SPRAWOZDANIE

PIOTR ROGOWSKI 192258

1. Cel projektu

Celem projektu jest zaimplementowanie dowolnej funkcji zawartej w bibliotece standardowej języka Python. Funkcja, którą wybrałem to *str.strip()*. Jako argument przyjmuje jedną, opcjonalną wartość w formie ciągu znaków, które zostaną usunięte z początku oraz końca wskazanego ciągu znaków. Przykład użycia funkcji: „piotr”.strip(„pro”) = „iot”. W przypadku niepodania argumentu funkcja usuwa białe znaki z początku i końca podanego ciągu znaków.

2. Implementacja funkcji

2.1 Język C++

```
string function(string = "", string = "\t\n\x0b\x0c\r\x1c\x1d\x1e\x1f ");
```

Rys 1. Fragment pliku nagłówkowego zawierający deklarację funkcji m.in. wartość opcjonalnego argumentu

```
string function(string characters, string exclude_characters) {
    int cut_head = 0, cut_tail = 0;

    for (int index = 0; index < characters.size(); index++) {
        bool found_in_excluded = false;
        for (int i = 0; i < exclude_characters.size(); i++) {
            if (characters[index] == exclude_characters[i]) {
                cut_head++;
                found_in_excluded = true;
                break;
            }
        }
        if (!found_in_excluded) break;
    }

    for (int index = characters.size() - 1; index >= 0; index--) {
        bool found_in_excluded = false;
        for (int i = 0; i < exclude_characters.size(); i++) {
            if (characters[index] == exclude_characters[i]) {
                cut_tail++;
                found_in_excluded = true;
                break;
            }
        }
        if (!found_in_excluded) break;
    }

    return characters.substr(cut_head, characters.size() - cut_tail - cut_head);
}
```

Rys 2. Kod źródłowy własnej implementacji funkcji w języku C++

2.2 Język Python

```python
def function(characters='', exclude_characters=None):
    exclude_characters = [ ord(char) for char in exclude_characters ] if \
        exclude_characters else [9, 10, 11, 12, 13, 28, 29, 30, 31, 32]
    cut_head = 0
    cut_tail = 0

    for index in range(len(characters)):
        if ord(characters[index]) in exclude_characters:
            cut_head += 1
        else: break

    for index in range(len(characters) - 1, -1, -1):
        if ord(characters[index]) in exclude_characters:
            cut_tail += 1
        else: break

    return characters[cut_head : len(characters) - cut_tail]
```

Rys 3. Kod źródłowy własnej implementacji funkcji w języku Python

3. Wygenerowane dane

Na potrzeby projektu wygenerowano 2000 unikalnych danych. Format zapisu danych to wartości kodów z przedziału [0; 127], który przyjęto ze względu na sposób kodowania znaków przez typ *string* w języku C++. Typ *string* traktuje pojedynczą wartość znaku jako *signed char*, przez co jest on w stanie przyjąć wartości z przedziału [-128, 127]. W instrukcji zawarta jest informacja, aby przyjąć założenie w formie znaków z przedziału [0, 255], co jest równoznaczne typem *unsigned char*. Jednak przy wpisaniu typu *unsigned char* do typu *string* staje on się typem char ze znakiem. Opisane powyżej uproszczenie zostało skonsultowane z prowadzącym oraz zaakceptowane. Połowa wygenerowanych danych zawiera opcjonalny parametr. Pojedynczy generowany przypadek został wygenerowany w dwóch liniach pliku tekstowego. W linii nieparzystej zapisane są kody znaków, dla których wywoływana będzie funkcja. W liniach parzystych zapisane zostały kody znaków podawane jako opcjonalny parametr. Przypadki testowe niezawierające opcjonalnego parametru posiadają linię parzystą pustą. Poniżej przedstawiono fragment pliku z wygenerowanymi danymi.

dataset.txt

```
1   113 8 101 70 17 31 77 62 126 72
2   113 8 126 72
3   113 18 107 52 47 91 112 74 99 76
4   113 18 99 76
```

Rys 4. Fragment pliku z danymi

```python
if __name__ == '__main__':
    args = parse_args()

    previous_characters = []
    previous_excluded = []

    for i in range(args.size):
        ascii_characters = None
        ascii_excluded = None

        while True:
            characters = generate_characters(args.characters_size)
            ascii_characters = convert_to_ascii(characters)

            excluded = generate_excluded_characters(characters, args.excluded_size) if args.excluded_size else ''
            ascii_excluded = convert_to_ascii(excluded) if args.excluded_size else ''

            if not ascii_characters in previous_characters and \
                (not ascii_excluded in previous_excluded or not args.excluded_size):
                previous_characters.append(ascii_characters)
                previous_excluded.append(ascii_excluded)
                break

        with open(args.output_path, 'a', encoding='utf8') as output:
            output.write(ascii_characters + '\n')
            output.write(ascii_excluded + '\n')
```

Rys 5. Fragment kodu umożliwiający generowanie danych testowych

4. Mierzenie czasu

4.1 Funkcje

W implementacji funkcji napisanej w języku Python do pomiaru czasu użyta została funkcja *time.time_ns()*. Funkcja zwraca czas, który upłynął w nanosekundach. Jako dokładność pomiaru czasu przez funkcję *time.time_ns()* przyjęto wartość 1 sekundy.

W implementacji funkcji napisanej w języku C++ do pomiaru czasu użyta została funkcja *now()* będąca częścią klasy *steady_clock* biblioteki *chrono*. Funkcja zwraca czas, który upłynął w dowolnej jednostce czasu dzięki rzutowaniu na jednostkę czasu. Jako dokładność pomiaru czasu przez funkcję również przyjęto wielkości 1 sekundy.

4.2 Sposób

Pomiar pojedynczego wywołania funkcji jest mniejszy niż założony błąd pomiaru, przez co nie jest to wiarygodne. Istnieje zatem potrzeba mierzenia czasu wykonania większej ilości wywołań funkcji. Ten sposób jednak wiąże się z dodatkowym czasem, jaki kompilator oraz interpreter muszą poświęcić na iterację po kolejnych wywołaniach funkcji. Aby policzyć tylko czas czystych obliczeń od czasu obliczeń w pętli potrzeba odjąć czas trwania pustej pętli. Zatem czas mierzony zostanie 3 razy: przed wywołaniem pustej pętli, po wywołaniu pustej pętli oraz po wywołaniu pętli z obliczeniami. Różnice pomiędzy poszczególnymi czasami są kolejno równe czasom trwania pustej pętli oraz pętli z obliczeniami. W celu uzyskania czasu samych obliczeń wystarczy obliczyć różnicę pomiędzy tymi wartościami.

W instrukcji wspomniany został także błąd względny, który ma mieć wartość nieprzekraczającą 1%. Zatem mając do wykonania 3 pomiary czasu (każdy z błędem pomiaru w wysokości 1 sekundy) przeprowadzone eksperymenty powinny trwać minimum 300 sekund, aby błąd względny był mniejszy niż 1%.

```cpp
auto t0 = chrono::steady_clock::now();

for (int r = 0; r < REPETITIONS; r++) {
    for (int dataset_index = 0; dataset_index < DATASET_SIZE; dataset_index++) {
        if (excluded_array[dataset_index].size()) { } else { }
    }
}

auto t1 = chrono::steady_clock::now();

for (int r = 0; r < REPETITIONS; r++) {
    for (int dataset_index = 0; dataset_index < DATASET_SIZE; dataset_index++) {
        if (excluded_array[dataset_index].size()){
           function(characters_array[dataset_index], excluded_array[dataset_index]);
        } else {
            function(characters_array[dataset_index]);
        }
    }
}
auto t2 = chrono::steady_clock::now();

int empty_loop = chrono::duration_cast<chrono::seconds>(t1 - t0).count();
int loop = chrono::duration_cast<chrono::seconds>(t2 - t1 ).count();
printf("%d [s] -> empty loop\n", empty_loop);
printf("%d [s] -> empty loop + calculations\n", loop);
printf("%d [s] -> pure calculations \n", loop - empty_loop);
```

Rys 6. Fragment kodu w języku C++ przedstawiający sposób pomiaru czasu

5. Test wydajności

Test wydajności funkcji str.strip():

```
PS C:\Users\rogus\Desktop\Jezyki Skryptowe Projekt1> python3 .\python\orginal_performance_test.py
184.3430658 [s] -> empty loop
528.2878797 [s] -> empty loop + calculations
343.9448139 [s] -> pure calculations
```

Rys 7. Wynik wywołania testu wydajności funkcji str.strip()

Implementacja własnej funkcji w Pythonie:

```
PS C:\Users\rogus\Desktop\Jezyki Skryptowe Projekt1> python3 .\python\custom_performance_test.py
26.1265592 [s] -> empty loop
602.0070054 [s] -> empty loop + calculations
575.8804462 [s] -> pure calculations
```

Rys 8. Wynik wywołania testu wydajności autorskiej funkcji w języku Python

Implementacja własnej funkcji w C++:

```
piotr@DESKTOP-L56EBAQ MINGW64 ~/Desktop/Jezyki Skryptowe Projekt1
$ ./scripts/run_cpp_tests.sh
1 [s] -> empty loop
436 [s] -> empty loop + calculations
435 [s] -> pure calculations
```

Rys 9. Wynik wywołania testu wydajności autorskiej funkcji w języku C++

Podsumowanie testów wydajności dla wszystkich implementacji:

Implementacja	Czas [s]	Błąd względny [%]	Ilość powtórzeń (n)	Średni czas jednego wywołania funkcji [ns]
str.strip()	344	0,872093023	1 500 000	114,7
Python	576	0,520833333	200 000	1440
C++	435	0,689655172	300 000	725

Tabela 1. Tabela wyników pomiaru wydajności oryginalnej funkcji z autorskimi w językach Python i C++

Najszybsza w działaniu okazała się funkcja biblioteki standardowej *str.strip()*. Kolejną najszybszą jest implementacja własna funkcji w języku C++. Dwa razy wolniejszą od implementacji w języku C++ okazała się implementacja własna w języku Python.

6. Test implementacji

W celu walidacji zwracanych przez wszystkie implementacje wyników zapisywane są one do plików tekstowych w formacie ciągów znaków. Napisany został odpowiedni skrypt w języku Python, który pozwala na wczytanie zawartości plików oraz ich porównanie.

result_comparator.py > ...

```python
RESULT_PATH_CPP = 'result_cpp.txt'
RESULT_PATH_PYTHON_CUSTOM = 'result_python_custom.txt'
RESULT_PATH_PYTHON_ORGINAL = 'result_python_orginal.txt'

result_cpp = []
result_python_custom = []
result_python_orginal = []

with open(RESULT_PATH_CPP) as file:
    result_cpp = file.readlines()

with open(RESULT_PATH_PYTHON_CUSTOM) as file:
    result_python_custom = file.readlines()

with open(RESULT_PATH_PYTHON_ORGINAL) as file:
    result_python_orginal = file.readlines()

for line_number in range(len(result_cpp)):
    if result_cpp[line_number] != result_python_custom[line_number] \
        or result_cpp[line_number] != result_python_orginal[line_number] \
        or result_python_custom[line_number] != result_python_orginal[line_number]:
        print(f'Błąd w linii: {line_number}')
        print(f'result_cpp: {result_cpp[line_number]}')
        print(f'result_python_custom: {result_python_custom[line_number]}')
        print(f'result_python_orginal: {result_python_orginal[line_number]}')
        exit(0)

print('OK')
```

Rys 10. Kod w języku Python realizujący funkcjonalność testowania implementacji

7. Podsumowanie

Najszybszą implementacją okazała się oryginalna implementacja standardowej biblioteki Pythona. Najprawdopodobniej działanie algorytmu zostało staranie przemyślane tak, aby możliwie zoptymalizować czas jego działania. Natomiast zaprojektowany przeze mnie algorytm odbiega pod kątem złożoności czasowej od oryginalnej implementacji. Różnica czasów wykonania algorytmu w języku C++ oraz Python najprawdopodobniej wynika z charakteru obu języków. Język C++ jest kompilowany do kodu maszynowego bezpośrednio wykonywanego przez procesor natomiast Python jest językiem interpretowanym co może mieć znaczny wpływ na czas wykonania obu programów. Również Python jest językiem dynamicznie typowanym co oznacza że wnioskuje on typy przechowywanych zmiennych w trakcie wykonywania programu, natomiast C++ jest językiem ze statycznym typowaniem zmiennych. Różnica ta również może mieć wpływ na wydajność implementacji w obu językach.