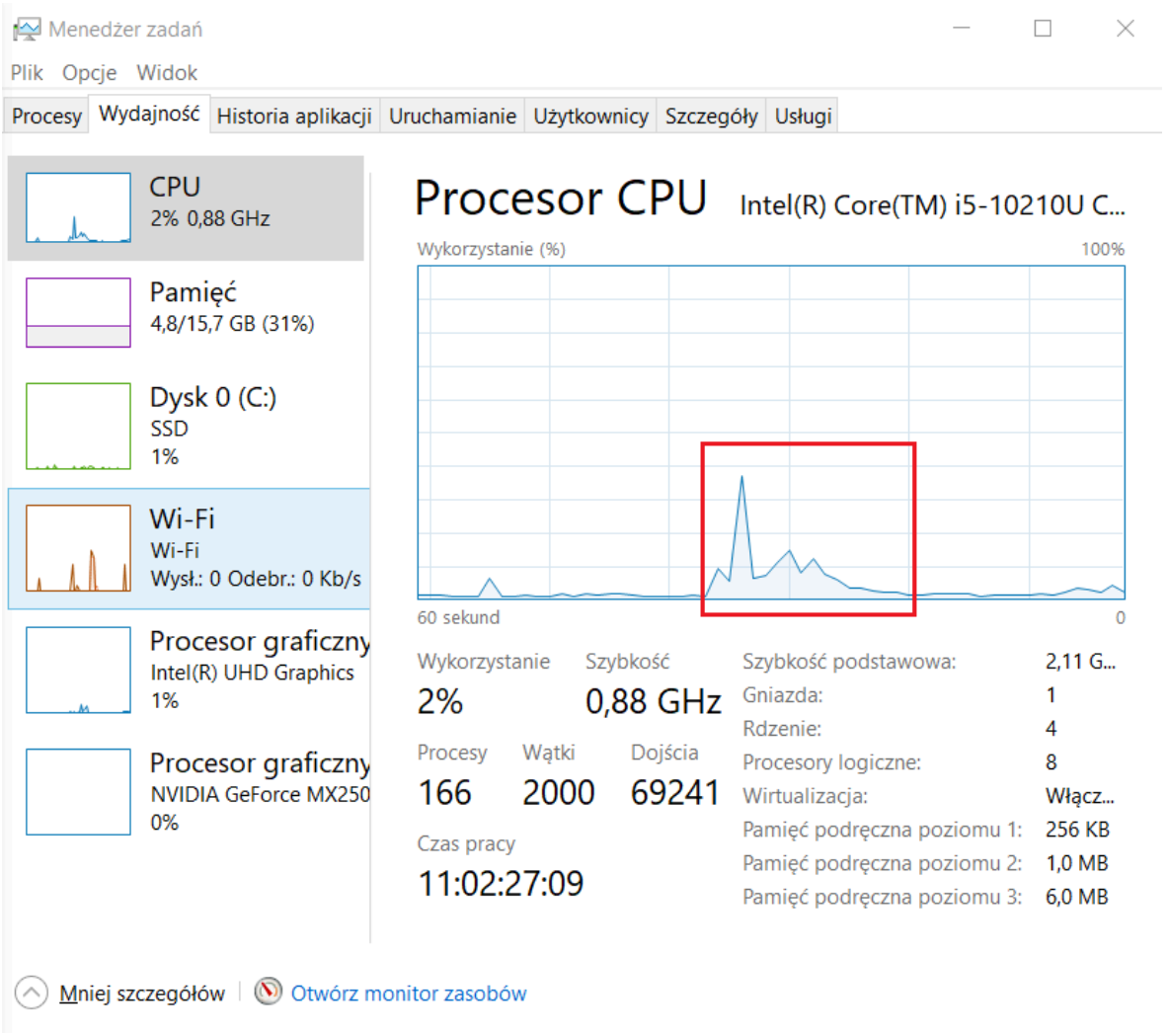
Średni czas wykonania programu to 1 ms. Zużycie procesora wahało się między 50 a 60%. Zużycie dysku wahało się między ok. 0,9 a 1 MB/s. Zużycie pamięci wahało się między 10 a 20 błędów stron na dysku/s.

6.2.2 Python

Algorytm napisany w języku Python charakteryzuje się relatywnie niskim zużyciem zasobów komputera, ale czas wykonania programu jest najgorszy spośród wszystkich implementacji.



Rysunek 6.9 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

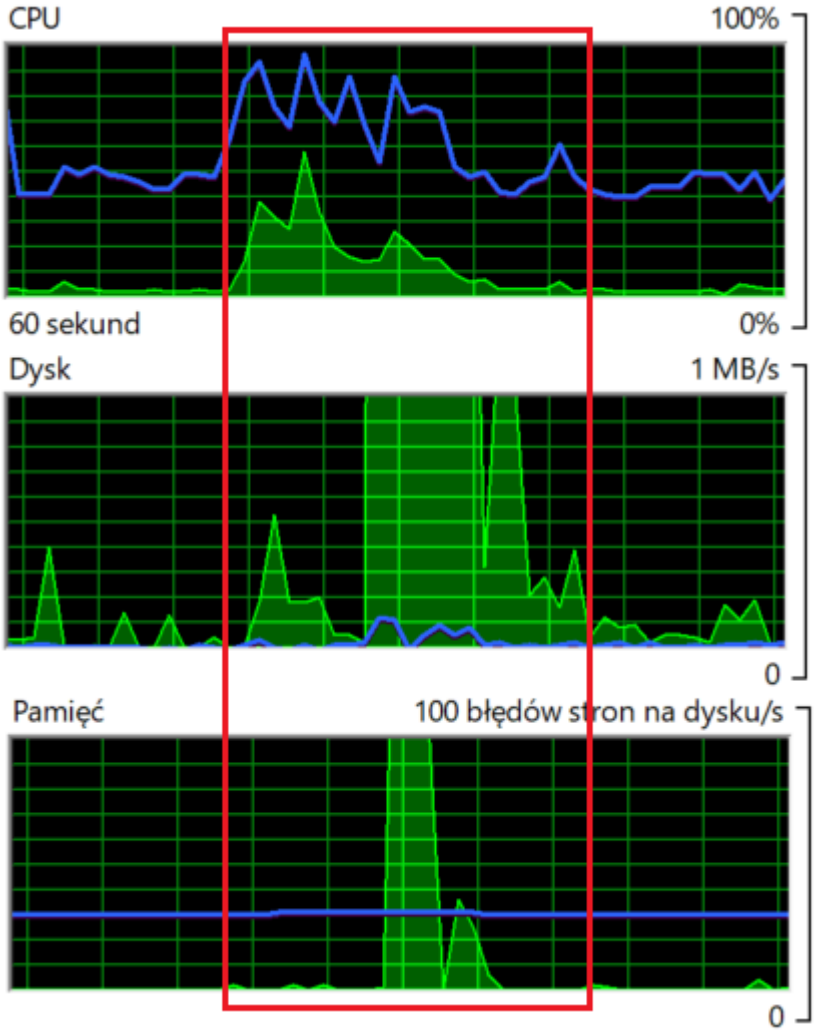


Rysunek 6.10 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

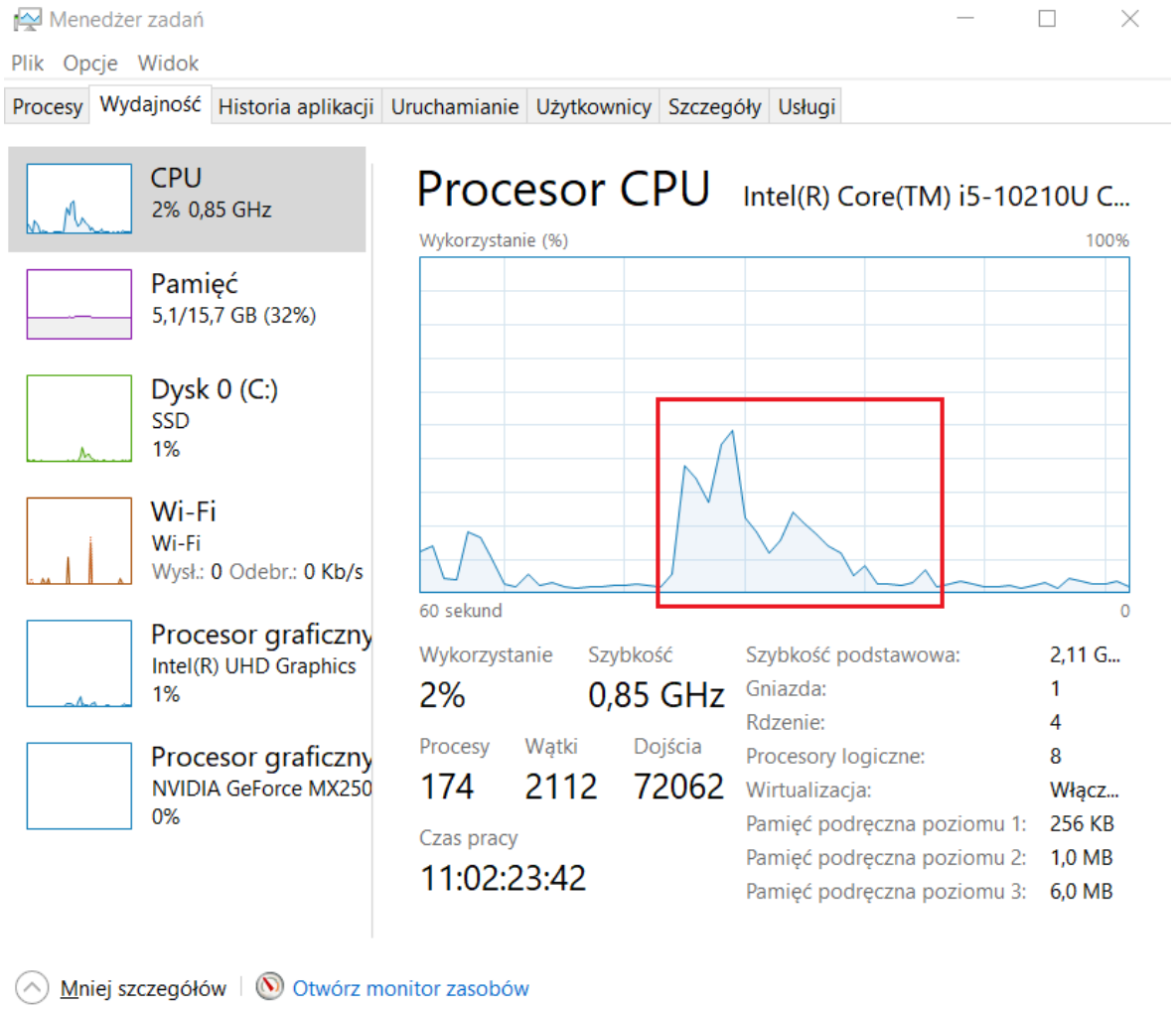
Średni czas wykonania programu to 414 ms. Zużycie procesora wahało się między 30 a 40%. Zużycie dysku wynosiło w szczytowych momentach 1 MB/s. Zużycie pamięci wynosiło między 10 a 20 błędów stron na dysku/s, ale zdarzały się skoki do 100 błędów stron na dysku/s.

6.2.3 Java

Algorytm napisany w języku Java zużywa stosunkowo dużo zasobów dysku i pamięci. Zużycie procesora pozostało na porównywalnym poziomie do innych implementacji algorytmu..



Rysunek 6.11 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]



Rysunek 6.12 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Średni czas wykonania programu to 25 ms. Zużycie procesora wahało się między 50 a 60%. Zużycie dysku przez większość czasu wahało się między ok. 0,9 a 1 MB/s. Zużycie pamięci wahało się między ok. 90 a 100 błędów stron na dysku/s.

W tabelach 6.3 i 6.4 przedstawiono czas wykonania programu wyrażony w milisekundach dla każdej implementacji algorytmu w czasie dziesięciu przeprowadzonych badań.

	1	2	3	4	5
C++	1 ms	1 ms	2 ms	1 ms	1 ms
Python	412 ms	414 ms	411 ms	415 ms	412 ms
Java	27 ms	25 ms	23 ms	24 ms	22 ms

Tabela 6.3 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej (testy od 1 do 5) [opracowanie własne]

	6	7	8	9	10
C++	2 ms	1 ms	1 ms	1 ms	1 ms
Python	414	415	417	416	414
Java	26	25	26	25	27

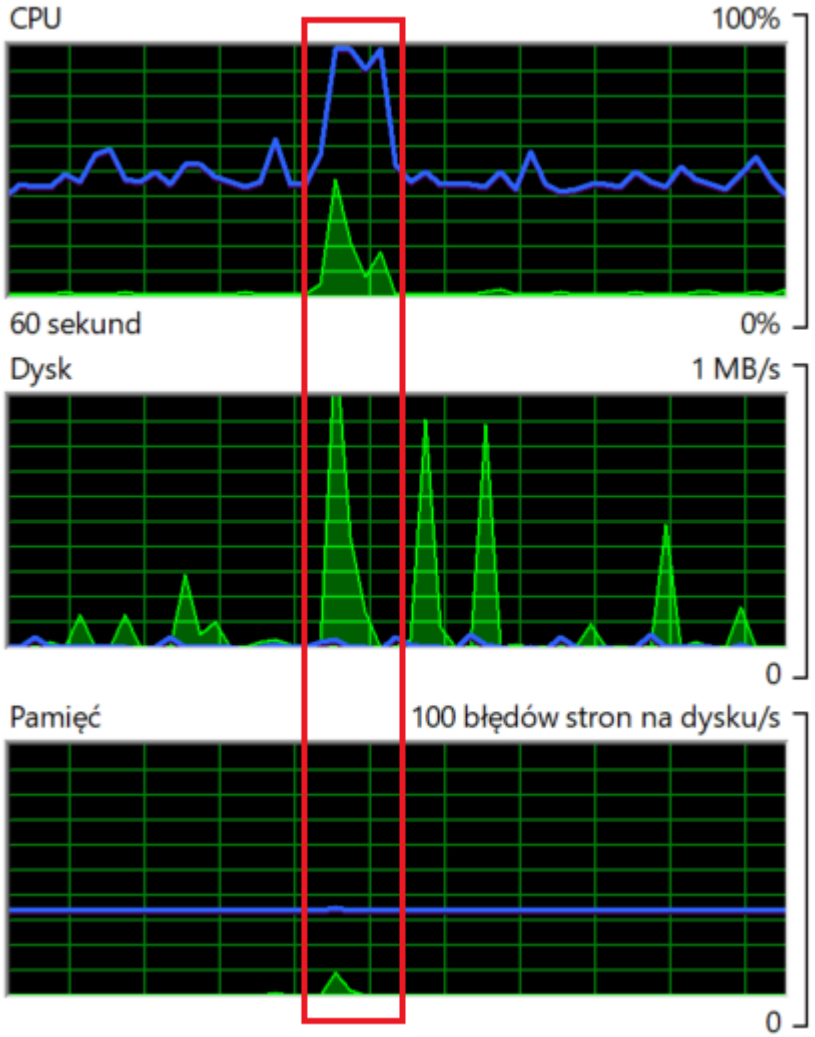
Tabela 6.4 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej (testy od 6 do 10) [opracowanie własne]

6.3. Wyniki dla algorytm sortowania szybkiego

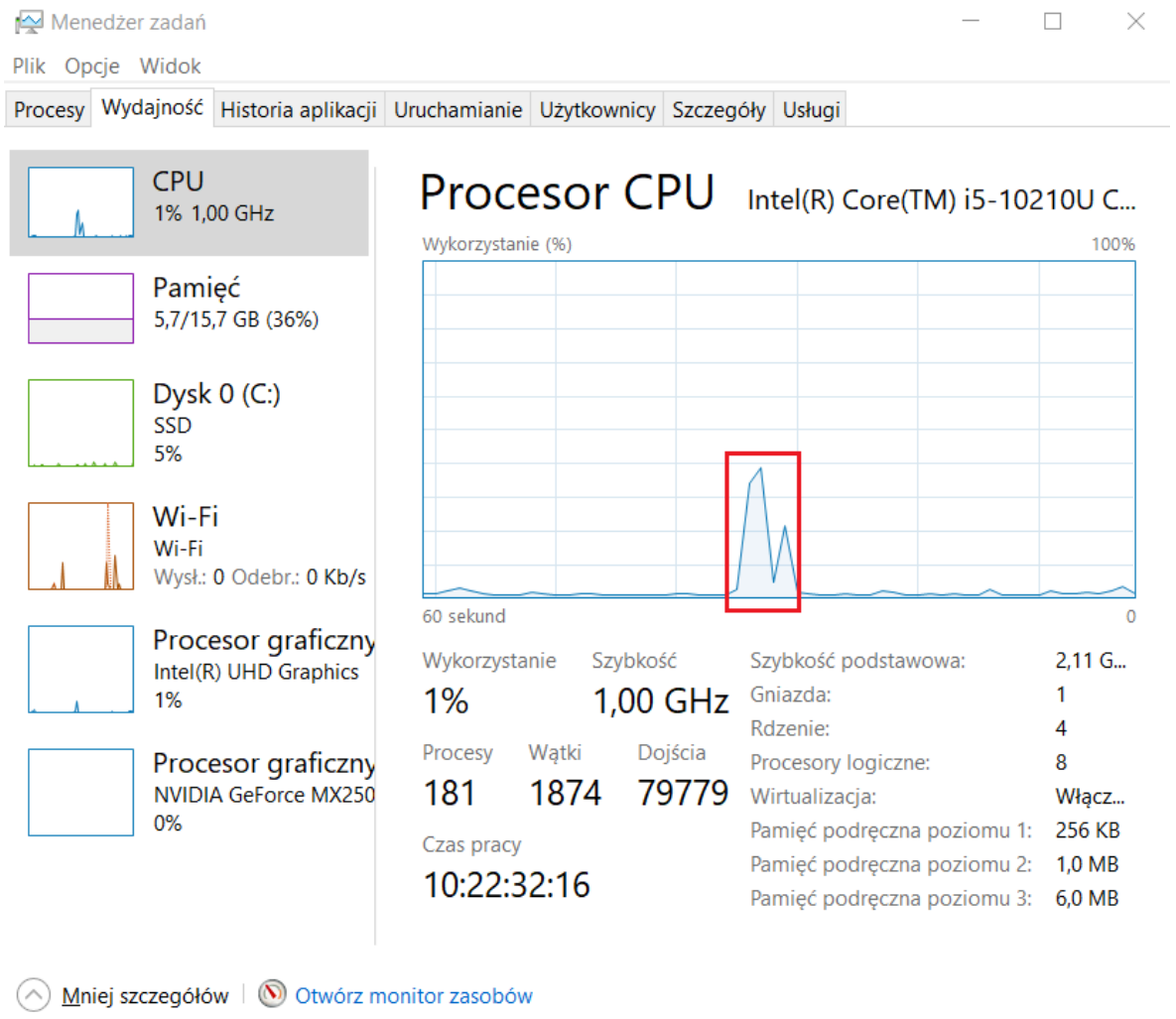
Wszystkie wersje implementacji algorytmu operują na tym samym zbiorze do posortowania. Było to osiem tysięcy losowych liczb z zakresu od 1 do 10000.

6.3.1 C++

Algorytm napisany w języku C++ charakteryzuje się dużym stopniem zużycia dysku sięgającym 1 MB/s.



Rysunek 6.13 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

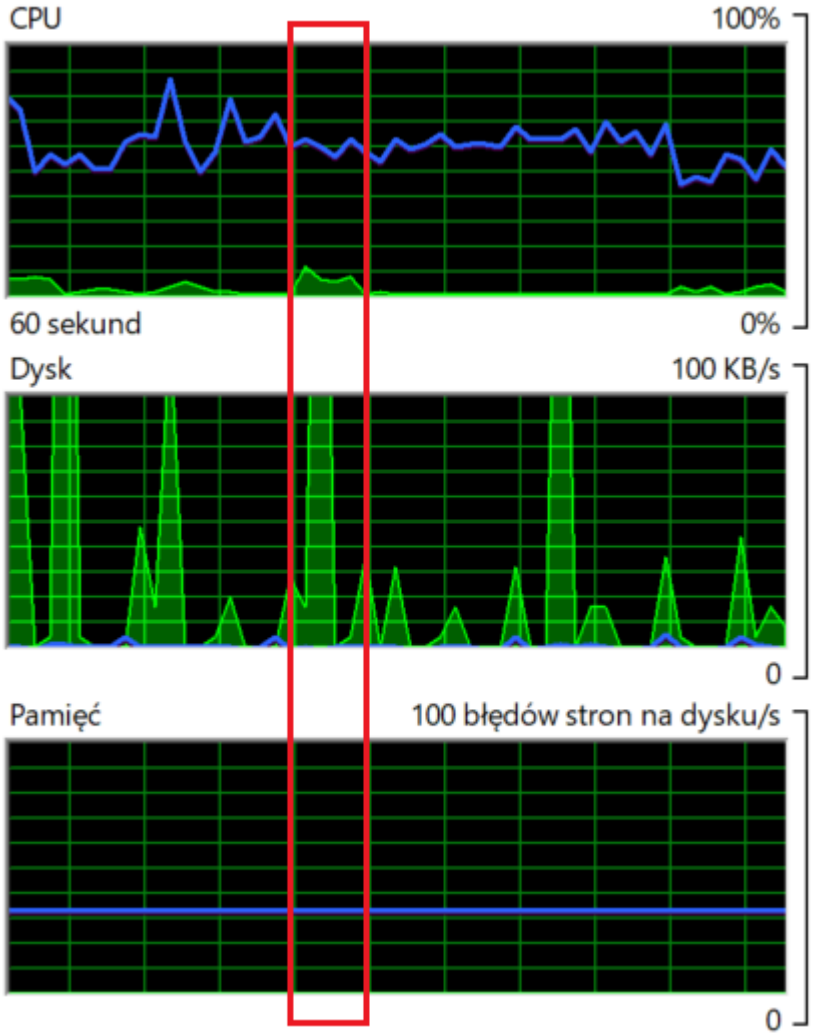


Rysunek 6.14 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

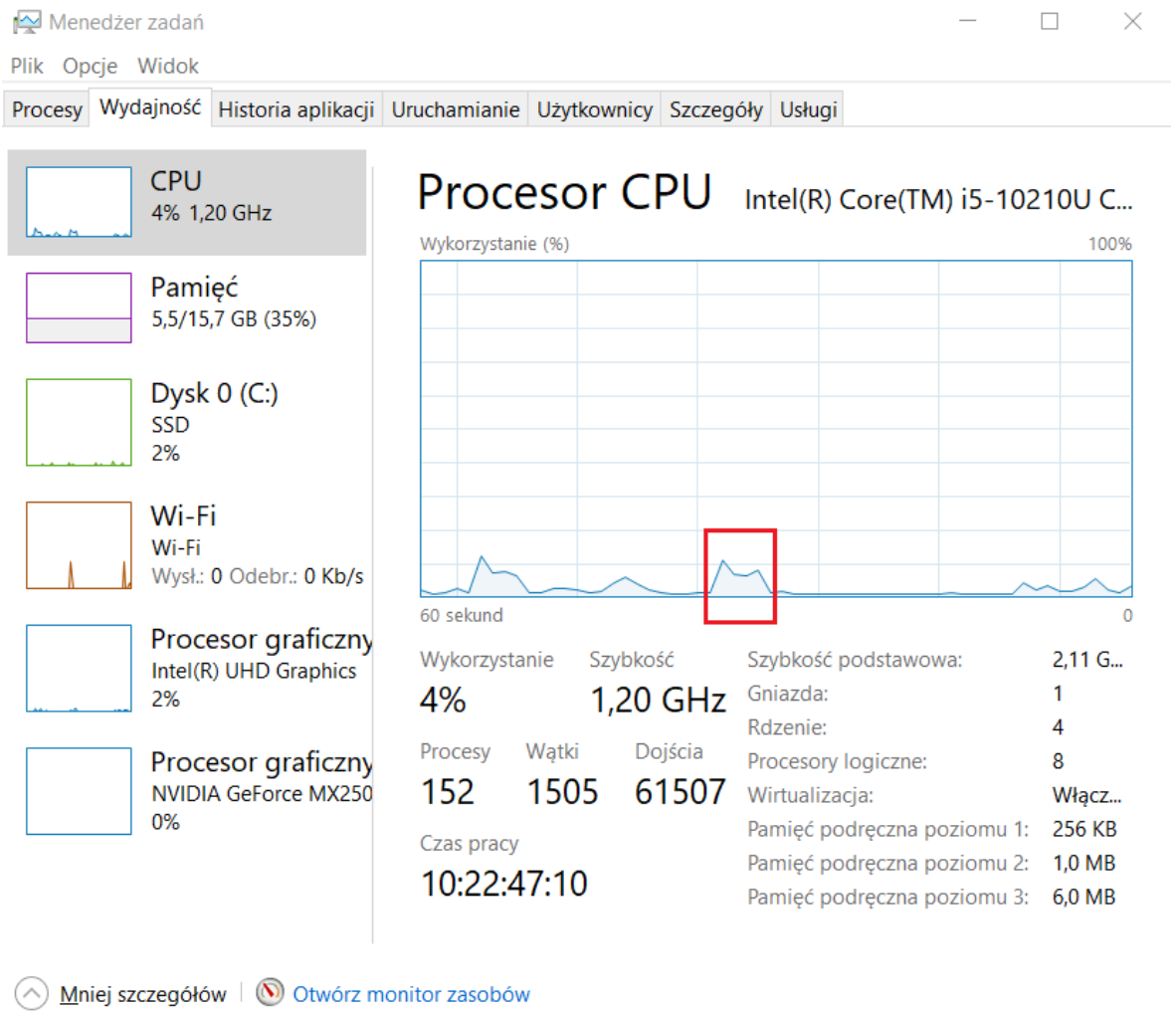
Średni czas wykonania programu to 4 ms. Zużycie procesora wahało się między 50 a 60%. Zużycie dysku wahało się między ok. 0,9 a 1 MB/s. Zużycie pamięci wahało się między ok. 10 a 20 błędów stron na dysku/s.

6.3.2 Python

Algorytm napisany w języku Python nie obciąża zasobów tak bardzo jak ten napisany w C++, ale czas wykonania programu jest wielokrotnie dłuższy.



Rysunek 6.15 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

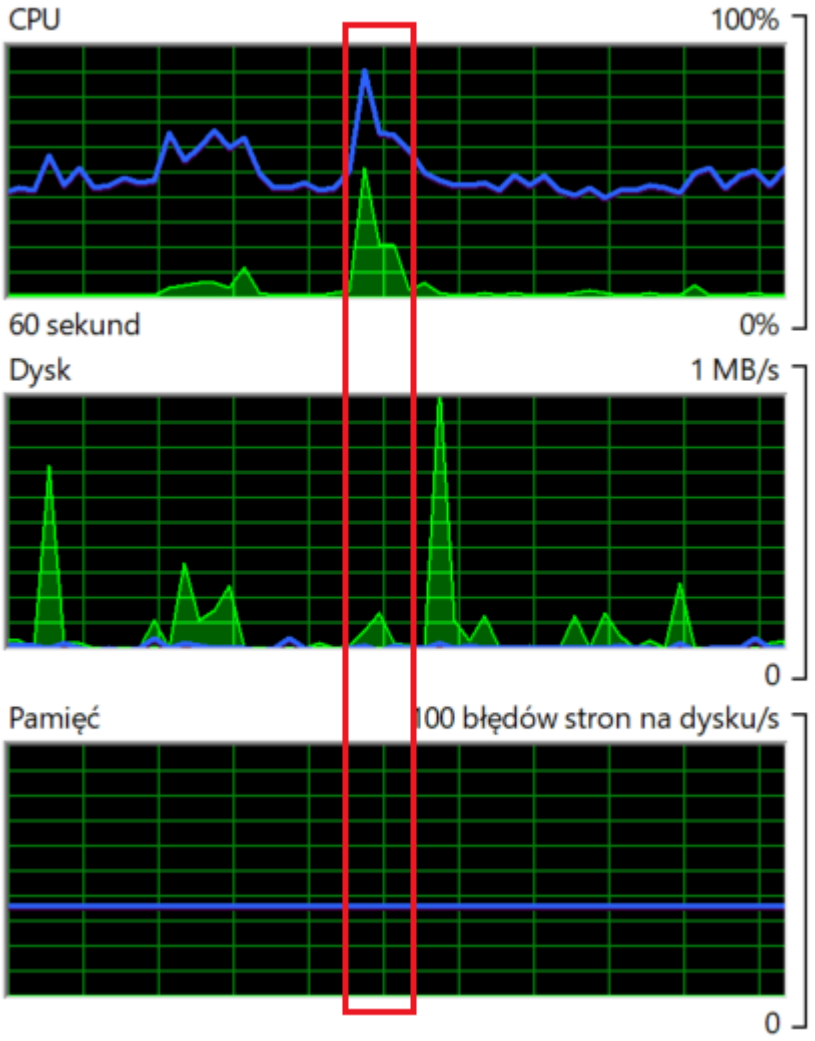


Rysunek 6.16 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

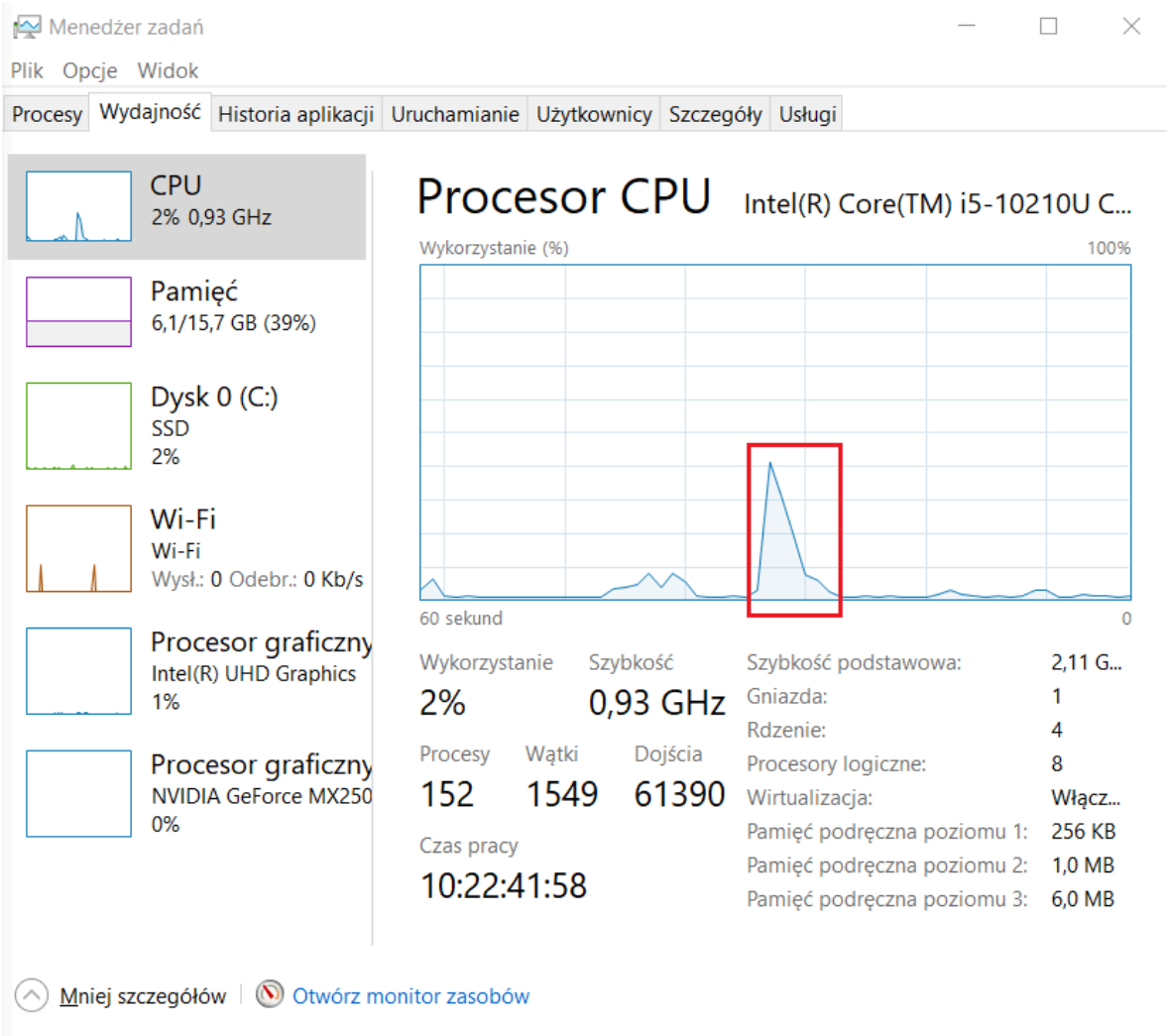
Średni czas wykonania programu to 23 ms. Zużycie procesora wahało się między 10 a 20%. Zużycie dysku dochodziło do 100 KB/s. Zużycie pamięci wynosiło poniżej 10 błędów stron na dysku/s.

6.3.3 Java

Algorytm napisany w języku Java nie obciąża zasobów komputera tak jak pozostałe implementacje. Charakteryzuje się przy tym krótkim czasem wykonania programu.



Rysunek 6.17 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]



Rysunek 6.18 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Średni czas wykonania programu to 3 ms. Zużycie procesora wahało się między 40 a 50%. Zużycie dysku wahało się między ok. 0,1 a 0,2 MB/s. Zużycie pamięci wynosiło poniżej 10 błędów stron na dysku/s.

W tabelach 6.5 i 6.6 przedstawiono czas wykonania programu wyrażony w milisekundach dla każdej implementacji algorytmu w czasie dziesięciu przeprowadzonych badań.

	1	2	3	4	5
C++	3 ms	4 ms	4 ms	5 ms	6 ms
Python	22 ms	22 ms	21 ms	24 ms	23 ms
Java	3 ms	2 ms	2 ms	3 ms	4 ms

Tabela 6.5 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu sortowania szybkiego (testy od 1 do 5) [opracowanie własne]

	6	7	8	9	10
C++	3 ms	3 ms	4 ms	5 ms	3 ms
Python	25 ms	23 ms	23 ms	24 ms	23 ms
Java	4 ms	3 ms	4 ms	3 ms	2 ms

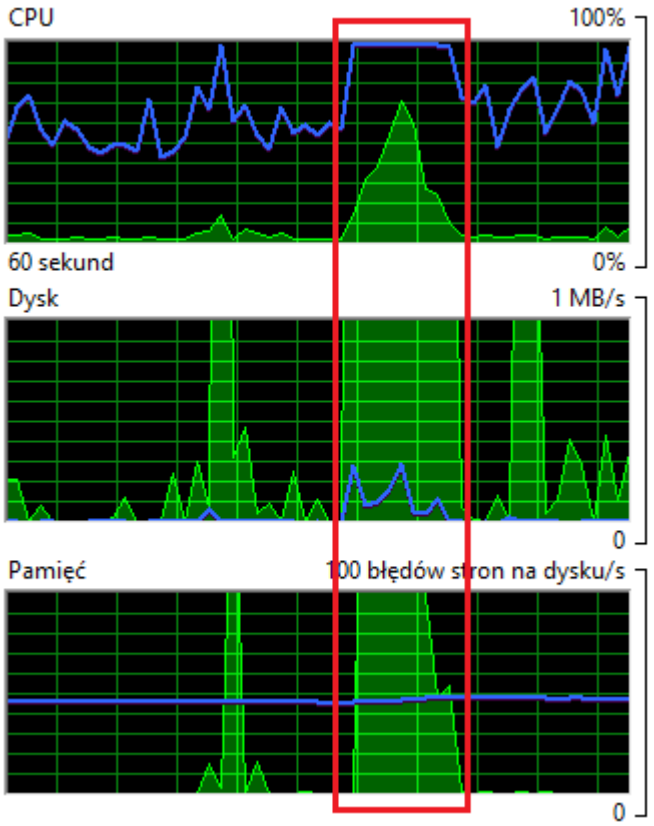
Tabela 6.6 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu sortowania szybkiego (testy od 6 do 10) [opracowanie własne]

6.4. Wyniki dla algorytmu Blowfish

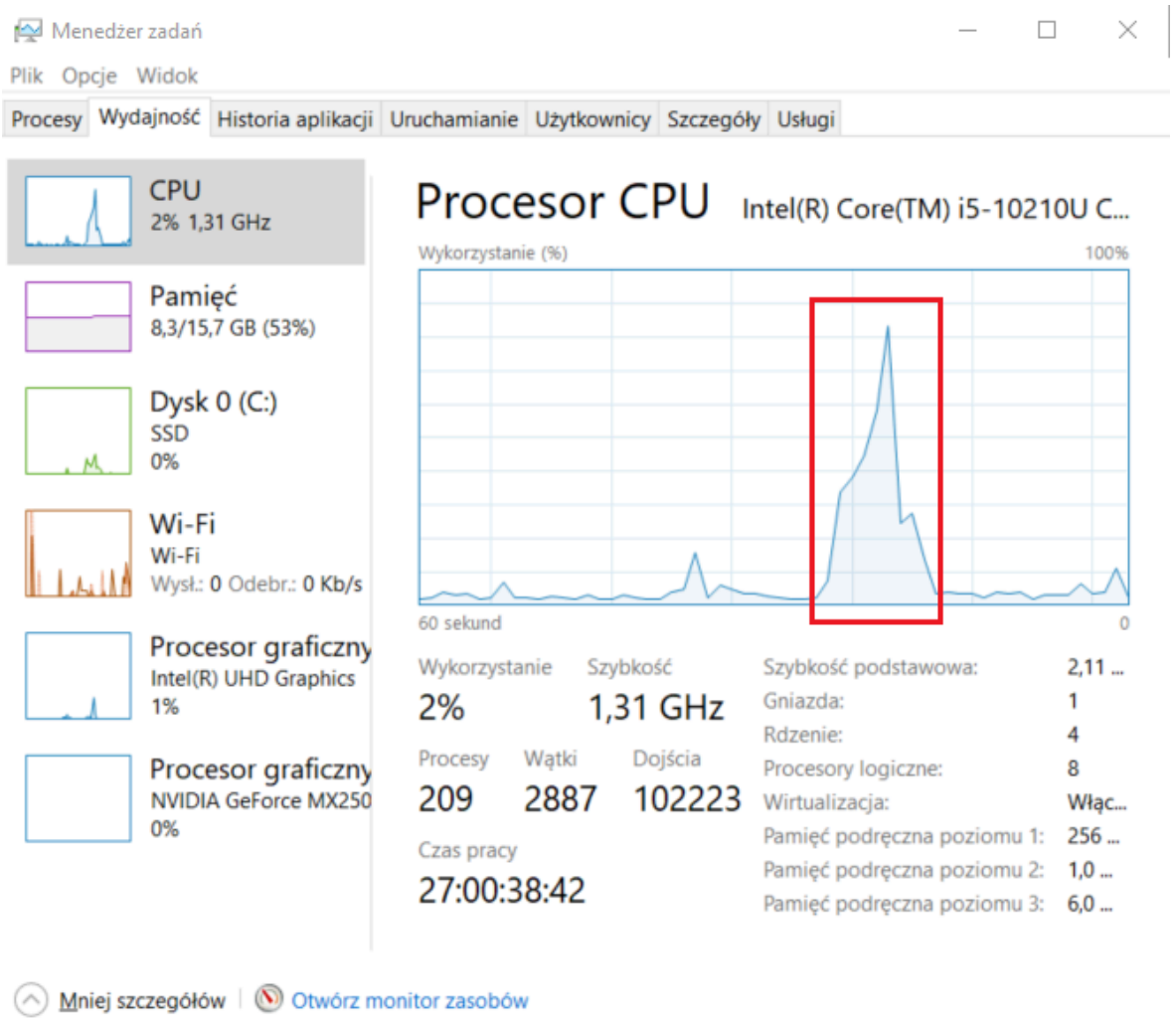
Wszystkie implementacje algorytmu zostały przetestowane na tym samym zestawie danych do zaszyfrowania.

6.4.1 C++

Implementacja algorytmu napisana w języku C++ w największym stopniu wykorzystuje zasoby komputera. Zużycie dysku przez cały czas wykonywania programu pozostawało na poziomie 1 MB/s.



Rysunek 6.19 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

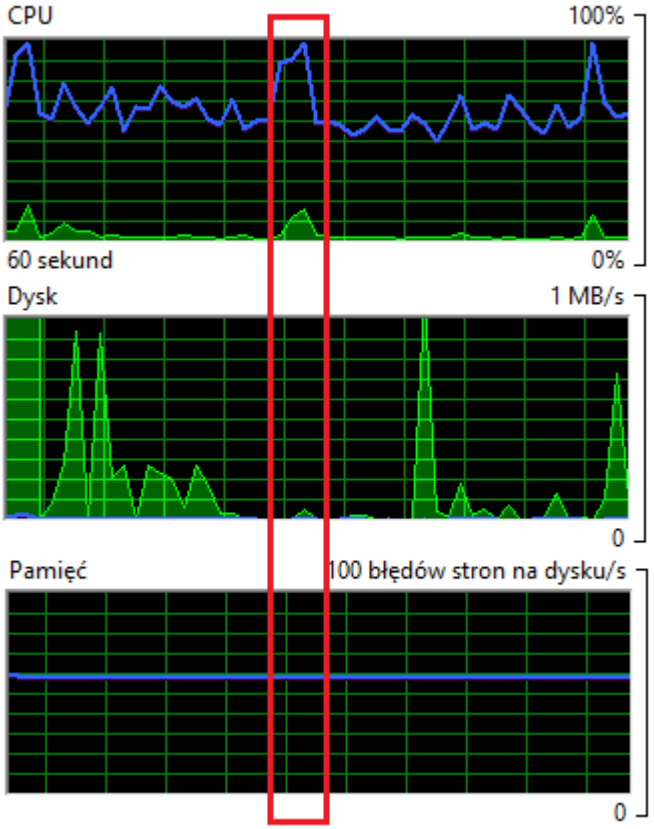


Rysunek 6.20 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

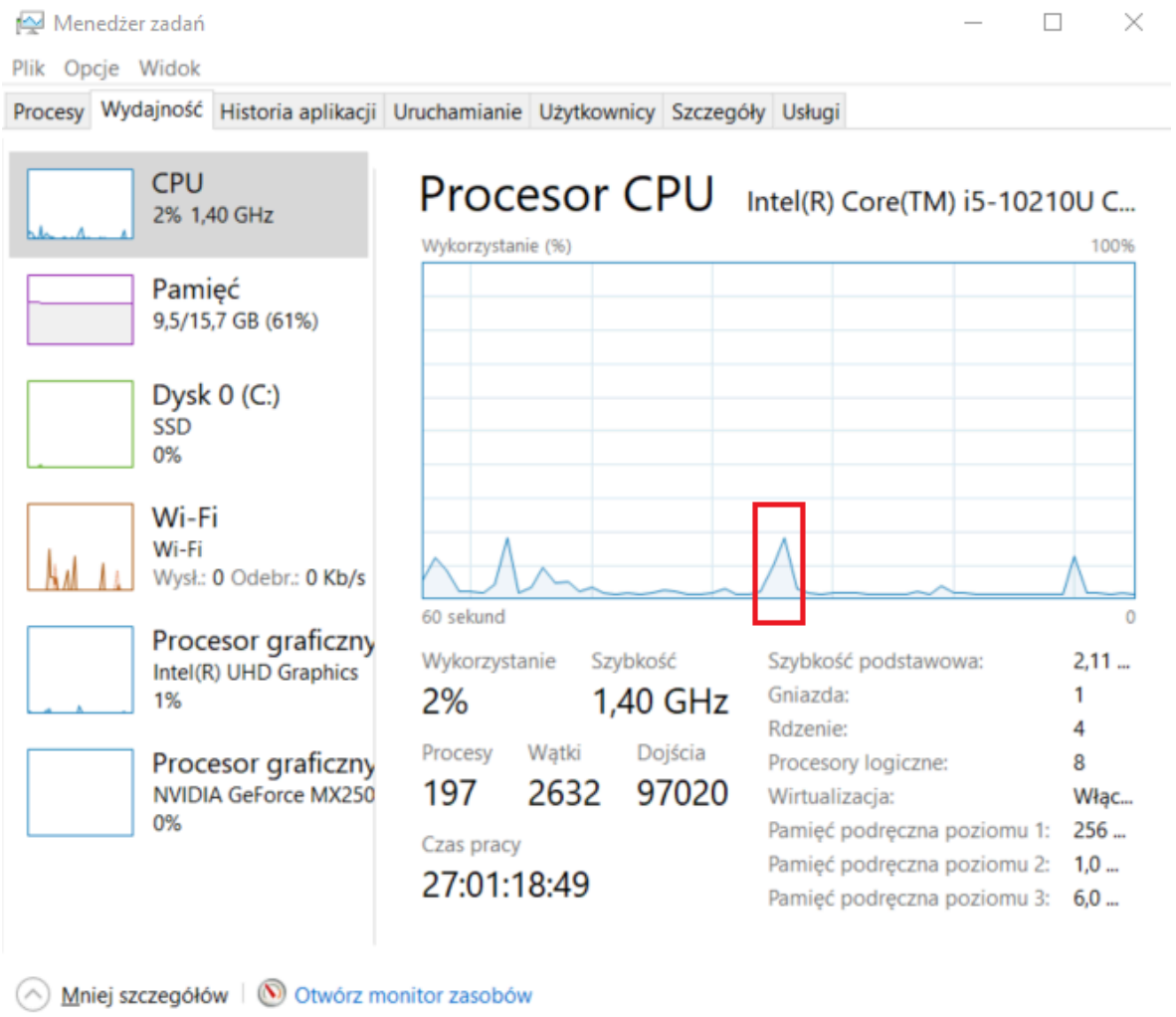
Średni czas wykonania programu to 5 ms. Zużycie procesora wahało się między 70 a 80%. Zużycie dysku wahało się między 0,9 a 1 MB/s. Zużycie pamięci w szczytowych momentach wahało się między 90 a 100 błędów stron na dysku/s.

6.4.2 Python

Implementacja algorytmu napisana w języku Python okazała się być tą, która charakteryzuje się najlepszą wydajnością. Zużycie zasobów pozostaje najniższe, podobnie jak czas wykonania programu.



Rysunek 6.21 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

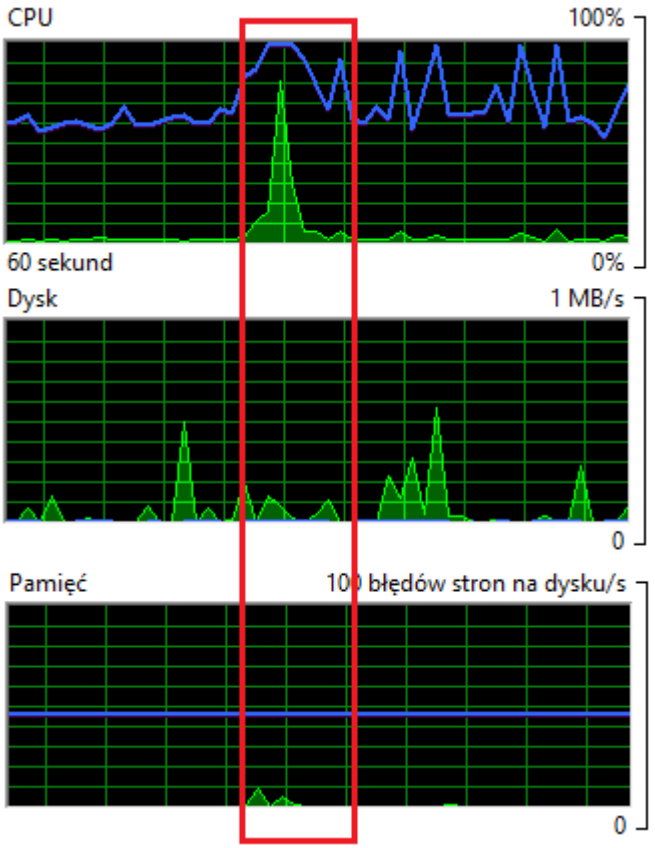


Rysunek 6.22 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

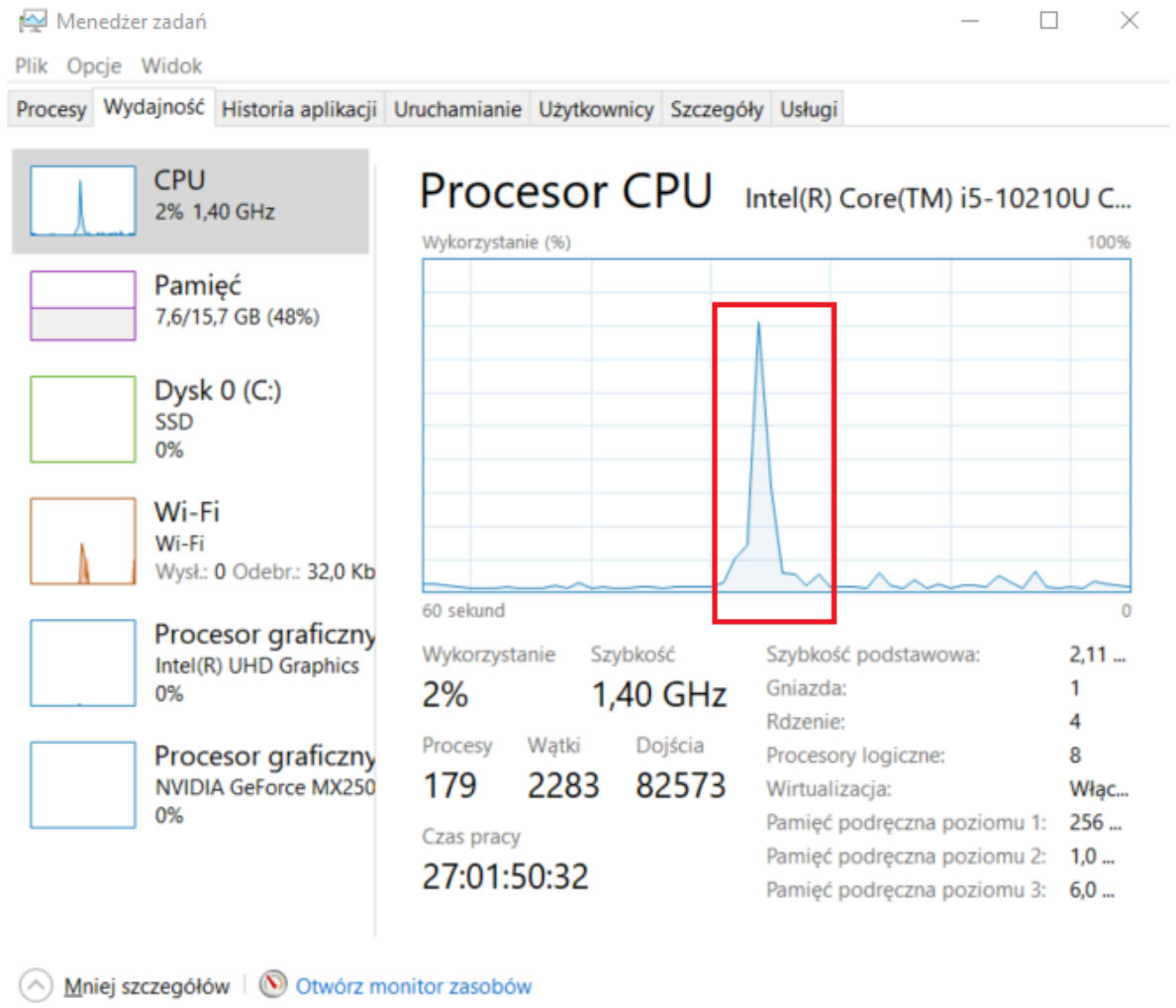
Średni czas wykonania programu to 3 ms. Zużycie procesora wahało się między 10 a 20%. Zużycie dysku wynosiło do 0,1 MB/s. Zużycie pamięci wynosiło do 10 błędów stron na dysku/s.

6.4.3 Java

Implementacja algorytmu napisana w języku Java charakteryzuje się podobnym czasem wykonania do tej napisanej w języku C++, ale zużywa przy tym zdecydowanie mniej zasobów komputera.



Rysunek 6.23 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]



Rysunek 6.24 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Średni czas wykonania programu to 4 ms. Zużycie procesora wahało się między 70 a 80%. Zużycie dysku wahało się między 0,1 a 0,2 MB/s. Zużycie pamięci w szczytowych momentach wahało się między 10 a 20 błędów stron na dysku/s.

W tabelach 6.7 i 6.8 przedstawiono czas wykonania programu wyrażony w milisekundach dla każdej implementacji algorytmu w czasie dziesięciu przeprowadzonych badań.

	1	2	3	4	5
C++	3 ms	4 ms	5 ms	5 ms	5 ms
Python	3 ms	3 ms	2 ms	4 ms	3 ms
Java	4 ms	3 ms	3 ms	4 ms	6 ms

Tabela 6.7 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu Blowfish (testy od 1 do 5) [opracowanie własne]

	6	7	8	9	10
C++	6 ms	6 ms	5 ms	4 ms	7 ms
Python	2 ms	5 ms	2 ms	3 ms	3 ms
Java	4 ms	5 ms	5 ms	3 ms	3 ms

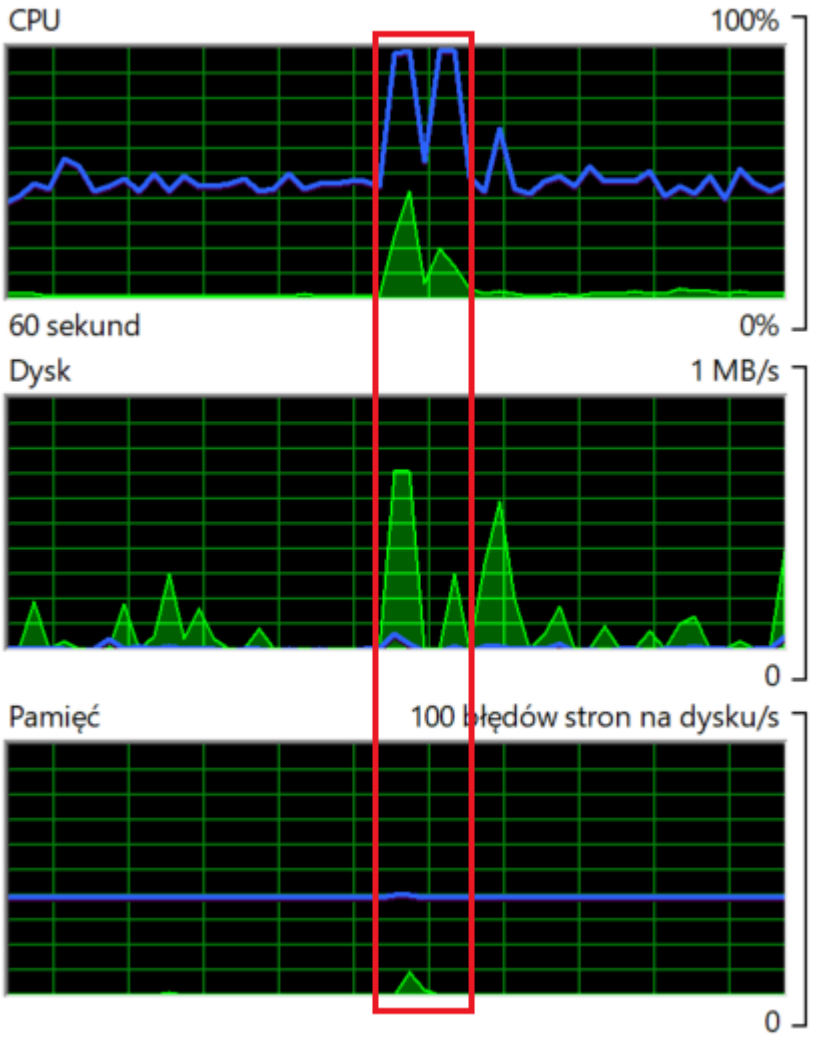
Tabela 6.8 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu Blowfish (testy od 6 do 10) [opracowanie własne]

6.5. Wyniki dla algorytmu kompresji metodą RLE

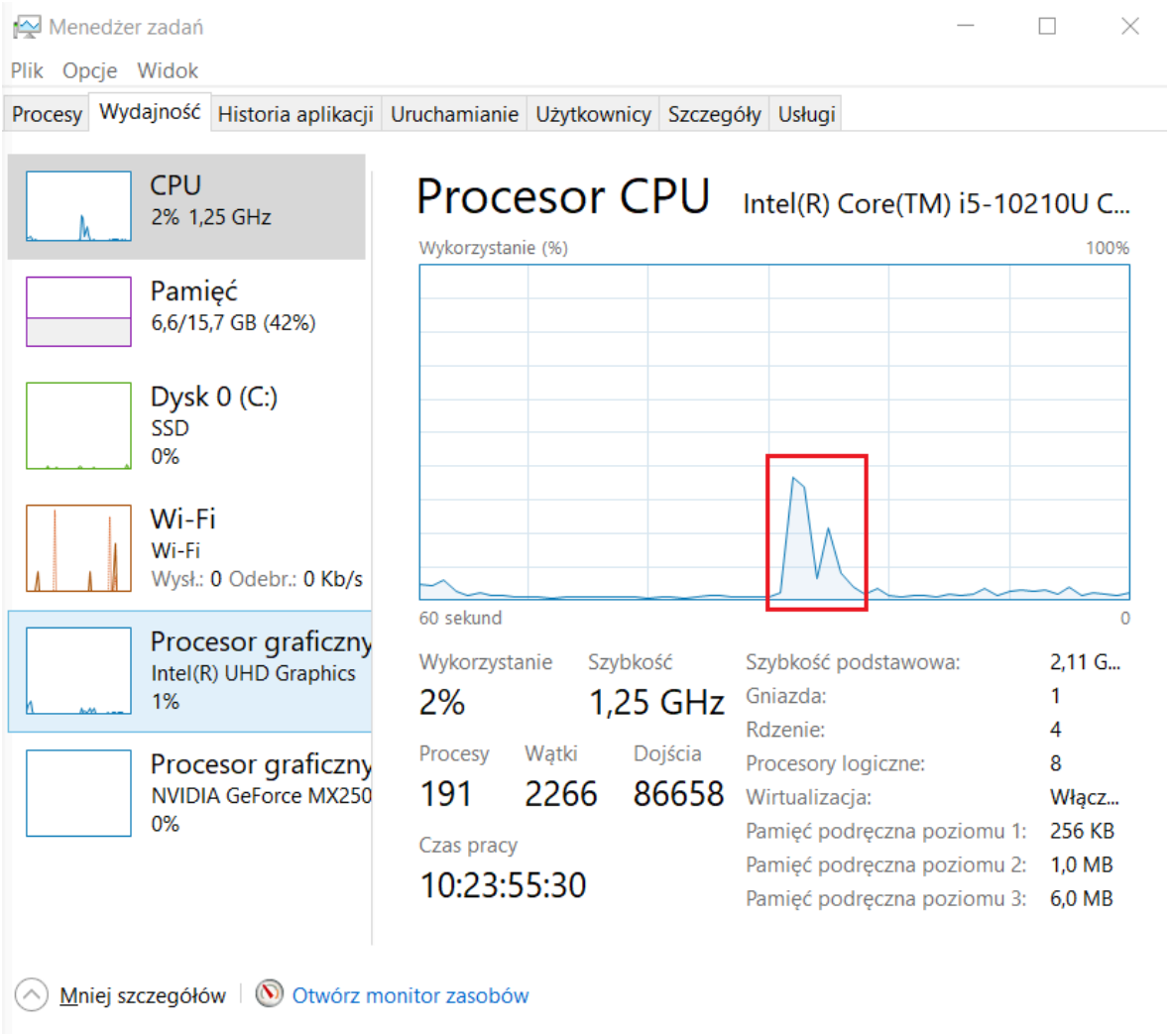
Wszystkie implementacje algorytmu zostały przetestowane na takim samym ciągu 1704 znaków.

6.5.1 C++

Implementacje algorytmu napisana w języku C++ charakteryzuje się średnim zużyciem zasobów procesora oraz dużym zużyciem zasobów dysku. Wykorzystanie pamięci pozostawało na niskim poziomie i nie przekraczało 10 błędów stron na dysku/s.



Rysunek 6.25 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

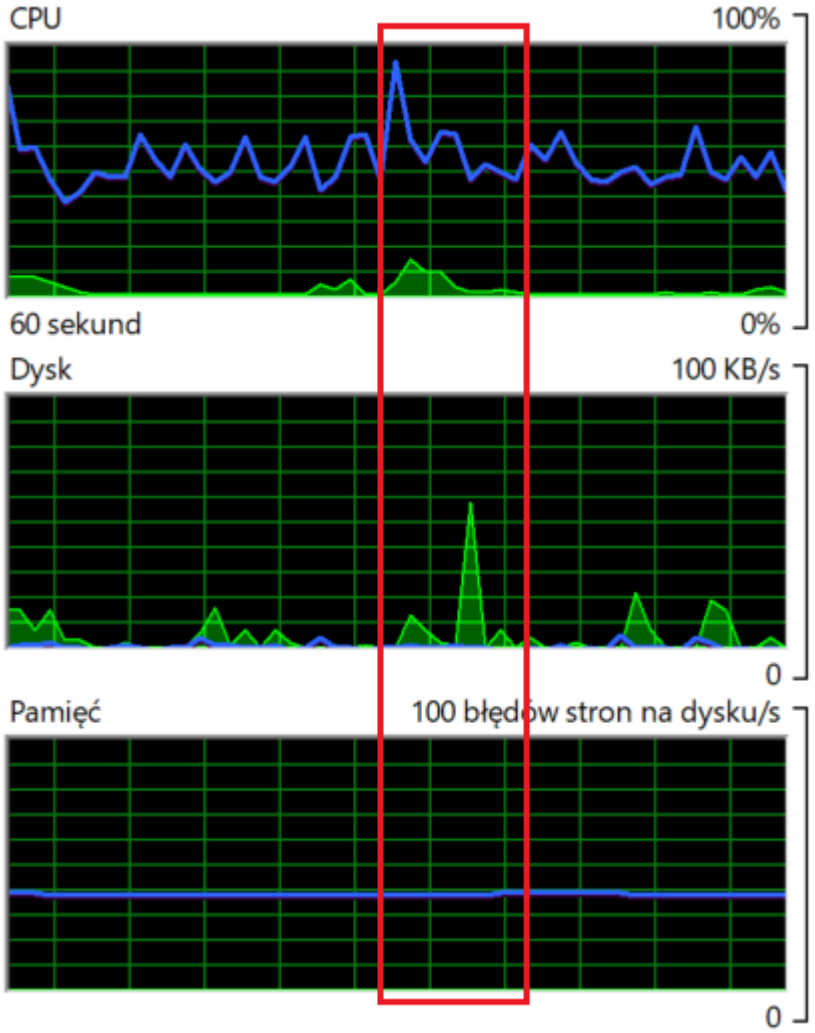


Rysunek 6.26 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

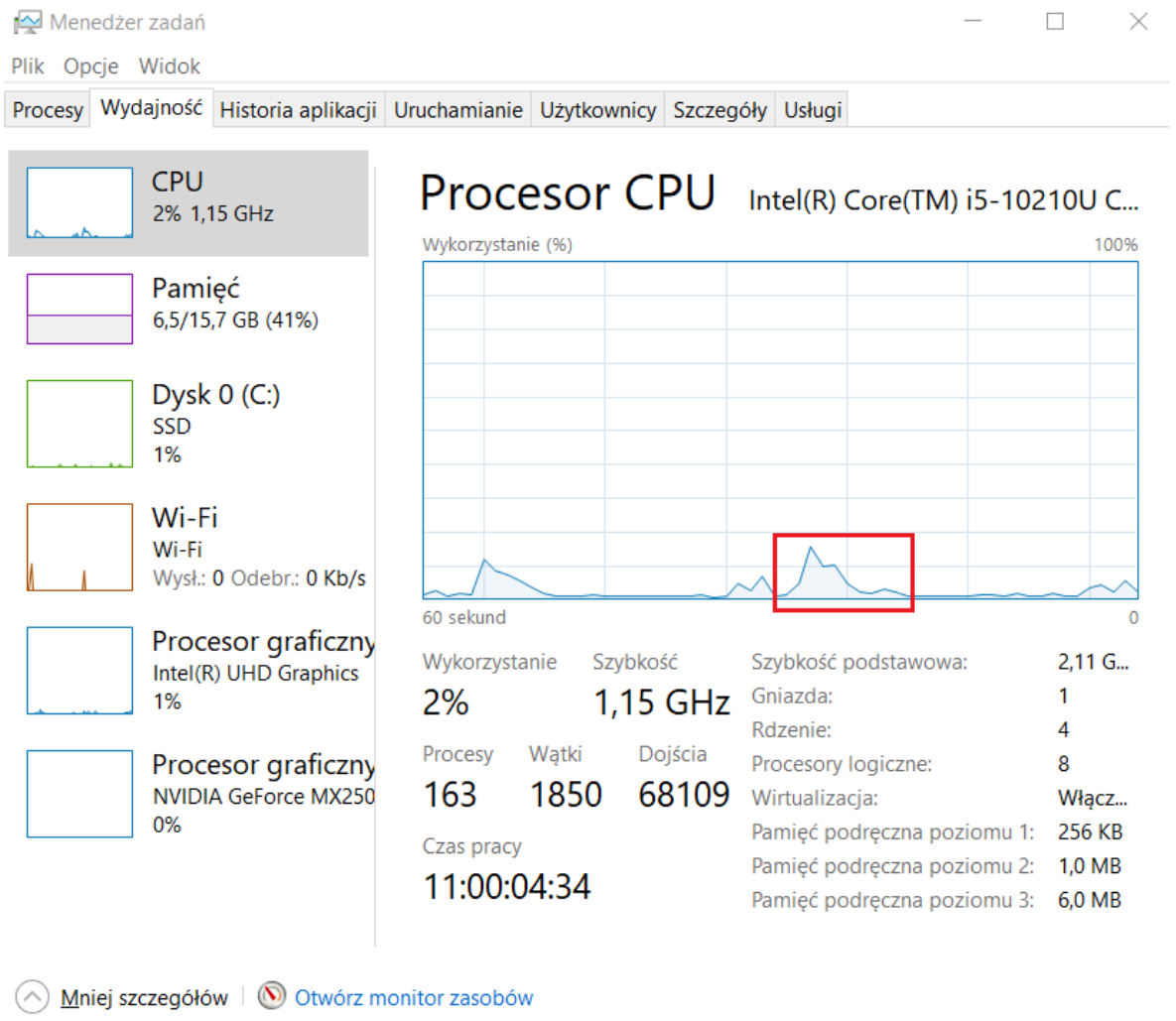
Średni czas wykonania programu to 3 ms. Zużycie procesora wahało się między 30 a 40%. Zużycie dysku wahało się między ok. 0,7 a 0,8 MB/s. Zużycie pamięci wyniosło poniżej 10 błędów stron na dysku/s.

6.5.2 Python

Implementacja algorytmu napisana w języku Python okazała się wydajna pod względem zużycia zasobów procesora i pamięci. Zużycie dysku osiągało relatywnie niski poziom dochodzący nawet do 60 KB/s.



Rysunek 6.27 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

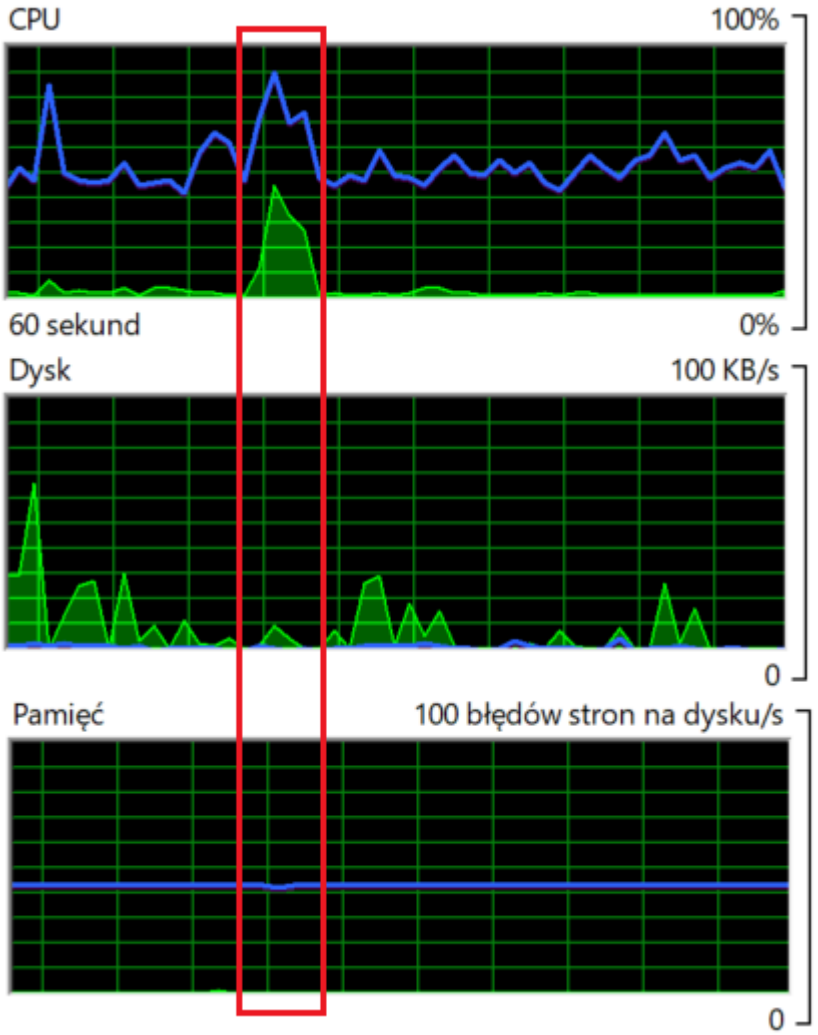


Rysunek 6.28 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

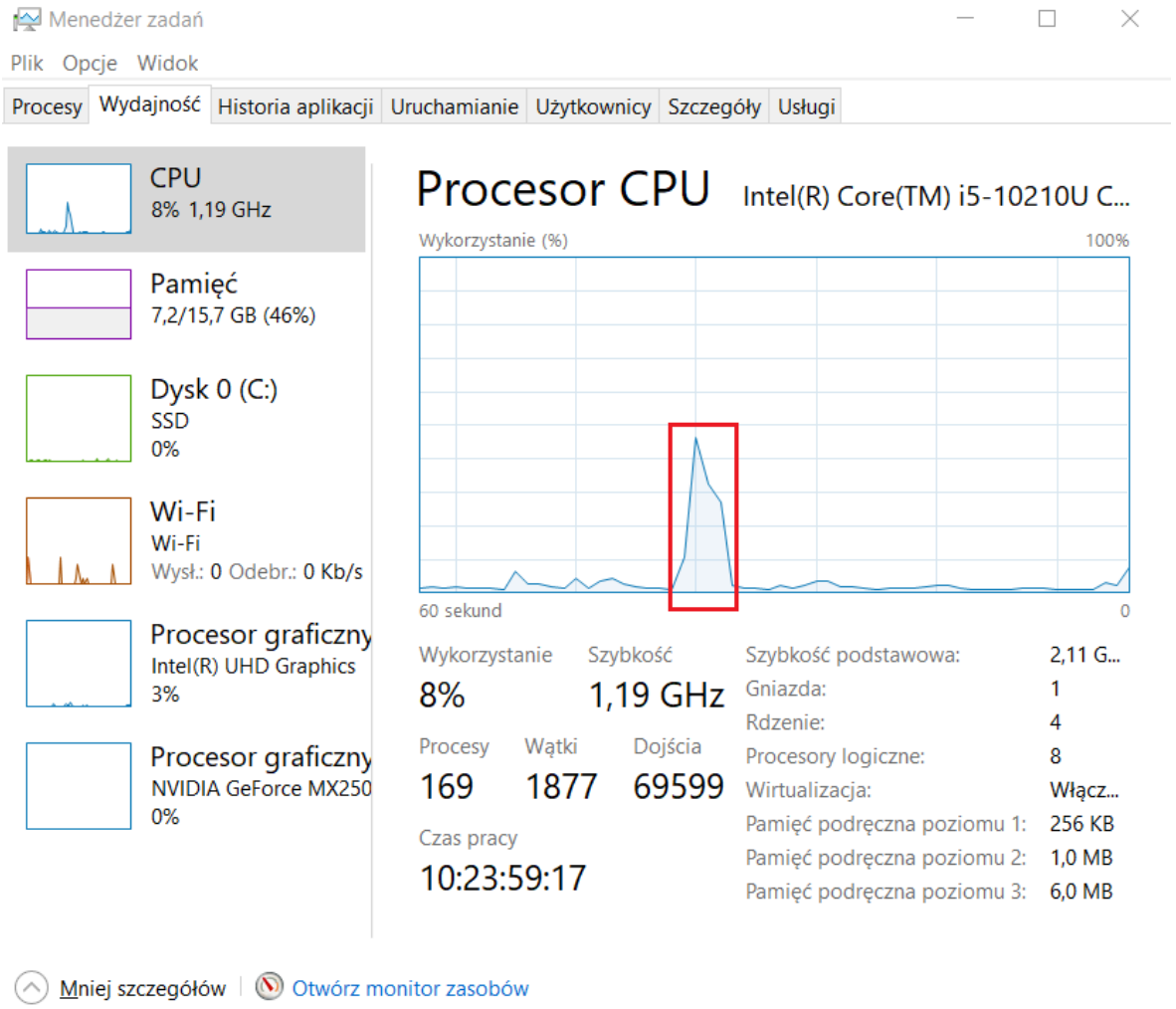
Średni czas wykonania programu to 5 ms. Zużycie procesora wahało się między 10 a 20%. Zużycie dysku wahało się między ok. 50 a 60 KB/s. Zużycie pamięci wyniosło poniżej 10 błędów stron na dysku/s.

6.5.3 Java

Implementacja algorytmu napisana w języku Java charakteryzuje się podobnym zużyciem zasobów procesora, ale zdecydowanie przewyższa wszystkie wcześniejsze implementacje pod względem wykorzystania zasobów dysku.



Rysunek 6.29 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]



Rysunek 6.30 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Średni czas wykonania programu to 3 ms. Zużycie procesora wahało się między 40 a 50%. Zużycie dysku wynosiło do 0,1 MB/s. Zużycie pamięci wyniosło poniżej 10 błędów stron na dysku/s.

W tabelach 6.9 i 6.10 przedstawiono czas wykonania programu wyrażony w milisekundach dla każdej implementacji algorytmu w czasie dziesięciu przeprowadzonych badań.

	1	2	3	4	5
C++	1 ms	3 ms	2 ms	4 ms	4 ms
Python	3 ms	7 ms	4 ms	5 ms	5 ms
Java	2 ms	5 ms	3 ms	4 ms	3 ms

Tabela 6.9 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą RLE (testy od 1 do 5) [opracowanie własne]

	6	7	8	9	10
C++	3 ms	5 ms	3 ms	3 ms	2 ms
Python	6 ms	4 ms	5 ms	5 ms	6 ms
Java	2 ms	3 ms	3 ms	2 ms	3 ms

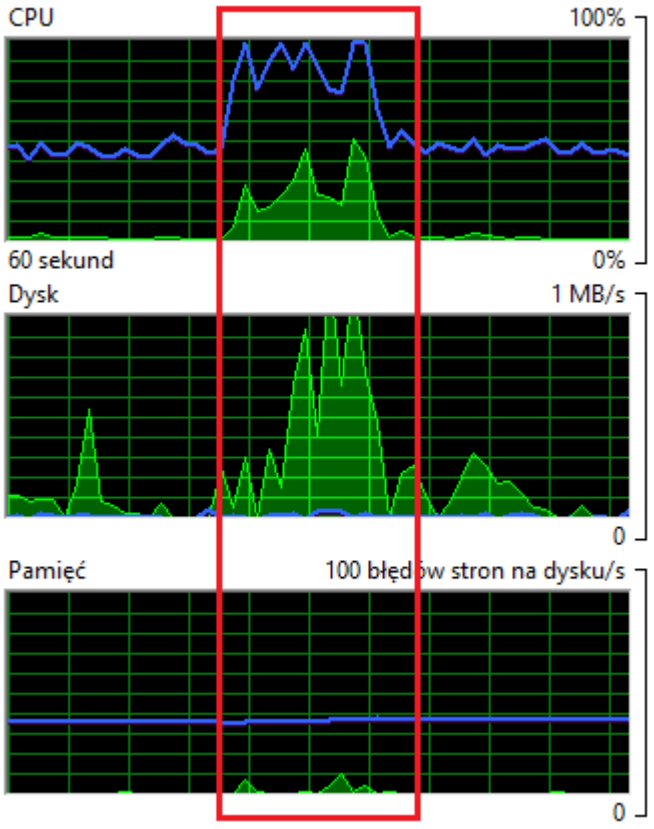
Tabela 6.10 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą RLE (testy od 6 do 10) [opracowanie własne]

6.6. Wyniki dla algorytmu kompresji metodą LZW

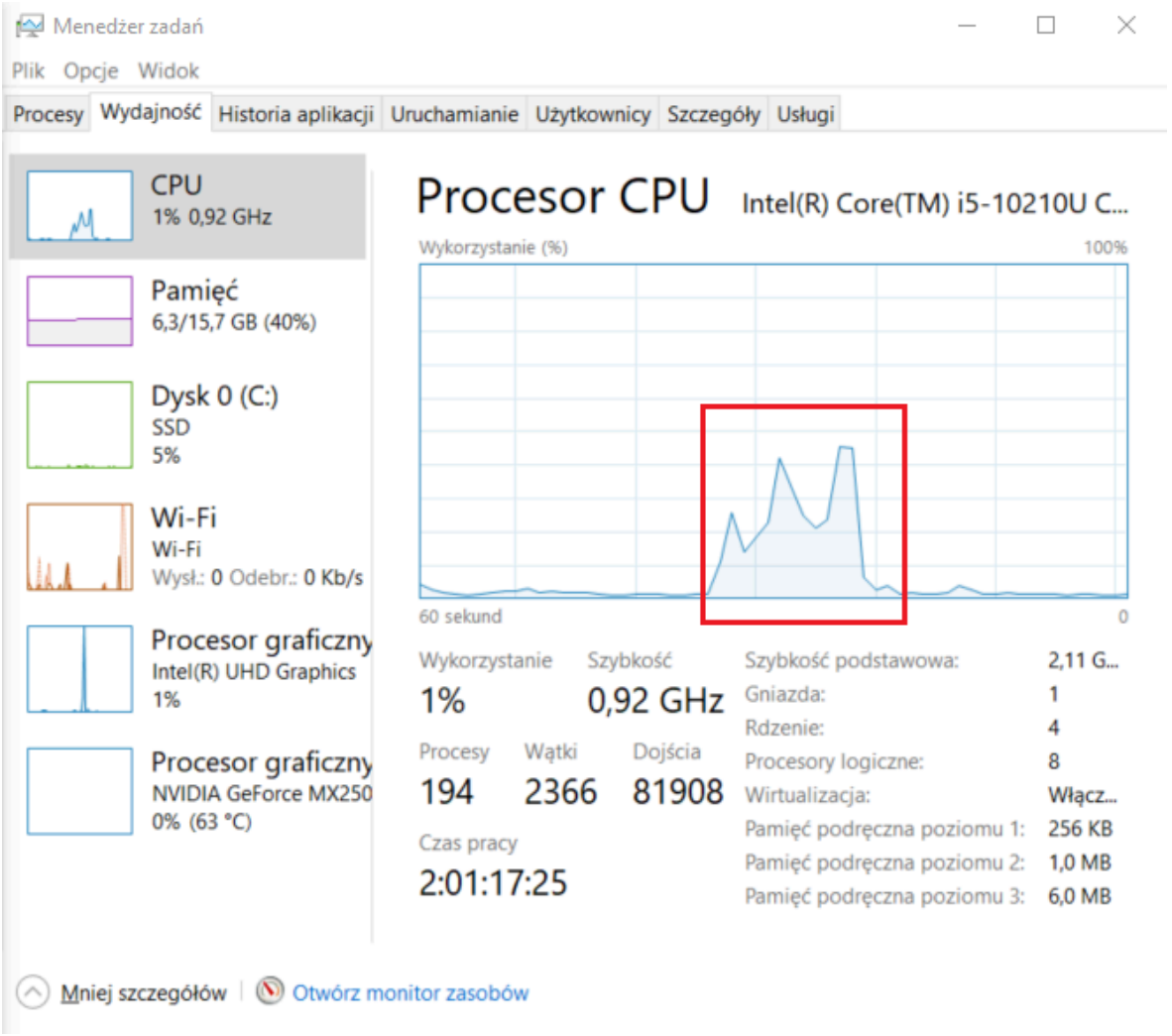
Wszystkie implementacje algorytmów zostały przetestowane na tym samym ciągu znaków: *WYS*WYGWYS*WYSWYSG*.

6.6.1 C++

Implementacja algorytmu napisana w języku C++ charakteryzuje się najdłuższym czasem wykonania programu spośród wszystkich implementacji. Zużycie dysku pozostawało na wysokim poziomie.



Rysunek 6.31 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

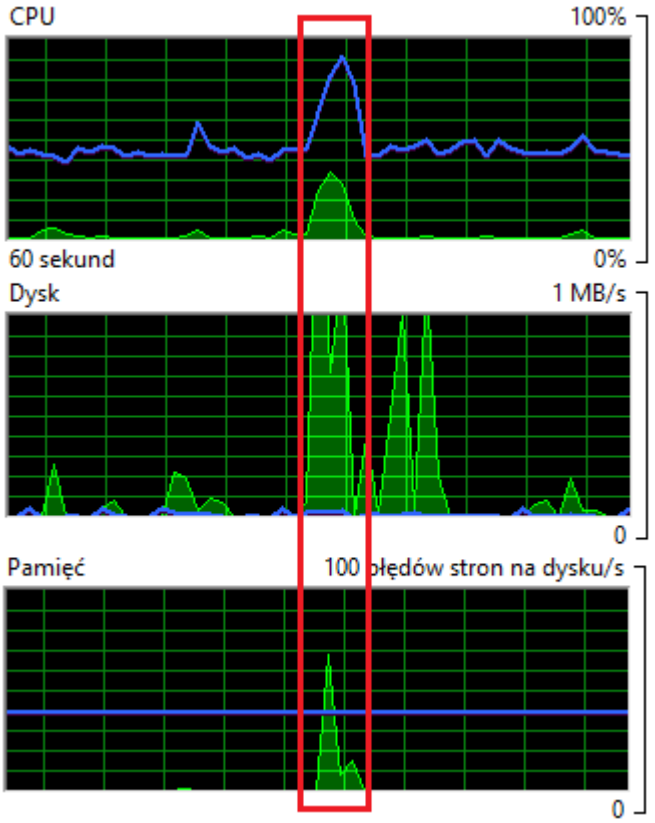


Rysunek 6.32 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

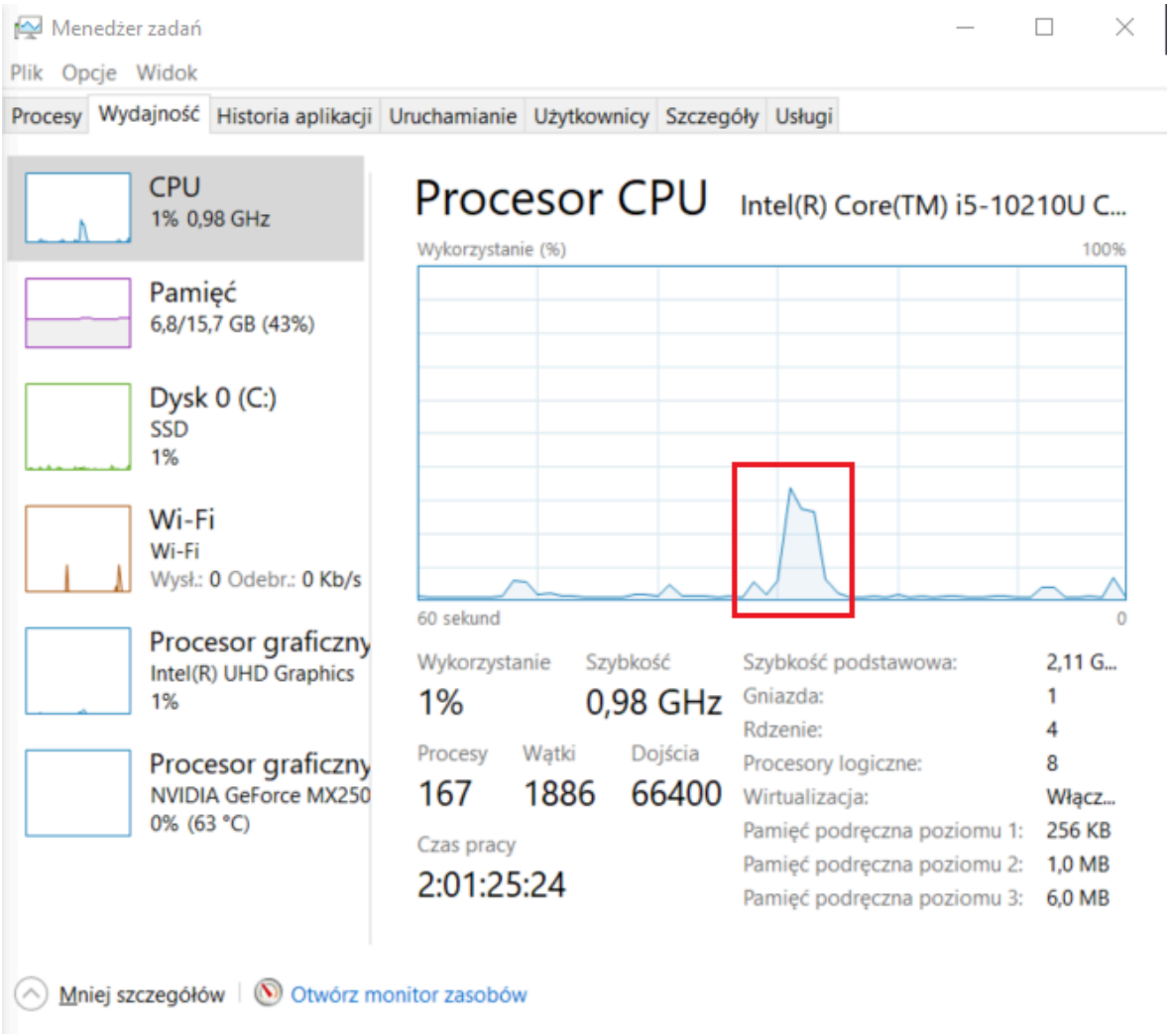
Średni czas wykonania programu to 7 ms. Zużycie procesora wahało się między 50 a 60%. Zużycie dysku wahało się między ok. 0,9 a 1 MB/s. Zużycie pamięci wyniosło poniżej 10 błędów stron na dysku/s.

6.6.2 Python

Implementacja algorytmu napisana w języku Python charakteryzuje się krótkim czasem wykonania. Duże pozostawało zużycie dysku. Wykorzystanie pamięci było największe spośród wszystkich implementacji algorytmu.



Rysunek 6.33 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

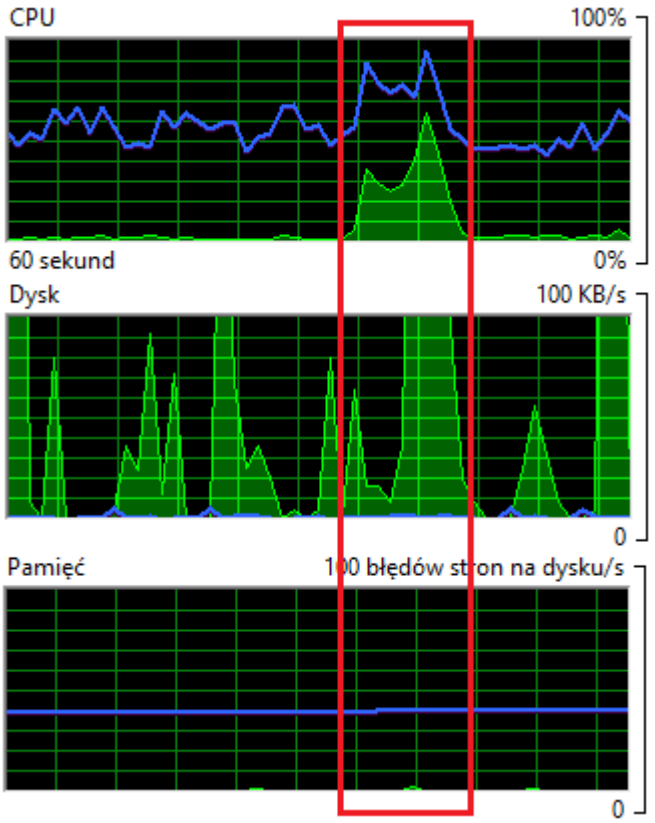


Rysunek 6.34 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

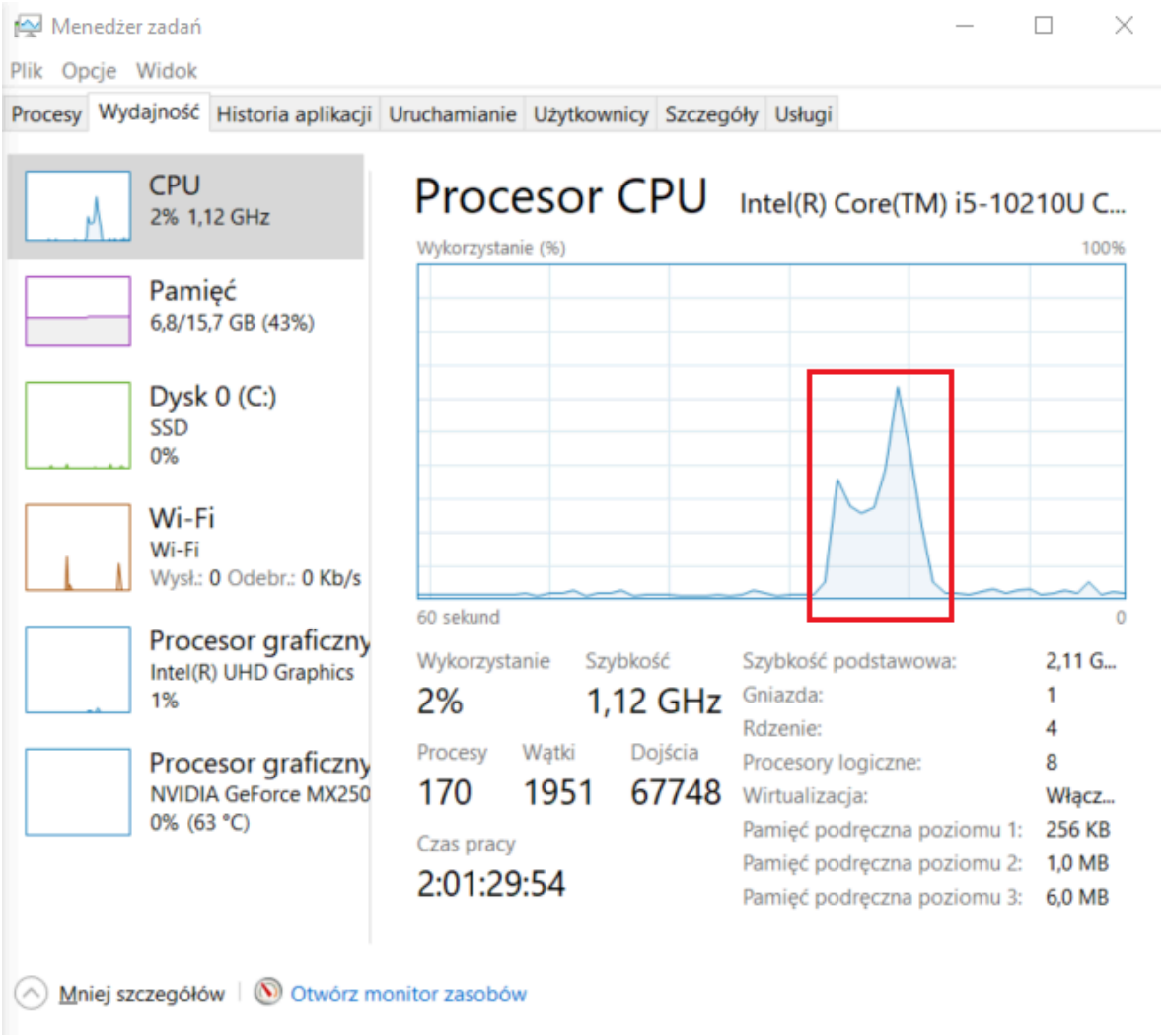
Średni czas wykonania programu to 2 ms. Zużycie procesora wahało się między 30 a 40%. Zużycie dysku wahało się między ok. 0,9 a 1 MB/s. Zużycie pamięci wyniosło poniżej 10 błędów stron na dysku/s.

6.6.3 Java

Implementacja algorytmu napisana w języku Java charakteryzuje porównywalnym zużyciem CPU do implementacji napisanej w języku C++. Jednakowo czas wykonania programu był krótszy, a zużycie dysku mniejsze.



Rysunek 6.35 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]



Rysunek 6.36 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Średni czas wykonania programu to 5 ms. Zużycie procesora wahało się między 70 a 80%. Zużycie dysku wahało się między ok. 90 a 100 KB/s. Zużycie pamięci wyniosło poniżej 10 błędów stron na dysku/s.

W tabelach 6.11 i 6.12 przedstawiono czas wykonania programu wyrażony w milisekundach dla każdej implementacji algorytmu w czasie dziesięciu przeprowadzonych badań.

	1	2	3	4	5
C++	4 ms	9 ms	5 ms	7 ms	7 ms
Python	1 ms	1 ms	3 ms	2 ms	3 ms
Java	4 ms	5 ms	6 ms	5 ms	5 ms

Tabela 6.11 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą LZW (testy od 1 do 5) [opracowanie własne]

	6	7	8	9	10
C++	6 ms	8 ms	7 ms	9 ms	8 ms
Python	1 ms	2 ms	2 ms	2 ms	3 ms
Java	7 ms	4 ms	5 ms	4 ms	5 ms

Tabela 6.12 Tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą LZW (testy od 6 do 10) [opracowanie własne]

7. Podsumowanie i wnioski

Podsumowanie wyników można przedstawić w następujący sposób. Kolorem zielonym zaznaczono najlepsze wyniki z każdego wiersza tabeli:

- Dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej:

	C++	Python	Java
Czas wykonania [ms]	32	214	8
Zużycie CPU [%]	30 - 40	20 - 30	30 - 40
Zużycie dysku [MB/s]	0,9 - 1	0,4 - 0,5	0,4 - 0,5
Zużycie pamięci [błędy stron na dysku/s]	20 - 30	< 10	< 10

Tabela 7.1 Porównanie wydajności dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej [opracowanie własne]

- Dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej:

	C++	Python	Java
Czas wykonania [ms]	1	414	25
Zużycie CPU [%]	50 - 60	30 - 40	50 - 60
Zużycie dysku [MB/s]	0,9 - 1	0,9 - 1	0,9 - 1
Zużycie pamięci [błędy stron na dysku/s]	10 - 20	10 - 20	90 - 100

Tabela 7.2 Porównanie wydajności dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej [opracowanie własne]

- Dla algorytmu sortowania szybkiego:

	C++	Python	Java
Czas wykonania [ms]	4	23	3
Zużycie CPU [%]	50 - 60	10 - 20	40 - 50
Zużycie dysku [MB/s]	0,9 - 1	< 0,1	0,1 - 0,2
Zużycie pamięci [błędy stron na dysku/s]	10 - 20	< 10	< 10

Tabela 7.3 Porównanie wydajności dla algorytmu sortowania szybkiego [opracowanie własne]

- Dla algorytmu Blowfish:

	C++	Python	Java
Czas wykonania [ms]	6	31	4
Zużycie CPU [%]	70 - 80	10 - 20	70 - 80
Zużycie dysku [MB/s]	0,9 - 1	< 0,1	0,1 - 0,2
Zużycie pamięci [błędy stron na dysku/s]	90 - 100	< 10	10 - 20

Tabela 7.4 Porównanie wydajności dla algorytmu Blowfish [opracowanie własne]

- Dla algorytmu kompresji metodą RLE:

	C++	Python	Java
Czas wykonania [ms]	3	5	3
Zużycie CPU [%]	30 - 40	10 - 20	30 - 50
Zużycie dysku [MB/s]	0,7 - 0,8	0,05 - 0,06	< 0,01
Zużycie pamięci [błędy stron na dysku/s]	< 10	< 10	< 10

Tabela 7.5 Porównanie wydajności dla algorytmu kompresji metodą RLE [opracowanie własne]

- Dla algorytmu kompresji metodą LZW:

	C++	Python	Java
Czas wykonania [ms]	7	2	5
Zużycie CPU [%]	50 - 60	30 - 40	70 - 80
Zużycie dysku [MB/s]	0,9 - 1	0,9 - 1	0,09 - 0,1
Zużycie pamięci [błędy stron na dysku/s]	< 10	< 10	< 10

Tabela 7.6 Porównanie wydajności dla algorytmu kompresji metodą LZW [opracowanie własne]

Przed przeprowadzeniem badań na podstawie wiedzy o specyfice każdego z języków programowania wybranych do testów można by spodziewać się, że najlepszym okaże się C++. Wyniki pokazują jednak, że najlepiej w badaniach poradził sobie język Python. Języki C++ i Java osiągały podobne wyniki, z niewielką przewagą drugiego z nich.

Powodem tego było dobranie małych zestawów danych do testów. W przypadku algorytmów, które w największym stopniu obciążały komputer (wyszukiwanie n-tej liczby ciągu Fibonacciego) można zaobserwować, że lepsze wyniki uzyskujemy dla języka C++. Implementacje algorytmów napisane w Javie nie dorównywały tym napisanym w C++ z uwagi na konieczność użycia maszyny wirtualnej.

Należy zatem wysunąć wnioski, że do projektów informatycznych związanych z algorytmiką wykorzystujących niewielkie zbiory danych najbardziej odpowiednim językiem będzie Python. Jeżeli danych będzie więcej, odpowiednim wyborem będzie C++. Należy się przy tym liczyć jednak z wyższym zużyciem zasobów komputera.

Bibliografia

[1] Herbert Schildt. *Java. Kompendium programisty. Wydanie VIII*, wyd. Helion

[2] Piotr Wróblewski. *Algorytmy, struktury danych i techniki programowania*, wyd. Helion

[3] *HTTP* - FAQ prowadzone przez Bjarne Stroustrupa [Online] https://www.stroustrup.com/bs_faq.html [dostęp 2023-12-03]

[4] Bjarne Stroustrup - *Why C++ is not just an Object-Oriented Programming Language*, AT&T Bell Laboratories Murray Hill

[5] *HTTP* - Python 3.12.0 dokumentacja [Online] https://docs.python.org/3/ [dostęp 2023-12-03]

[6] *HTTP* - dokumentacja Visual Studio Code 1.83 [Online] https://code.visualstudio.com/docs [dostęp 2023-12-03]

[7] *HTTP* - witryna poświęcona Visual Studio [Online] https://visualstudio.microsoft.com/pl/ [dostęp 2023-12-03]

[8] *HTTP* - witryna poświęcona IntelliJ IDEA [Online] https://www.jetbrains.com/idea/ [dostęp 2023-12-03]

[9] *HTTP* - witryna poświęcona PyCharm [Online] https://www.jetbrains.com/pycharm/ [dostęp 2023-12-03]

[10] *HTTP* - ankieta przedstawiająca najpopularniejsze języki programowania wybrane przez użytkowników witryny Stack Overflow w 2022 roku https://survey.stackoverflow.co/2022/#most-popular-technologies-language [dostęp 2023-12-03]

[11] *HTTP* - dokumentacja języka HTML [Online] https://developer.mozilla.org/en-US/docs/Web/HTML?retiredLocale=pl [dostęp 2023-12-03]

[12] *HTTP* - dokumentacja języka SQL [Online] https://www.red-gate.com/products/sql-doc/ [dostęp 2023-12-03]

[13] *HTTP* - dokumentacja języka JavaScript [Online] https://developer.mozilla.org/en-US/docs/Web/JavaScript [dostęp 2023-12-03]

[14] *HTTP* - dokumentacja języka TypeScript [Online] https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html [dostęp 2023-12-03]

[15] *HTTP* - dokumentacja BASH [Online] https://www.gnu.org/software/bash/manual/bash.html [dostęp 2023-12-03]

[16] *HTTP* - encyklopedia PWN [Online] https://encyklopedia.pwn.pl [dostęp 2023-12-03]

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald. R. Rivest. *Wprowadzenie do algorytmów*. wyd. Wydawnictwa Naukowo-Techniczne

[18] *HTTP* - opis algorytmu Blowfish [Online] https://www.schneier.com/cryptography/archives/1994/09/description_of_a_new.html [zarchiwizowano 2016-03-04] https://web.archive.org/web/20160304200440/https://www.schneier.com/cryptography/archives/1994/09/description_of_a_new.html [dostęp 2024-01-14]

[19] Artur Przelaskowski. *Kompresja danych: podstawy, metody bezstratne, kodery obrazów*. wyd. BTC

Spis rysunków

Rysunek 4.1. Kod algorytmu w wersji rekurencyjnej napisany w języku C++. Użyte zostały funkcje duration_cast w celu obliczenia czasu wykonania programu [opracowanie własne]
Rysunek 4.2. Przykład inicjalizacji tablicy P i czterech tablic S napisany w języku C++ [opracowanie własne]
Rysunek 6.1 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.2 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.3 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.4 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.5 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.6 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.7 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.8 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.9 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.10 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.11 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.12 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.13 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.14 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Rysunek 6.15 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.16 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.17 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.18 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.19 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.20 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie
własne]
Rysunek 6.21 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.22 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.23 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.24 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.25 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.26 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.27 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.28 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.29 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]
Rysunek 6.30 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Rysunek 6.31 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Rysunek 6.32 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Rysunek 6.33 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Rysunek 6.34 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Rysunek 6.35 Wyniki z monitora zasobów. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Rysunek 6.36 Wyniki z menedżera zadań. Czerwonym prostokątem zaznaczono okres, w którym działał algorytm [opracowanie własne]

Spis tabel

Tabela 6.1 tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej (testy od 1 do 5) [opracowanie własne]
Tabela 6.2 tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej (testy od 6 do 10) [opracowanie własne]
Tabela 6.3 tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej (testy od 1 do 5) [opracowanie własne]
Tabela 6.4 tabela przedstawiająca czas wykonania algorytmów dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej (testy od 6 do 10) [opracowanie własne]
Tabela 6.5 tabela przedstawiająca czas wykonania algorytmów dla algorytmu sortowania szybkiego (testy od 1 do 5) [opracowanie własne]
Tabela 6.6 tabela przedstawiająca czas wykonania algorytmów dla algorytmu sortowania szybkiego (testy od 6 do 10) [opracowanie własne]
Tabela 6.7 tabela przedstawiająca czas wykonania algorytmów dla algorytmu Blowfish (testy od 1 do 5) [opracowanie własne]
Tabela 6.8 tabela przedstawiająca czas wykonania algorytmów dla algorytmu Blowfish (testy od 6 do 10) [opracowanie własne]
Tabela 6.9 tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą RLE (testy od 1 do 5) [opracowanie własne]
Tabela 6.10 tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą RLE (testy od 6 do 10) [opracowanie własne]
Tabela 6.11 tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą LZW (testy od 1 do 5) [opracowanie własne]
Tabela 6.12 tabela przedstawiająca czas wykonania algorytmów dla algorytmu kompresji metodą LZW (testy od 6 do 10) [opracowanie własne]
Tabela 7.1 Porównanie wydajności dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji rekurencyjnej [opracowanie własne]
Tabela 7.2 Porównanie wydajności dla algorytmu wyszukiwania n-tej liczby ciągu Fibonacciego w wersji iteracyjnej [opracowanie własne]
Tabela 7.3 Porównanie wydajności dla algorytmu sortowania szybkiego [opracowanie własne]

Tabela 7.4 Porównanie wydajności dla algorytmu Blowfish [opracowanie własne]
Tabela 7.5 Porównanie wydajności dla algorytmu kompresji metodą RLE [opracowanie własne]
Tabela 7.6 Porównanie wydajności dla algorytmu kompresji metodą LZW [opracowanie własne]