JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 26 (2023) 93–99
Received: 13 October 2022
Accepted: 26 October 2022

Comparative analysis of C and Python on the basis of the execution time of applications implementing selected algorithms

Analiza porównawcza języków C oraz Python na podstawie czasu wykonania aplikacji realizujących wybrane algorytmy

Paweł Rysak*

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The article deals with a comparative analysis of the speed of code execution written in the C language and Python. Its primary purpose was not to seek a simple answer to the question of which language would be more efficient, but what is the scale of differences in the performance of these languages. In order to determine the performance of compiled and scripting languages, a comparison of the languages was made using the following algorithms: the algorithm for solving the Hanoi tower problem, the Huffman encoding algorithm and the algorithm for converting numbers into text. Each of the listed algorithms was implemented in both languages. Then the execution time of the programs was measured and the results were obtained to determine the extent of the differences in their execution speed. C language applications executed 6 to 188 times faster than Python language applications.

Keywords: performance; algorithms; C; Python

Streszczenie
Artykuł dotyczy analizy porównawczej szybkości wykonywania kodu przez język C oraz Python. Jej podstawowym celem nie było szukanie prostej odpowiedzi na pytanie, który z języków będzie wydajniejszy, tylko jaka jest skala różnic w wydajności tych języków. W celu określenia wydajności języka kompilowanego oraz skryptowego dokonano zestawienia języków na przykładzie następujących algorytmów: algorytm rozwiązujący problem wieży Hanoi, algorytm kodowania Huffmana oraz algorytm zamiany liczb na tekst. Każdy z wymienionych algorytmów został zaimplementowany w obydwu językach. Następnie dokonano pomiaru czasu realizacji programów, którego wyniki pozwoliły na określenie skali różnic w szybkości ich wykonania. W języku C aplikacje wykonywały się od 6 do 188 razy szybciej niż aplikacje w języku Python.

Słowa kluczowe: wydajność; algorytmy; język C; Python

*Corresponding author
Email address: pavelrysak@gmail.com (P. Rysak)

1. Wstęp

Ze względu na różnorodność języków programowania, wybór odpowiedniego języka do osiągnięcia zamierzonych celów jest bardzo ważny, szczególnie w dzisiejszych czasach, gdy rozmiar i poziom skomplikowania aplikacji oraz systemów ciągle rosną. Przy wytypowaniu odpowiedniego języka do stworzenia aplikacji priorytety są zróżnicowane. Może to być łatwość rozszerzalności i utrzymania kodu lub właśnie wysoka wydajność i niskie zużycie zasobów. W przypadku chęci uzyskania kodu, który będzie łatwy w utrzymaniu i rozszerzaniu, powinno się wziąć pod uwagę języki z czytelnym i klarownym kodem np. Python. Jeżeli ważna jest wydajność i niskie zużycie zasobów, pod uwagę powinno wziąć się języki kompilowane np. C.

Główną różnicą pomiędzy językami C oraz Python jest to, że C jest językiem kompilowanym, a Python językiem interpretowanym. Program napisany w języku kompilowanym przed uruchomieniem musi zostać przetłumaczony na język zrozumiały dla komputera, natomiast język skryptowy charakteryzuje się tym, iż w momencie uruchomienia programu kod jest interpretowany linia po linii. Takie podejście utrudnia znalezienie błędów w kodzie programu, gdyż będą one widoczne dopiero w trakcie uruchomienia programu, w przeciwieństwie do języka kompilowanego, gdzie informacje o błędach są wyświetlane jeszcze przed uruchomieniem programu.

W niniejszym artykule dokonano porównania reprezentantów tych dwóch grup języków. Celem badań było określenie skali różnic wydajności języka C oraz Python. W pracy zestawiono ze sobą czas wykonania algorytmów zaimplementowanych w ww. językach: algorytm rozwiązujący problem wieży Hanoi, algorytm kodowania Huffmana oraz algorytm zamiany liczb na tekst.

2. Przegląd literatury

Pierwszym przeanalizowanym artykułem jest praca autorstwa Farzeen Zehra, Maha Javed, Darakhshan Khan, Maria Pasha pt. *Comparative Analysis of C++ and Python in Terms of Memory and Time* [1]. Podjęto się tutaj porównania technik zarządzania pamięcią w języku C++ oraz Python. Dokonano analizy czasu oraz wykorzystania pamięci w czterech algorytmach:

algorytm szukający, algorytm sortujący, algorytm dodający nowe elementy do struktury danych oraz algorytm usuwający elementy. Algorytmy zaimplementowano w ww. językach. Autorzy doszli do wniosku, że kod zaimplementowany w języku Python potrzebuje więcej czasu na wykonanie, gdyż konwertuje on kod wiersz po wierszu do kodu maszynowego. W języku C++ kod programu w całości ulega kompilacji jeszcze przed uruchomieniem programu.

Kolejny artykuł napisany przez Lutz Prechelt pt. *An empirical comparison of seven programming languages* dotyczy porównania czasu wykonania kodu, zużycia pamięci oraz długości kodu algorytmu zaimplementowanego w siedmiu językach, w tym także C oraz Python [2]. Program ma za zadanie ładować do pamięci słownik oraz plik z numerami telefonów, a następnie konwertować numery telefonów na tekst. Badania pokazały, że Python w większości przypadków lepiej radzi sobie z obsługą złożonych zbiorów danych.

Następnym przeanalizowanym artykułem jest praca napisana przez Hailong Zhang oraz Jun Nie pt. *Program Performance Test based on Different Computing Environment* [3]. W tej pracy przeprowadzono pomiar czasu wykonania algorytmu obliczającego odległość sferyczną pomiędzy dwoma punktami. Przeanalizowano różne warianty kodu, m.in. Python, C oraz Cython czyli zastosowanie typów danych oraz funkcji z języka C w Python. Wyniki jednoznacznie wskazały, że programy zaimplementowane w języku Cython mogą być o wiele bardziej wydajne w porównaniu do języka Python.

W kolejnym artykule pt. *A comparison of implementations of basic evolutionary algorithm operations in different languages* [4] napisanym przez Juan-Julián Merelo-Guervós i inni, zdecydowano się na zmierzenie prędkości wykonywania operacji algorytmów ewolucyjnych, zaimplementowanych w różnych językach, m.in. w C oraz w Python. Wybrano typowe dla algorytmów ewolucyjnych operacje: mutacja Bitflip, krzyżowanie oraz operacja OneMax. Polegają one odpowiednio na zmianie wartości konkretnego bitu w łańcuchu, zmianie wartości bitów pomiędzy łańcuchami oraz na zapętleniu łańcuchów i zliczaniu wystąpień wartości 1. Autorzy doszli do wniosku, że w większości przypadków algorytmy szybciej wykonywały się w języku C. Wyjątkiem była tu operacja krzyżowania.

Ostatnią pracą w przeglądzie literatury jest artykuł pt. *Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers* [5] napisany przez Valeriu Manuel Ionescu oraz Florentina Magda Enescu. Dotyczy on wydajności języka C oraz MicroPython w odniesieniu do dość popularnych mikrokontrolerów: STM32 oraz ESP32. Wykorzystano tutaj algorytm szyfrowania SHA-256 oraz algorytm wyznaczania sum kontrolnych CRC-32. Wyniki otrzymane przez autorów jednoznacznie stwierdzają, że wydajność MicroPython jest zdecydowanie mniejsza niż języka C. Na korzyść języka MicroPython przemawia łatwość implementacji algorytmów na mikrokontrolerach, bez konieczności wprowadzania specjalnych zmian w kodzie.

3. Opis środowiska

Testy przeprowadzono na komputerze z zainstalowanym systemem operacyjnym Windows 11, z procesorem Intel Core i7-8750h o bazowej częstotliwości 2,2 GHz zaś maksymalnej 4,1 GHz oraz pamięcią RAM o pojemności 16 GB.

Python został poddany badaniu w wersji 3.10.4, a C w wersji C17. W obydwu językach zastosowano optymalizację. Jako kompilator języka C użyto MSVC w wersji 14.32.

4. Metody badawcze

Podjęto się zaimplementowania algorytmu rozwiązującego problem wieży Hanoi, algorytmu kodowania Huffmana oraz algorytmu zamiany liczb na tekst w językach C oraz Python. Wartościami mierzalnymi był czas wykonania algorytmów. W przypadku języka C stworzony uprzednio kod programu został skompilowany w trybie „release" do pliku exe. Następnie w czasie testów był uruchamiany bez skorzystania ze środowiska programistycznego przy użyciu wyłącznie konsoli wiersza poleceń systemu Windows. Kod napisany w Pythonie również był uruchamiany w ten sam sposób. Takie podejście zapewniło zminimalizowanie błędów pomiarowych spowodowanych przez środowisko programistyczne poprzez chwilowe „zawieszenie się". Każdy z algorytmów uruchomiono 30 razy.

Pomiaru czasu wykonania w języku C dokonano przy pomocy biblioteki time, deklarując zmienne typu całkowitego clock_t w odpowiednich miejscach programu i przypisując do nich wartość zwracaną przez funkcję clock [6]. Następnie odjęto od siebie te wartości i podzielono przez ilość cykli wykonywanych przez procesor w ciągu sekundy (ang. „ticks") [7]. Użycie metody w kodzie przedstawione jest na Listingu 1. Dokładność tej metody wynosi 0,001 s. Wynika to z wartości przechowywanej w wyrażeniu makro CLOCKS_PER_SEC, która w obecnej konfiguracji środowiska wyniosła 1000.

Listing 1: Przykład wykorzystania metod pomiaru czasu w języku C

```c
#include <stdio.h>
#include <time.h>

int main() {
    double time_spent = 0.0;
    clock_t begin = clock();

    for (int i = 0; i < 1000; i++)
    {
        printf("%d \n", i);
    }

    clock_t end = clock();
    time_spent += (double)(end - begin)
                  / CLOCKS_PER_SEC;
    printf("Time is %f  seconds ",
           time_spent);
    return 0;
}
```

W języku Python, pomiaru czasu wykonania programu dokonano przy użyciu biblioteki time, w podobny sposób co w języku C, odejmując od siebie wartości zwrócone przez funkcję time [8]. Dokładność tej metody jest różna dla każdego komputera i zależy od ilości cykli wykonywanych przez procesor w ciągu sekundy. Na komputerze, na którym przeprowadzano badania otrzymano dokładność 10^{-17} s. Użycie metody przedstawiono na Listingu 2.

Listing 2: Przykład wykorzystania metod pomiaru czasu w języku Python

```python
import time

if __name__ == '__main__':
    start_time = time.time()

    for x in range(1000):
        print(x)

    print("--- %s ---" % (time.time()
                          - start_time))
```

5. Algorytmy

5.1. Algorytm rozwiązujący problem wieży Hanoi

Rozwiązanie problemu wieży Hanoi polega na przeniesieniu wieży zbudowanej z krążków o różnych średnicach ze słupka A na słupek C posługując się słupkiem B [9]. Przy wykonywaniu tego zadania należy przenosić tylko jeden krążek jednocześnie oraz trzeba pamiętać, aby zawsze mniejszy krążek leżał na większym.

W przypadku implementacji tego algorytmu zastosowano 25 krążków. Została użyta funkcja rekurencyjna, w której liczba krążków (n) ulega stopniowemu zmniejszaniu.

Listing 3: Kod algorytmu rozwiązującego problem wieży Hanoi w języku Python

```python
import time

def Hanoi(n, _from, _to, _aux):
    if n == 1:
        move(n, _from, _to)
        return
    Hanoi(n-1, _from, _aux, _to)
    move(n, _from, _to)
    Hanoi(n-1, _aux, _to, _from)


def move(n, _from, _to):
    global step
    step = step + 1


if __name__ == '__main__':
    start_time = time.time()
    step = 0
    n = 25

    Hanoi(n, 'A', 'C', 'B')
    print("--- %s ---" % (time.time()
                          - start_time))
```

Kod algorytmu rozwiązującego problem wieży Hanoi [10] w języku Python został zaprezentowany na Listingu 3. Dodatkowo umieszczono w tym programie metodę pomiaru czasu wykonania programu, czego dokonano przed wywołaniem właściwej funkcji o nazwie Hanoi oraz zaraz za nią.

Kod algorytmu [10] w języku C został przedstawiony na Listingu 4, gdzie także umieszczono użycie metody mierzącej czas wykonania programu. Dokonano tego bezpośrednio przed oraz za wywołaniem właściwej funkcji o nazwie Hanoi. Taki schemat pomiaru czasu został zastosowany do pozostałych algorytmów.

Listing 4: Kod algorytmu rozwiązującego problem wieży Hanoi w języku C

```c
#include <stdio.h>
#include <time.h>

void hanoi(int n, char _from,
           char _to, char _aux);
void move(int n, char _from, char _to);

void hanoi(int n, char _from,
           char _to, char _aux) {
    if (n == 1) {
        move(n, _from, _to);
        return;
    }
    hanoi(n - 1, _from, _aux, _to);
    move(n, _from, _to);
    hanoi(n - 1, _aux, _to, _from);
}

int step = 0;
void move(int n, char _from, char _to)
           { ++step; }

int main() {
    double time_spent = 0.0;
    int n = 25;
    clock_t begin = clock();
    hanoi(n, 'A', 'C', 'B');
    clock_t end = clock();
    time_spent += (double)(end - begin)
                  / CLOCKS_PER_SEC;
    printf("Time is %f  seconds ",
           time_spent);
    return 0;
}
```

5.2. Algorytm kodowania Huffmana

W informatyce algorytmy kompresujące mają za zadanie zmniejszać rozmiar danych. Spośród metod kompresji danych możemy wyróżnić dwa rodzaje: kompresja stratna oraz kompresja bezstratna. W algorytmie kompresji stratnej niektóre fragmenty danych są tracone. W algorytmach kompresji bezstratnej dane sprzed kompresji są identyczne jak po kompresji. Do tej właśnie grupy należy algorytm kodowania Huffmana.

Algorytm [11] jako dane wejściowe przyjmuje tekst, w którym zliczane są wystąpienia poszczególnych liter. Następnie ze zbioru liter zawierających przyporządko-

wane im liczby wystąpień, dwie litery o najmniejszej liczbie wystąpień są łączone w nowy liść w drzewie bianrnym, którego wartość będzie odpowiadać sumie wystąpień wcześniej wspomnianych liter. W kolejnym kroku brana jest litera o najmniejszej wartości wystąpień, z pominięciem tych, które zostały już wykorzystane, a następnie znowu tworzony jest nowy liść mający wartość odpowiadającą sumie wartości wystąpień dobranej litery i utworzonego wcześniej liścia. Kroki te są powtarzane aż do momentu, kiedy pozostanie jeden element. W ten sposób powstaje drzewo binarne, w którym skrajne elementy (liście) mają przypisaną wartość zero, jeśli znajdują się po lewej stronie lub wartość 1, jeśli znajdują się po prawej stronie. Kody poszczególnych liter są otrzymywane w wyniku przypisywania odpowiadających im wartości binarnych idąc od góry drzewa.

Zaimplementowany algorytm ma za zadanie przyjąć tekst o dowolnej długości, a następnie wyznaczyć kody poszczególnych liter. Program uruchomiono dla tekstu wejściowego, anglojęzycznego zawierającego 1074000 słów, czyli 6158000 znaków.

Na Listingu 5 dotyczącym fragmentu kodu w Python [12,13] przedstawiono użycie biblioteki heapq, dostarczającej metody pozwalające łatwiej zaimplementować algorytm. Dostarcza ona funkcje umożliwiające utworzenie stosu (heapify) i wstawienie do niego posortowanych elementów (heappush). Kod ze względu na te usprawnienia stał się znacznie krótszy od kodu w języku C.

Listing 5: Fragment kodu algorytmu kodowania Huffmana w języku Python

```
def buildHuffmanTree(text):
    if len(text) == 0:
        return

    freq = {}
    for c in text:
        freq[c] = freq.get(c, 0) + 1

    pq = [Node(k, v) for k, v in
          freq.items()]
    heapq.heapify(pq)

    while len(pq) != 1:

        left = heappop(pq)
        right = heappop(pq)

        total = left.freq + right.freq
        heappush(pq, Node(None, total,
                          left, right))

    root = pq[0]

    huffmanCode = {}
    encode(root, '', huffmanCode)
```

Listing 6 przedstawia funkcję sortującą elementy przed wstawieniem ich do drzewa (minHeapify) oraz funkcję zamieniającą węzły (swapNode) zaimplementowaną w języku C [12,13]. W tym przypadku wszystkie funkcje napisano bez użycia gotowych bibliotek. W związku z tym kod algorytmu stał się bardziej zrozumiały, lecz w konsekwencji uzyskano dość sporą liczbę linii kodu, co może przełożyć się na ogólną wydajność algorytmu.

Listing 6: Fragment kodu algorytmu kodowania Huffmana w języku C

```
void minHeapify(struct MinHeap* minHeap,
                int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->freq
        < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->freq
        < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapNode(&minHeap->array[smallest]
            , &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

void swapNode(struct Node** a,
              struct Node** b) {
    struct Node* t = *a;
    *a = *b;
    *b = t;
}
```

5.3. Algorytm zamiany liczb na tekst

Istota algorytmu polega na wczytaniu pliku tekstowego, w którym zapisane są losowo wygenerowane, 20-cyfrowe numery oraz wzorzec dekodowania. Fragment pliku wejściowego został przedstawiony na listingu 7. Program korzystając ze wzorca wyświetla w wyniku tekst. Uruchomiono go dla 10000 wierszy.

Listing 7: Fragment pliku wejściowego

```
1 78048452524596100000:13=Z;17=Y;23=
  X;21=W;35=V;40=U;42=T;48=S;56=R;57
  =Q;65=P;70=O;75=N;79=M;84=L;85=K;9
  6=J;0=X;1=A;2=B;3=C;4=D;5=E;6=F;7=
  G;8=H;9=I
2 53601617947958700000:13=Z;17=Y;23=
  X;21=W;35=V;40=U;42=T;48=S;56=R;57
  =Q;65=P;70=O;75=N;79=M;84=L;85=K;9
  6=J;0=X;1=A;2=B;3=C;4=D;5=E;6=F;7=
  G;8=H;9=I
3 79669760798126900000:13=Z;17=Y;23=
  X;21=W;35=V;40=U;42=T;48=S;56=R;57
  =Q;65=P;70=O;75=N;79=M;84=L;85=K;9
  6=J;0=X;1=A;2=B;3=C;4=D;5=E;6=F;7=
  G;8=H;9=I
4 37117332951862000000:13=Z;17=Y;23=
  X;21=W;35=V;40=U;42=T;48=S;56=R;57
  =Q;65=P;70=O;75=N;79=M;84=L;85=K;9
  6=J;0=X;1=A;2=B;3=C;4=D;5=E;6=F;7=
  G;8=H;9=I
```

Kod algorytmu zaimplementowanego w języku C przestawiono na listingu 8. Kod okazał się znacznie dłuższy niż w języku Python. Najpierw została zliczona liczba wystąpień podciągu w słowie wejściowym, następnie przy użyciu obliczonej wartości została alokowana określona ilość pamięci na wynik. Takie podejście zapewnia zarezerwowanie optymalnej ilości pamięci w celu jak najefektywniejszego działania programu.

Listing 8: Fragment kodu algorytmu zamiany liczb na tekst w języku C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <windows.h>

char* replaceWord(const char* source,
                  const char* oldWord,
                   const char* newWord);
char* rtrim(char* s);

char* replaceWord(const char* source,
                  const char* oldWord,
                   const char* newWord) {
    char* result;
    int i, cnt = 0;
    int newWordlen = strlen(newWord);
    int oldWordlen = strlen(oldWord);

    for (i = 0; source[i] != '\0'; ++i) {
        if (strstr(&source[i], oldWord)
            == &source[i]) {
            ++cnt;
            i += oldWordlen - 1;
        }
    }

    result = (char*)malloc(i + cnt *
             (newWordlen - oldWordlen)
              + 1);
```

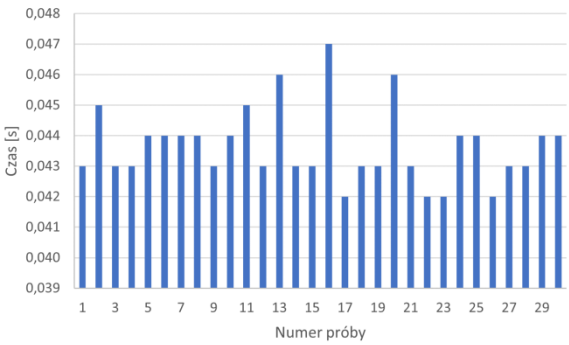
Kod algorytmu zaimplementowanego w języku Python przestawiono na Listingu 9. W porównaniu z kodem programu w C kod okazał się bardzo krótki. Jest to spowodowane tym, że Python posiada wiele gotowych funkcji, z których można skorzystać. W tym przypadku skorzystano z funkcji replace, która umożliwiła zamianę liczby na tekst. Funkcja jest wywoływana na łańcuchu znaków. Przyjmuje ona w parametrach ciąg znaków, który ma być zamieniony oraz nowy ciąg, który zastąpi poprzedni. Dodatkowym opcjonalnym parametrem jest liczba określająca, jak dużo wystąpień podciągu w słowie ma być zamienione. W opisywanym algorytmie parametr tej funkcji został pominięty, co oznacza, że replace będzie zamieniać wszystkie napotkane podciągi. W algorytmie użyto również funkcji split, która, umożliwia podzielenie napisu. Funkcja przyjmując w parametrach określony znak oraz liczbę oznaczającą ilość podzieleń tekstu, dzieli tekst w miejscach, w których napotka na wcześniej zdefiniowany znak, pod warunkiem, że nie przekroczy ona określonej liczby podziałów tekstu. W omawianym algorytmie jako parametr funkcji podano jedynie pierwszy z nich.

Listing 9: Kod algorytmu zamiany liczb na tekst w języku Python

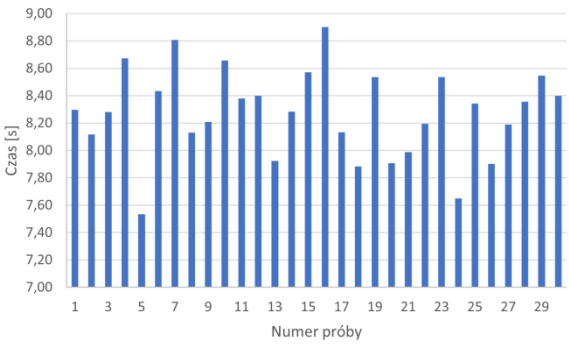
```python
import time

if __name__ == '__main__':
    start_time = time.time()

    file = open("input.txt", "r")

    for line in file.readlines():

        inputLine = line.strip()

        split = inputLine.split(":")
        answer = split[0]
        for rule in split[1].split(';'):

            ruleSplit = rule.split('=')

            answer = answer.replace
            (ruleSplit[0], ruleSplit[1])
```

6. Wyniki badań

W tym rozdziale zaprezentowano uzyskane wyniki badań w postaci wykresów. Wartości przedstawione na wykresach dotyczą każdej z 30 podjętych prób uruchomienia każdego algorytmu.



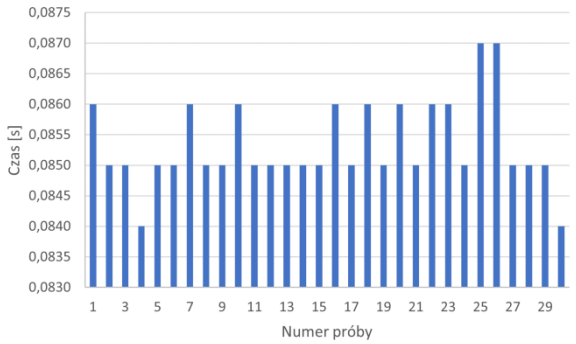
Rysunek 1: Czas wykonania algorytmu rozwiązującego problem wieży Hanoi w języku C.



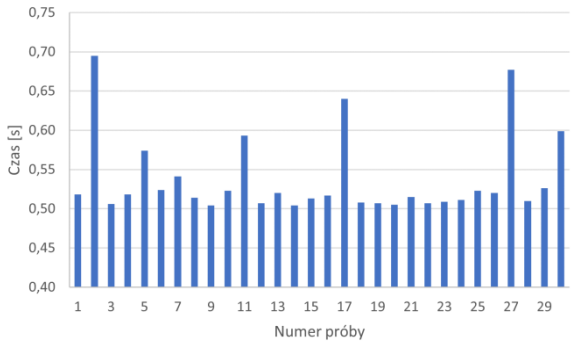
Rysunek 2: Czas wykonania algorytmu rozwiązującego problem wieży Hanoi w języku Python.

Jak widać na Rysunku 1 znaczna część uzyskanych wartości czasu oscyluje w granicach 0,043 s. Górna granica uzyskanych czasów wynosi 0,047 s, zaś dolna

osiąga 0,042 s. Na podstawie Rysunku 2, przedstawiającego czasy uzyskane przez algorytm wieża Hanoi w języku Python nie można jednoznacznie stwierdzić w okolicy jakiej wartości oscyluje znaczna część czasów. Wynika to ze zbyt dużej rozbieżności pomiędzy uzyskanymi wynikami. Odchylenie standardowe wyników języka C oraz Python wynosi odpowiednio 0,001 s i 0,317 s. W przypadku języka C średni czas wykonania algorytmu wyniósł 0,044 s, jeśli chodzi o Python uzyskano średnią wynoszącą 8,271 s. Wyniki te pokazują, że język C okazał się w tym przypadku szybszy blisko 188 razy.



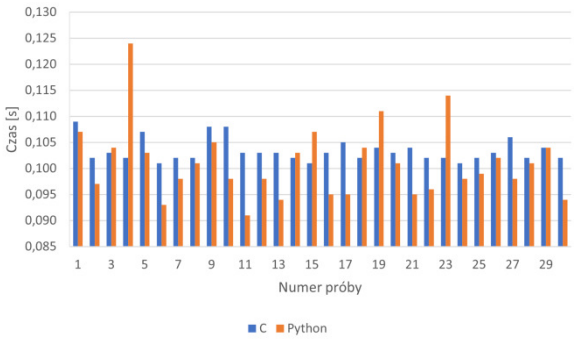
Rysunek 3: Czas wykonania algorytmu kodowania Huffmana w języku C.



Rysunek 4: Czas wykonania algorytmu kodowania Huffmana w języku Python.

Analizując wykres przedstawiony na Rysunku 3 można stwierdzić, że większość wartości oscyluje w granicach 0,085 s. Najdłuższy czas wynosi 0,087 s, a najkrótszy 0,084 s. W przypadku tego samego algorytmu zaimplementowanego w języku Python (patrz Rysunek 4), widać, że czas wykonania większości prób waha się od 0,50 s do 0,55 s. Odchylenie standardowe w przypadku języka C wynosi 0,0007 s, a w przypadku języka Python sięga 0,0500 s. Różnica między tymi wartościami nie jest już tak duża, jak w przypadku algorytmu rozwiązującego problem wieży Hanoi. Średnia wartość czasu wykonania algorytmu w języku C wyniosła 0,085 s, zaś w przypadku języka Python 0,537 s. Otrzymane średnie pozwalają stwierdzić, że język C okazał się szybszy około 6 razy.

Wyniki przedstawione na powyższych rysunkach jednoznacznie dowodzą uzyskania większej wydajności przez język C, pomimo różnej złożoności obliczeniowej algorytmu rozwiązującego problem wieży Hanoi oraz algorytmu kodowania Huffmana. W pierwszym z ww. algorytmów różnice w czasie wykonania są zdecydowanie większe niż w przypadku drugiego algorytmu. Wynika to z tego, że algorytm rozwiązujący problem wieży Hanoi jest zbudowany z wykorzystaniem funkcji rekurencyjnej, która jest często mniej optymalna niż funkcja wykorzystująca pętle.



Rysunek 5: Czas wykonania algorytmu zamiany liczb na tekst w języku C oraz Python.

W przypadku algorytmu zamiana liczb na tekst, wyniki zdecydowanie się różnią od wcześniej opisywanych. Na powyższym Rysunku 5 widać wyraźnie zbliżoną wydajność tych języków. Wyniki dotyczące czasu uzyskanego przez Python są nieco bardziej zróżnicowane. Różnica między najkrótszym a najdłuższym czasem wynosi aż 0,033 s. Jest ona zdecydowanie mniejsza niż w przypadku języka C, gdzie wynosi ona 0,008 s. Odchylenie standardowe wyników jest znowu większe dla algorytmu w języku Python, które wynosi 0,007 s, podczas gdy w języku C osiąga ono 0,002 s. Równocześnie różnica między odchyleniami standardowymi jest najmniejsza i wynosi zaledwie 0,005 s. Wyniki są zbliżone do tego stopnia, że na podstawie wykresu nie można jednoznacznie wskazać wydajniejszego języka programowania. Stwierdzić to można na podstawie porównania średnich wartości czasów, które wynoszą dla języka C 0,103 s oraz dla języka Python 0,101 s.

7. Wnioski

Biorąc pod uwagę specyfikę algorytmów zaimplementowanych w obydwu językach oraz uzyskane wyniki, można stwierdzić, że C jest językiem szybszym od 6 do 188 razy od języka Python. Jak widać skala różnic pomiędzy czasem wykonania programów z zaimplementowanym algorytmem rozwiązującym problem wieży Hanoi oraz kodowania Huffmana jest bardzo duża. Kiedy złożoność obliczeniowa algorytmu wzrasta, tak jak w przypadku algorytmu rozwiązującego problem wieży Hanoi, różnice w czasie są jeszcze bardziej widoczne. Jedyny przypadek, w którym Python uzyskał nieco lepszy czas to algorytm zamiany liczb na tekst. Może to wynikać z faktu, że Python lepiej sobie radzi z przetwarzaniem złożonych zbiorów danych. Na korzyść języka Python przemawia mniejsza liczba linii kodu co wyraźnie widać na zamieszczonych listingach W celu uzyskania lepszych czasów wykonania programów dla tego języka, można zastosować język zawierający elementy składni Python i C, czyli Cython. Jest on

o wiele bardziej wydajny od tradycyjnego języka Python.

Problem, którego rozwiązania podjęto się w tej pracy jest bardzo szeroki i podejmowany przez wielu badaczy. Niniejszy artykuł jest tylko małym dodatkiem, który być może chociaż w małym stopniu przyczyni się pośrednio do udoskonalania programów użytkowych.

Literatura

[1] F. Zehra, M. Javed, D. Khan, M. Pasha, Comparative Analysis of C++ and Python in Terms of Memory and Time, Preprints (2020), https://doi.org/10.20944/preprints202012.0516.v1.

[2] L. Prechelt, An empirical comparison of seven programming languages, Computer 33 (10) (2000) 23-29, https://doi.org/10.1109/2.876288.

[3] H. Zhang, J. Nie, Program Performance Test based on Different Computing Environment, 2016 IEEE International Conference of Online Analysis and Computing Science (ICOACS) (2016) 174-177.

[4] J. -J. Merelo-Guervós et al., A comparison of implementations of basic evolutionary algorithm operations in different languages, 2016 IEEE Congress on Evolutionary Computation (CEC) (2016) 1602-1609.

[5] V. M. Ionescu, F. M. Enescu, Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers, 2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME) (2020) 234-237.

[6] Sposób pomiaru czasu wykonania programu w języku C, https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9 , [01.02.2022].

[7] Opis wyrażenia makro CLOCKS_PER_SEC, https://www.educative.io/answers/what-is-clockspersec-in-c , [25.09.2022].

[8] Sposób pomiaru czasu wykonania programu w języku Python, https://pynative.com/python-get-execution-time-of-program/ , [01.02.2022].

[9] Opis problemu dotyczącego algorytmu rozwiązującego problem wieży Hanoi, https://pl.wikipedia.org/wiki/Wie%C5%BCe_Hanoi , [01.02.2022].

[10] Kod algorytmu rozwiązującego problem wieży Hanoi w języku C oraz Python, https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/ , [03.04.2022].

[11] Opis przebiegu algorytmu kodowania Huffmana, https://binarnie.pl/kodowanie-huffmana/ , [01.02.2022].

[12] Kod algorytmu kodowania Huffmana w języku C oraz Python, https://www.programiz.com/dsa/huffman-coding, [03.04.2022].

[13] Kod algorytmu kodowania Huffmana w języku C oraz Python, https://www.techiedelight.com/huffman-coding/ , [03.04.2022].