Optymalizacja kodu na różne architektury
Zadanie domowe nr 1

Wojciech Michaluk

13.04.2025

1 Opis zadania

Celem zadania jest implementacja funkcji, która przetwarza duże ciągi tekstowe (np. zawartość dużego pliku), wykonując szereg operacji normalizacyjnych oraz optymalizacyjnych, takich jak:

- usuwanie znaków niedrukowalnych - można przyjąć, że np. o kodach spoza zakresu 32 do 126,
- zamiana sekwencji *białych znaków* (spacje, tabulatory, nowe linie) na pojedynczą spację,
- konwersja wszystkich liter do małych liter,
- konwersja wszystkich znaków interpunkcyjnych na przecinki,
- wyeliminowanie duplikatów wyrazów występujących bezpośrednio po sobie.

1.1 Weryfikacja posiadanego modelu procesora

Upewnijmy się, jakie są parametry procesora.

Parametr procesora	**Wartość parametru**
Producent	Intel
Model	i5-8300H
Liczba rdzeni	4
Liczba procesorów logicznych (wątków)	8
Częstotliwość podstawowa	2.30 GHz
Pamięć podręczna (poziom 1)	256 kB
Pamięć podręczna (poziom 2)	1 MB
Pamięć podręczna (poziom 3)	8 MB

Tabela 1: Podstawowe parametry procesora i ich wartości

Co prawda konkretnie w tym problemie nie powinno to być aż tak istotne, bowiem zadanie dotyczy przetwarzania tekstu. Zwrócilibyśmy na to szczególną uwagę podczas np. różnego typu działań na macierzach, tam niewątpliwie możnaby sensownie użyć jednostek wektorowych. Niemniej jednak warto być świadomym, na czym się pracuje.

2 Implementacja i analiza kolejnych wersji rozwiązania

2.1 Wersja bazowa

W celu realizacji zadanych funkcjonalności, używam najpierw standardowych metod `std::string` i algorytmów języka C++. Kod nie jest zbytnio zoptymalizowany pod kątem wydajności. Poniżej zamieszczam pełen kod programu, natomiast w następnych wersjach skupię się na zmianach, jakie nastąpią.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <string>
#include <iostream>
#include <cctype>

#define REPEATS 50000

static double gtod_ref_time_sec = 0.0;

// Adapted from the bl2_clock() routine in the BLIS library
double dclock() {
    double the_time, norm_sec;
    struct timeval tv;
    gettimeofday(&tv, NULL);

    if(gtod_ref_time_sec == 0.0)
        gtod_ref_time_sec = (double)tv.tv_sec;

    norm_sec = (double)tv.tv_sec - gtod_ref_time_sec;
    the_time = norm_sec + tv.tv_usec * 1.0e-6;

    return the_time;
}

// function to be optimized
std::string process_large_string(const std::string & s) {
    std::string result, last, current;
    bool flag;

    for(int i = 0; i < s.length(); i++) {
        if(isspace(s[i])) {
            if(current != last) {
                if(current.length()) {
                    result = result + current;
                    last = current;
                }

                if(flag)
                    result = result + '_';
            }

            current.clear();
            flag = false;
        } else if(ispunct(s[i])) {
            if(current.length() && current != last) {
                result = result + current;
                last = current;
            }

            result = result + ',';
            current.clear();
```

```
            flag = true;
        } else if(s[i] >= 0x20 && s[i] <= 0x7E) {
            current = current + (char)tolower(s[i]);
            flag = true;
        }
    }

    return result;
}

int main(int argc, const char* argv[]) {
    int i, j, k, iret;
    double dtime;
    std::string s, result, line;

    std::cout << "Let's start processing the file\n";

    while(getline(std::cin, line))
        s += line + "\n";

    // we start the clock
    dtime = dclock();

    for(int i = 0; i < REPEATS; i++)
        result = process_large_string(s);

    // calculate the time taken
    dtime = dclock() - dtime;

    std::cout << result << "\n";
    std::cout << "Time: " << dtime << "\n";
    fflush(stdout);

    return iret;
}
```

Listing 1: Bazowa wersja kodu - *normalize1.cpp*

Najbardziej interesującą nas funkcją (w tym i w pozostałych wariantach) będzie `process_large_string`, która pozwala osiągnąć zamierzony cel.
UWAGA! Jako słowa przyjmuję ciągi znaków w odpowiednim przyjętym zakresie kodów, natomiast białe znaki oraz znaki interpunkcyjne nie wchodzą w skład słów, rozdzielają je.

2.1.1 Opis implementacji

Funkcja `dclock` jest funkcją pomocniczą o treści identycznej jak na laboratorium. Służy ona do mierzenia czasu działania programu.
Szczegółowy opis funkcji `process_large_string`:

- wejście do funkcji (zawartość pliku) jest przekazywane poprzez argument `s`,
- wynik zapisywany jest w zmiennej `result`. Oprócz niej używam zmiennych pomocniczych `last` i `current` (przechowują odpowiednio poprzednie i obecne słowo - są używane przy wykrywaniu duplikatów) oraz `flag` - używana do wykrywania sekwencji białych znaków,
- używam funkcji z biblioteki `cctype`: wykrywam białe znaki z użyciem `isspace`; znaki interpukcyjne z pomocą `ispunct`; z kolei konwersję do małych liter obsługuję poprzez `tolower`,

- w zależności od rodzaju znaku (1. *whitespace* / 2. interpunkcyjny / 3. inny, o kodzie z zakresu 32 do 126) podejmuję odpowiednią akcję - czyli odpowiednio:

    1. sprawdzam, czy nie wykryto duplikatu - jeżeli nie, to sprawdzam, czy obecne słowo (`current`) nie jest puste, wtedy dopisuję je do wyniku i aktualizuję ostatnie słowo tym wyrazem; następnie sprawdzam, czy nie wystąpiła sekwencja białych znaków (czyli czy flaga jest ustawiona) - jeżeli nie, to dodaję spację do wyniku; poza tym resetuję flagę i obecne słowo,
    2. sprawdzam, czy nie ma duplikatu słowa oraz czy obecne słowo nie jest puste - w takim razie do wyniku dopisuję obecne słowo i aktualizuję ostatnie słowo; dopisuję przecinek do wyniku, resetuję obecne słowo i ustawiam flagę,
    3. dopisuję ten znak skonwertowany do małego do obecnego słowa i ustawiam flagę.

Funkcja `main` wygląda praktycznie tak samo jak na laboratorium (poza kosmetycznymi zmianami). Wczytuję w niej zawartość pliku, następnie wielokrotnie wywołuję wspomnianą funkcję (dla pomiarów czasowych przyjąłem 50000 razy) i dokonuję mierzenia czasu.

2.1.2 Pomiary czasowe, analiza profilera

Aby analiza profilerem miała sens, kompiluję kod bez optymalizacji `-O2` (profiler pokazuje wtedy `frame_dummy`, co utrudnia analizę). W tym i w następnych przypadkach używam profilera *gprof*, tak jak na laboratorium. Kompilacja i czas wykonania programu:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ normalize1.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
Time: 44.1895
```

Plik testowy **test** to załączony do laboratorium nr 2 plik *lab1.tex*. W późniejszej części wykonam testy wszystkich wersji programu dla różnych rozmiarów problemów oraz różnych proporcji występowanych modyfikacji tekstu. Wtedy również zastosuję optymalizację kompilatora (flaga `-O2`).
Jak widać, czas wykonania to prawie minuta i będzie stanowił on wyznacznik dla kolejnych wersji programu. Zobaczmy, co pokazuje profiler:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ -pg normalize1.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
...Documents/semestr6/OKNRA/zadania/zad1$ gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 58.24      3.25     3.25    50000    65.00    87.80  process_large_string(std::__cxx11
 20.43      4.39     1.14                             _init
  9.95      4.95     0.56 342100000     0.00     0.00  std::__cxx11::basic_string<char,
  4.66      5.21     0.26 74950000      0.00     0.00  __gnu_cxx::__enable_if<std::__is_
  3.41      5.39     0.19 74950000      0.00     0.01  bool std::operator!=<char, std::c
  2.24      5.52     0.12 47050000      0.00     0.00  std::__cxx11::basic_string<char,
  0.90      5.57     0.05      220    227.27   227.27  std::__cxx11::basic_string<char,
  0.18      5.58     0.01  4550000      0.00     0.00  std::char_traits<char>::compare(c
  0.00      5.58     0.00        2      0.00     0.00  dclock()
  0.00      5.58     0.00        1      0.00     0.00  __static_initialization_and_destr
  ...
```

Większość czasu spędzamy wewnątrz funkcji `process_large_string` - widzimy 50000 wywołań, bowiem tyle powtórzeń wywołania funkcji wykonuję dla miarodajnych wyników czasowych. Stosunkowo duży udział (około 20%) ma funkcja `_init`, która zapewne jest związana z używaniem funkcji bibliotecznych i powiązanym z tym

tworzeniem wielu obiektów (takie informacje znalazłem, szukając powodu wystąpienia tego składnika). Warto zwrócić uwagę na trzecią pozycję, związaną z alokowaniem nowego stringa za każdym razem, kiedy jest potrzeba jego zmiany (ponad 340 milionów wywołań!) - ten problem powinien zostać rozwiązany po 2. optymalizacji. Kolejne pozycje (z istotnym udziałem czasowym), jeżeli się nie mylę, również są związane z alokacją, ale na potrzeby wywołań funkcji "znakowych" z biblioteki `cctype`.

2.2 Optymalizacja nr 1 - prealokacja pamięci

Zgodnie z poleceniem: *Upewnij się, że wynikowy std::string ma zarezerwowaną odpowiednią ilość pamięci przed rozpoczęciem przetwarzania.*

```
...
bool flag;

// first optimization
result.reserve(s.length());

for(int i = 0; i < s.length(); i++) {
...
```

Listing 2: *normalize2.cpp* - zmiany względem *normalize1.cpp*

2.2.1 Opis implementacji

Jak widać powyżej, jedyna zmiana zachodzi w optymalizowanej funkcji - polega ona na zarezerwowaniu dla zmiennej `result` tyle pamięci, ile wynosi długość argumentu `s` - prosta obserwacja: wynik na pewno nie będzie dłuższy niż wejście, bo jedynie zmieniamy lub "usuwamy" znaki - zatem tyle wystarczy.

2.2.2 Pomiary czasowe, analiza profilera

Kompilacja i czas wykonania programu:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ normalize2.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
Time: 41.8034
```

Czas wykonania jest krótszy niż dla wersji podstawowej, ale dużej różnicy nie ma - zapewne wynika to z tego, że sama zmiana też nie była jakaś znacząca - ale stanowi dobre podstawy do kolejnych modyfikacji. Zobaczmy, co pokazuje profiler:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ -pg normalize2.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
...Documents/semestr6/OKNRA/zadania/zad1$ gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 61.53      3.55     3.55    50000    71.00    89.80  process_large_string(std::__cxx11
 21.66      4.80     1.25                             _init
  7.63      5.24     0.44 342100000     0.00     0.00  std::__cxx11::basic_string<char,
  5.72      5.57     0.33 74950000      0.00     0.00  __gnu_cxx::__enable_if<std::__is_
  1.39      5.65     0.08 74950000      0.00     0.01  bool std::operator!=<char, std::c
  1.39      5.73     0.08 47050000      0.00     0.00  std::__cxx11::basic_string<char,
  0.52      5.76     0.03      220    136.36   136.36  std::__cxx11::basic_string<char,
```

```
  0.17      5.77     0.01  4550000     0.00     0.00  std::char_traits<char>::compare(c
  0.00      5.77     0.00        2     0.00     0.00  dclock()
  0.00      5.77     0.00        1     0.00     0.00  __static_initialization_and_destr
  ...
```

Po pierwszej optymalizacji dużo się nie zmieniło. Oczywiście udziały % są delikatnie inne, ale względny udział i kolejność pod tym względem, jak i liczba wywołań wymienionych funkcji są bardzo podobne. Przekonamy się, czy następna optymalizacja rozwiąże problem kosmicznej wręcz liczby wywołań alokatora.

2.3 Optymalizacja nr 2 - in-place transformation

Aby zminimalizować liczbę alokacji, będę modyfikował ciąg w miejscu, jeżeli tylko jest to możliwe.
Uwaga: bazuję na *normalize2.cpp*, tzn. uwzględniam też pierwszą optymalizację.

```
...
for(int i = 0; i < s.length(); i++) {
    if(isspace(s[i])) {
        if(current != last) {
            if(current.length()) {
                result += current; // 2nd optimization
                last = current;
            }

            if(flag)
                result += ' '; // 2nd optimization
        }

        current.clear();
        flag = false;
    } else if(ispunct(s[i])) {
        if(current.length() && current != last) {
            result += current; // 2nd optimization
            last = current;
        }

        result += ','; // 2nd optimization
        current.clear();
        flag = true;
    } else if(s[i] >= 0x20 && s[i] <= 0x7E) {
        current += (char)tolower(s[i]); // 2nd optimization
        flag = true;
    }
}
...
```

Listing 3: *normalize3.cpp* - zmiany względem *normalize2.cpp*

2.3.1 Opis implementacji

Tutaj również zmieniam tylko optymalizowaną funkcję - zmiany pojawiają się w pętli (w liniach oznaczonych komentarzem, pozostałe umieściłem dla czytelności). Te zmiany gwarantują, że zamiast alokacji nowego stringa następuje jego modyfikacja w miejscu.

2.3.2 Pomiary czasowe, analiza profilera

Kompilacja i czas wykonania programu:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ normalize3.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
Time: 8.3909
```

Udało się, poprzez wydawałoby się nie aż tak dużą optymalizację, znacznie przyspieszyć działanie programu - aż pięciokrotnie! Zobaczmy, co pokazuje profiler:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ -pg normalize3.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
...Documents/semestr6/OKNRA/zadania/zad1$ gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 69.45      2.91     2.91    50000    58.20    64.10  process_large_string(std::__cxx11
 22.43      3.85     0.94                             _init
  3.58      4.00     0.15 74950000     0.00     0.00  __gnu_cxx::__enable_if<std::__is_
  3.34      4.14     0.14 74950000     0.00     0.00  bool std::operator!=<char, std::c
  0.95      4.18     0.04      220   181.82   181.82  std::__cxx11::basic_string<char,
  0.12      4.18     0.01  4550000     0.00     0.00  std::char_traits<char>::compare(c
  0.12      4.19     0.01                             std::char_traits<char>::lt(char c
  0.00      4.19     0.00        2     0.00     0.00  dclock()
  0.00      4.19     0.00        1     0.00     0.00  __static_initialization_and_destr
  ...
```

Można powiedzieć, że potwierdziły się nasze przewidywania. Nieco wzrósł % udział `process_large_string` oraz `_init`, natomiast nie straszy już 300 milionów wywołań do alokatora.
Co prawda nadal są wywołania związane z używaniem funkcji znakowych, ale ze względu na specyfikę programu nie ograniczymy tego. Na szczęście ich udział jest już mniejszy (choć zauważalny), a wywołań jest o rząd wielkości mniej (niecałe 75 milionów).

2.4 Optymalizacja nr 3 - iteratory i algorytmy standardowe

Wykorzystuję iteratory oraz funkcje z biblioteki standardowej, takie jak `std::transform`, `std::unique` oraz `std::remove_if`.
Uwaga: bazuję na *normalize3.cpp*, tzn. uwzględniam też poprzednie optymalizacje.
Uwaga nr 2: wedle przyjętych założeń wydzielania słów, ta wersja okazała się dość problematyczna w implementacji. Moim głównym celem było zachowanie równoważnego działania programu (poprawnego przy tych założeniach).

```
...
#include <cctype>
#include <algorithm>
#include <vector>
#include <sstream>

#define REPEATS 50000
...
```

```cpp
// helper function to detect ASCII characters out of given codes scope
bool nonprintable(char & ch) {
    return ch < 0x20 || ch > 0x7E;
}

// transformations regards to task contents
char transform_character(char & ch) {
    if(isspace(ch)) {
        return '␣';
    } else if(ispunct(ch)) {
        return ',';
    }

    return tolower(ch);
}

// helper function to split string, based on:
// https://medium.com/@ryan_forrester_/splitting-strings-in-c-a-complete-guide-cf16283
void split_string(std::vector<std::string> & words, const std::string & str) {
    std::istringstream iss(str);
    std::string word;

    while (iss >> word)
        words.emplace_back(word);
}

// helper function so that DRY rule is respected
void fill_between_words(
    const std::string & s, std::string & result, int pos1, int pos2) {
    bool flag = true;

    for(pos1; pos1 < pos2; pos1++) {
        if(isspace(s[pos1]) && flag) {
            result += "␣"; // 2nd optimization
            flag = false;
        } else if(ispunct(s[pos1])) {
            result += ","; // 2nd optimization
            flag = true;
        }
    }
}

// function to be optimized
std::string process_large_string(std::string s) {
    std::string result, copy;
    std::vector<std::string> split;
    int old_pos = 0, new_pos;

    // first optimization
    result.reserve(s.length());

    // third optimization - std library algorithms
    std::transform(s.begin(), s.end(), s.begin(), transform_character);
    auto end = std::remove_if(s.begin(), s.end(), nonprintable);
```

```
    s.erase(end, s.end());

    // preparing & saving a copy at this stage
    copy.resize(s.length());
    std::copy(s.begin(), s.end(), copy.begin());

    // trick to help splitting words
    std::transform(
        copy.begin(), copy.end(), copy.begin(), [](char ch){
        if(ch == ',') return ' '; return ch; });

    // split string into tokens (words separated by spaces)
    split_string(split, copy);

    // eliminating word duplicates
    auto words_end = std::unique(split.begin(), split.end());

    // joining words back together, including "lost" commas and single whitespaces
    for(auto it = split.begin(); it != words_end; ) {
        new_pos = s.find(*it, old_pos);
        fill_between_words(s, result, old_pos, new_pos);
        old_pos = new_pos + (*it).length();
        result += *it++; // 2nd optimization
    }

    // commas or spaces at the end (after last word)
    fill_between_words(s, result, old_pos, s.length());

    return result;
}
...
```

Listing 4: *normalize4.cpp* - zmiany względem *normalize3.cpp*

2.4.1 Opis implementacji

W tej wersji zmian jest już dużo więcej. Przede wszystkim dodałem kilka funkcji pomocniczych. Aby korzystać z algorytmów standardowych i potrzebnych struktur danych, uwzględniam nagłówki bibliotek `<algorithm>`, `<vector>`, `<sstream>`.
Dodałem funkcję `nonprintable`, która pełni rolę predykatu w `std::remove_if` - sprawdza, czy znak ma kod poza akceptowanym zakresem. Funkcja `transform_character` jest używana do zamiany znaków zgodnie z poleceniem zadania. Do dzielenia łańcucha znaków na słowa, co jest pomocne przy usuwaniu duplikatów, służy funkcja `split_string`, której implementacja jest zainspirowana tym artykułem.
Dla czytelności umieściłem optymalizowaną funkcję w całości. Wyjątkowo nie przekazuję argumentu `s` przez referencję, bowiem modyfikuję go, poza tym tworzona zmienna `copy` po wstępnych przetworzeniach zachowuje kopię wejścia. Wykorzystując wymienione wcześniej funkcje, dokonałem kolejno (kod po komentarzu *third optimization...*):

1. transformacji znaków - konwersji znaków interpunkcyjnych na przecinek, białych znaków na spację oraz pozostałych znaków na ich "małą" wersję,
2. usunięcia znaków o kodach spoza zakresu 32 do 126,
3. zapisania kopii wejścia po tych zmianach,
4. działając dalej na kopii: zamiany przecinków na spacje, żeby zgodnie z założeniami wydzielić słowa,

5. wydzielenia słów oraz usunięcia duplikatów z wykorzystaniem funkcji `std::unique`,

6. po usunięciu duplikatów, "sklejenia" z powrotem słów, przy zachowaniu przecinków i pojedynczych spacji. W tym celu wykorzystuję "oryginalne" wejście (ale po transformacjach), bowiem w nim są one zachowane. Za pomocą funkcji `find` znajduję położenia kolejnych słów (już bez duplikatów) i w zakresie pozycji od poprzedniego słowa do obecnego "przepisuję" odpowiednio spacje i przecinki.
   Odpowiada za to funkcja `fill_between_words`. Jako że interesują nas tylko białe znaki i znaki interpunkcyjne, to właśnie takie przypadki są obsługiwane (przy czym może wystąpić sekwencja białych znaków, stąd zmienna `flag`, działająca podobnie jak w poprzednich wersjach).

2.4.2 Pomiary czasowe, analiza profilera

Kompilacja i czas wykonania programu:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ normalize4.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
Time: 30.9316
```

Ta wersja programu jest odmienna od pozostałych. Zapewne ze względu na wspomniane trudności implementacyjne i nieco zagmatwaną realizację algorytmu, osiągi czasowe są nienajlepsze (choć oczywiście lepsze niż pierwszych dwóch wersji programu). Zobaczmy, co pokazuje profiler:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ -pg normalize4.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
...Documents/semestr6/OKNRA/zadania/zad1$ gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 15.01      2.56     2.56 2156950000     0.00     0.00  __gnu_cxx::__normal_iterator<ch
 12.40      4.67     2.12 2495700000     0.00     0.00  __gnu_cxx::__normal_iterator<ch
 10.84      6.53     1.85 1078050000     0.00     0.00  bool __gnu_cxx::operator!=<char
  9.26      8.11     1.58 2156600000     0.00     0.00  __gnu_cxx::__normal_iterator<ch
  6.04      9.13     1.03 369600000      0.00     0.00  transform_character(char&)
  3.99      9.81     0.68    50000       0.01     0.10  __gnu_cxx::__normal_iterator<char
  3.81     10.46     0.65 47100000       0.00     0.00  fill_between_words(std::__cxx11::
  3.40     11.04     0.58    50000       0.01     0.07  __gnu_cxx::__normal_iterator<char
  2.99     11.55     0.51 369600000      0.00     0.00  bool __gnu_cxx::__ops::_Iter_pre
  2.87     12.04     0.49 369600000      0.00     0.00  nonprintable(char&)
  2.75     12.52     0.47    50000       0.01     0.08  __gnu_cxx::__normal_iterator<char
  1.96     12.85     0.34    50000       0.01     0.01  __gnu_cxx::__ops::_Iter_pred<bool
  1.82     13.16     0.31                               _init
  1.61     13.44     0.28 354700000      0.00     0.00  process_large_string(std::__cxx1
  1.47     13.69     0.25 375150000      0.00     0.00  __gnu_cxx::__normal_iterator<std
  1.35     13.91     0.23 338750000      0.00     0.00  std::remove_reference<char&>::ty
  1.20     14.12     0.20    50000       0.00     0.33  process_large_string(std::__cxx11
  1.17     14.32     0.20 94600000       0.00     0.00  bool __gnu_cxx::operator!=<std::_
  1.03     14.49     0.17    50000       0.00     0.02  __gnu_cxx::__normal_iterator<std:
  ...
```

Z powodu użycia algorytmów standardowych output profilera jest bardzo długi (sama "tabelka" miała ponad 100 linijek - ograniczyłem pokazywane tutaj wyniki do tych z udziałem powyżej 1%). Jako że zdefiniowałem wiele funkcji pomocniczych, które są używane w funkcjach z biblioteki `<algorithm>`, to ich udział (czas, który "spędzamy" w tych funkcjach) jest większy niż bezpośrednio w samej funkcji `process_large_string`.

Struktura jest spłaszczona - największy udział wynosi około 15%, najwięcej czasu zajmują funkcje związane z iterowaniem po łańcuchu znaków. Liczba wywołań do nich jest abstrakcyjnie duża - rzędu miliardów. Możliwe, że to właśnie z powodu tak dużej liczby wywołań funkcji ta wersja jest wolniejsza - mimo że nie neguję zapewnionej efektywnej implementacji samych używanych algorytmów standardowych.

2.5 Optymalizacja nr 4 - wersja bare metal

Operuję na surowych tablicach znaków z użyciem wskaźników (`char*`) oraz funkcji `memcpy` do kopiowania łańcuchów znaków.
Uwaga: bazuję na *normalize3.cpp*, tzn. uwzględniam też wcześniejsze optymalizacje, poza bezpośrednio poprzednią - ze względu na słabsze wyniki oraz charakter tej optymalizacji.

```
...
#include <cctype>
#include <cstring>

#define REPEATS 50000
...

// function to be optimized
// fourth optimization - char* instead of std::string
void process_large_string(const char* s, char* result, size_t size) {
    std::string last, current;
    bool flag;

    // to use in memcpy
    char* space = (char*)"␣";
    char* comma = (char*)",";

    for(int i = 0; i < size; i++) {
        if(isspace(s[i])) {
            if(current != last) {
                if(current.length()) {
                    memcpy(result, current.data(), current.length()); // 4th optimizati
                    result += current.length();
                    last = current;
                }

                if(flag) {
                    memcpy(result, space, 1); // 4th optimization
                    result += 1;
                }
            }

            current.clear();
            flag = false;
        } else if(ispunct(s[i])) {
            if(current.length() && current != last) {
                memcpy(result, current.data(), current.length()); // 4th optimization
                result += current.length();
                last = current;
            }

            memcpy(result, comma, 1); // 4th optimization
            result += 1;
```

```
            current.clear();
            flag = true;
        } else if(s[i] >= 0x20 && s[i] <= 0x7E) {
            current += (char)tolower(s[i]); // 2nd optimization
            flag = true;
        }
    }

    *result = 0;
}
...

while(getline(std::cin, line))
    s += line + "\n";

char* results = (char*)malloc(sizeof(char) * s.length());

// we start the clock
dtime = dclock();

for(int i = 0; i < REPEATS; i++) {
    process_large_string(s.data(), results, s.length());
    result = results;
}

// calculate the time taken
...

fflush(stdout);
free(results);

return iret;
...
```

Listing 5: *normalize5.cpp* - zmiany względem *normalize3.cpp*

2.5.1 Opis implementacji

Zmiany w tej wersji obejmują funkcję `process_large_string` (dla czytelności zamieszczam ją w całości) oraz funkcję `main` (tę z kolei we fragmentach), przy czym są one związane głównie z operowaniem na wskaźnikach, a nie typie `std::string` - zasadniczo, algorytm jest identyczny. Aby korzystać z funkcji `memcpy`, dodaję nagłówek `<cstring>`. Tam, gdzie wcześniej używałem `result += ...` teraz jest `memcpy(result, ...)` i później przesunięcie wskaźnika `result` o odpowiednią liczbę pozycji. Typ zwracany zmieniłem na `void`, bowiem wynik jest zapisywany poprzez wskaźnik `result` przekazany jako argument.
W samej funkcji nadal używam typu `std::string` - zmienne `last` i `current`. Są to zmienne pomocnicze, natomiast gdybym miał używać ich jako wskaźników, powstałyby pewne problemy:

- jako że nie znam rozmiaru słowa, musiałbym albo przygotować dość dużą tablicę statyczną (w skrajnym przypadku długości całego wejścia), albo przydzielać pamięć dynamicznie (ale tu też musiałbym mieć gwarancję, że pamięci wystarczy),
- mógłbym też ustawiać wskaźniki na odpowiedniej pozycji wejścia, żeby nie przydzielać dodatkowej pamięci, ale wtedy, śledząc poprzednie i aktualne słowo, musiałbym wstawiać 0 tam, gdzie ono się kończy –> modyfikacja łańcucha wejściowego! W takim przypadku zapewne potrzebna byłaby jego kopia.

Jako że nie znalazłem sprytnego rozwiązania problemu, a przedstawione pomysły wiążą się z dodatkowym narzutem, który mógłby wpłynąć na efektywność optymalizacji i skomplikowanie kodu, pozostałem przy tych zmiennych, tak jak było wcześniej.
Z kolei w funkcji `main` alokuję dynamicznie pamięć na łańcuch z wynikiem (`results`), która na końcu jest zwalniana (`free...`), a w pętli przekazuję odpowiednie argumenty zgodnie z sygnaturą funkcji.

2.5.2 Pomiary czasowe, analiza profilera

Kompilacja i czas wykonania programu:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ normalize5.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
Time: 5.34654
```

Udało się poprawić szybkość programu względem wersji 3. - na tej wersji bowiem bazuje ta optymalizacja. Zastosowanie wskaźników na łańcuch znaków zamiast stringów przyspiesza działanie programu (rezygnujemy ze zbędnej tak naprawdę otoczki obiektowej, która nie jest potrzebna dla poprawności algorytmu). Zobaczmy, co pokazuje profiler:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ -pg normalize5.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ ./a.out < test
...
...Documents/semestr6/OKNRA/zadania/zad1$ gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 71.23      2.03     2.03    50000    40.60    45.60  process_large_string(char const*
 20.00      2.60     0.57                             _init
  5.26      2.75     0.15 74950000     0.00     0.00  bool std::operator!=<char, std::c
  3.51      2.85     0.10 74950000     0.00     0.00  __gnu_cxx::__enable_if<std::__is_
  0.00      2.85     0.00  4550000     0.00     0.00  std::char_traits<char>::compare(c
  0.00      2.85     0.00      220     0.00     0.00  std::__cxx11::basic_string<char,
  0.00      2.85     0.00        2     0.00     0.00  dclock()
  0.00      2.85     0.00        1     0.00     0.00  __static_initialization_and_destr
   ...
```

Struktura wyników powraca do już znajomej sprzed poprzedniej wersji, przy czym tylko 4 pozycje mają istotny udział procentowy. Ponad 70% czasu spędzamy w funkcji `process_large_string`, równo 20% w `_init`, a pozostałe niecałe 10% jest związane z funkcjami znakowymi.

2.6 Optymalizacja nr 5 - wersja równoległa

Przetwarzam tekst w wielu wątkach z wykorzystaniem `OpenMP`.
Uwaga: bazuję na *normalize5.cpp*.
Uwaga nr 2: równoległe przetwarzanie dotyczy transformacji i "usuwania" odpowiednich znaków, po którym sklejam uzyskane fragmenty wyniku. Następnie wykrywanie duplikatów oraz sekwencji białych znaków odbywa się dla całego tekstu w jednym wątku, bowiem podzielenie tekstu mogłoby spowodować przeoczenie duplikatów.

```
...
#include <cstring>
#include <cmath>
#include <omp.h>
```

```
#define REPEATS 50000
...

// fifth optimization - function that can be run concurrently
void transform_commas_and_whitespaces(const char* s, char* result, size_t size) {
    // to use in memcpy
    char* space = (char*)" ";
    char* comma = (char*)",";

    for(int i = 0; i < size; i++) {
        if(isspace(s[i])) {
            memcpy(result, space, 1); // 4th optimization
            result += 1;
        } else if(ispunct(s[i])) {
            memcpy(result, comma, 1); // 4th optimization
            result += 1;
        } else if(s[i] >= 0x20 && s[i] <= 0x7E) {
            memcpy(result, s+i, 1); // 4th optimization
            result += 1;
        }
    }

    *result = 0;
}

// function to be optimized
...

for(int i = 0; i < size; i++) {
    if(s[i] == ' ') {
        if(current != last) {
...

        flag = false;
    } else if(s[i] == ',') {
        if(current.length() && current != last) {
...

        flag = true;
    } else {
        current += (char)tolower(s[i]); // 2nd optimization
...

int main(int argc, const char* argv[]) {
    int i, j, k, iret, num_threads, part_len;
    double dtime;
    std::string s, result, line, entry;

    std::cout << "Let's start processing the file\n";

    while(getline(std::cin, line))
        s += line + "\n";
```

```
    // prepare for dividing data
    #pragma omp parallel
    {
        #pragma omp single
        num_threads = omp_get_num_threads();
    }

    part_len = s.length() / num_threads;
    char **results = (char**)malloc(sizeof(char*) * num_threads);
    char* results_ = (char*)malloc(sizeof(char) * s.length());

    for(int i = 0; i < num_threads; i++)
        results[i] = results_ + i * part_len;

    // we start the clock
    dtime = dclock();

    for(int i = 0; i < REPEATS; i++) {
        #pragma omp parallel
        {
            int id = omp_get_thread_num();

            if(id < num_threads - 1) {
                transform_commas_and_whitespaces(
                    s.data() + id * part_len, results[id], part_len);
            } else {
                transform_commas_and_whitespaces(
                    s.data() + id * part_len, results[id],
                    s.length() - (num_threads - 1) * part_len);
            }
        }

        for(int j = 0; j < num_threads; j++)
            entry += results[j];

        process_large_string(entry.data(), results_, entry.length());
        result = results_;
        entry.clear();
    }

    // calculate the time taken
    dtime = dclock() - dtime;

    std::cout << result << "\n";
    std::cout << "Time: " << dtime << "\n";
    fflush(stdout);
    free(results);
    free(results_);

    return iret;
}
```

Listing 6: *normalize6.cpp* - zmiany względem *normalize5.cpp*

2.6.1 Opis implementacji

Względem poprzedniej wersji dodałem nagłówki `<cmath>` i `<omp.h>`, żeby korzystać z `OpenMP` oraz obliczać długość fragmentu tekstu, który trafia do jednego wątku. Funkcja, która jest wykonywana współbieżnie, to `transform_commas_and_whitespaces`, w której zamieniam każdy biały znak na spację, znak interpunkcyjny na przecinek oraz znak o kodzie z przyjętego zakresu przepisuję bez zmian. W samej optymalizowanej funkcji doszło do pewnych uproszczeń, bo po zamianie wystarczy sprawdzać spacje, przecinki, a pozostałe przypadki dają gwarancję znaku z przyjętego zakresu.
W tym przypadku największe zmiany zaszły w funkcji `main`, którą dla czytelności zamieszczam w całości. W uproszczeniu kolejno:

1. wczytuję dane z wejścia standardowego,
2. odczytuję liczbę wątków (trzeba to robić w bloku `#pragma omp parallel`),
3. obliczam długość fragmentu tekstu dla pojedynczego wątku, na podstawie tego ustawiam wskaźniki, które będą przekazane do funkcji przetwarzanej współbieżnie i wykonuję tę funkcję,
4. po wstępnym przetworzeniu "sklejam" wyniki i wykonuję funkcję `process_large_string`, korzystając z tego rezultatu sklejenia, **już nie współbieżnie**.

2.6.2 Pomiary czasowe, analiza profilera

Kompilacja i czas wykonania programu:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ -fopenmp normalize6.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ OMP_NUM_THREADS=5 ./a.out < test
...
Time: 7.30966
```

Jako że korzystam z `OpenMP`, przy kompilacji dodaję flagę `-fopenmp`. Z kolei przy uruchamianiu programu dodaję `OMP_NUM_THREADS=5` (dla spójności, wszędzie gdzie uruchamiam tę wersję programu, korzystam właśnie z 5 wątków). Ze względu na to, że jedynie część algorytmu jest wykonywana współbieżnie, wynik czasowy nie jest lepszy niż poprzedniej wersji (co więcej, jest zauważalnie gorszy, choć nie jest źle). Zobaczmy, co pokazuje profiler:

```
...Documents/semestr6/OKNRA/zadania/zad1$ g++ -fopenmp -pg normalize6.cpp
...Documents/semestr6/OKNRA/zadania/zad1$ OMP_NUM_THREADS=5 ./a.out < test
...
...Documents/semestr6/OKNRA/zadania/zad1$ gprof ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 44.87      1.40     1.40   132808    10.54    10.54  transform_commas_and_whitespaces(
 39.42      2.63     1.23    50000    24.60    28.60  process_large_string(char const*
  9.29      2.92     0.29                             _init
  4.49      3.06     0.14 74950000     0.00     0.00  __gnu_cxx::__enable_if<std::__is_
  1.92      3.12     0.06 74950000     0.00     0.00  bool std::operator!=<char, std::c
  0.00      3.12     0.00  4550000     0.00     0.00  std::char_traits<char>::compare(c
  0.00      3.12     0.00      220     0.00     0.00  std::__cxx11::basic_string<char,
  0.00      3.12     0.00        2     0.00     0.00  dclock()
  0.00      3.12     0.00        1     0.00     0.00  __static_initialization_and_destr
  ...
```

Rezultat jest interesujący. Główny udział (po około 40%) w czas wykonania programu mają dwie funkcje: wykonywana współbieżnie `transform_commas_and_whitespaces` oraz `process_large_string` (ona już niewspółbieżnie). Dalej są funkcja `_init` (z mniejszym niż wcześniej udziałem) oraz funkcje znakowe, podobnie jak wcześniej.

3 Testy programu dla różnych przypadków testowych

3.1 Przygotowanie przypadków testowych

Aby przetestować programy w przypadku tekstów różnej natury, wygenerowałem 3 warianty tekstów:

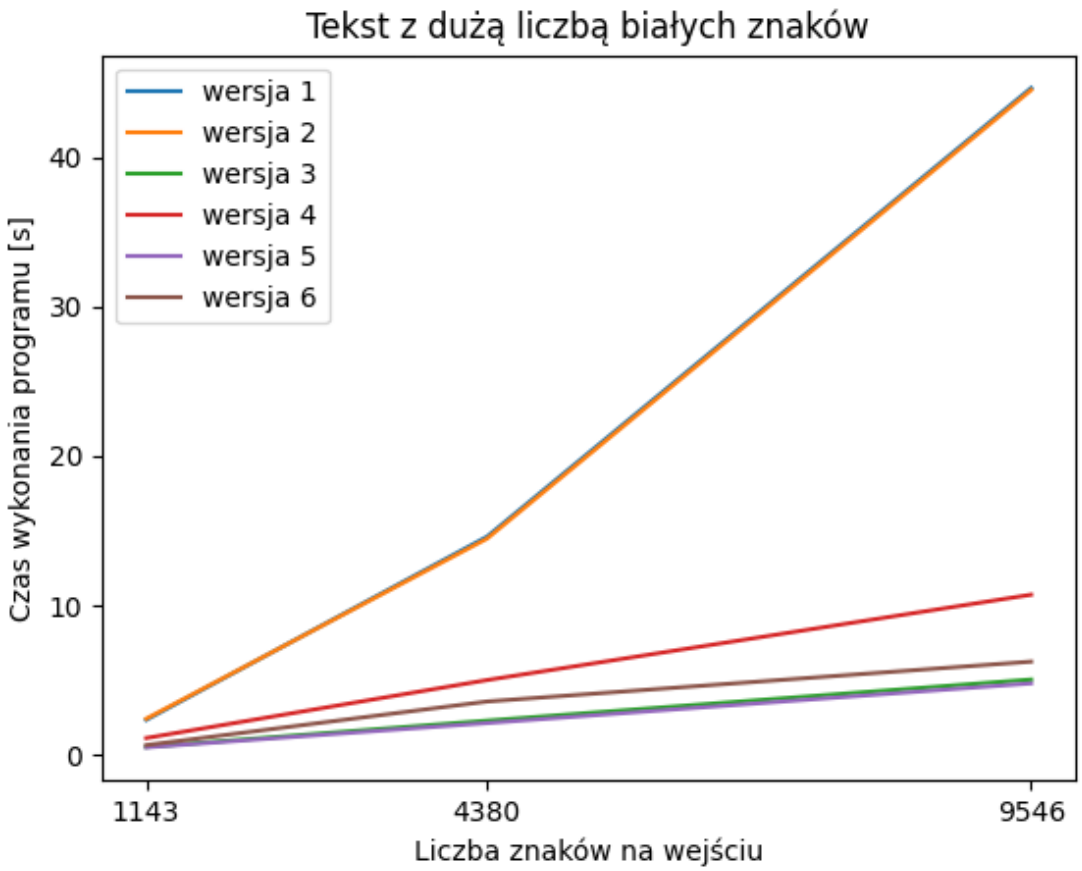
1. tekst, w którym występuje ponadprzeciętna liczba białych znaków,
2. tekst, w którym występuje ponadprzeciętna liczba znaków interpunkcyjnych,
3. tekst, w którym występuje dużo duplikatów słów występujących bezpośrednio po sobie.

Ponadto, dla każdego z tych wariantów przygotowałem teksty o różnej objętości:

1. krótki - długości około 1000 znaków,
2. średni - długości około 4000 znaków,
3. długi - długości około 9000 znaków.

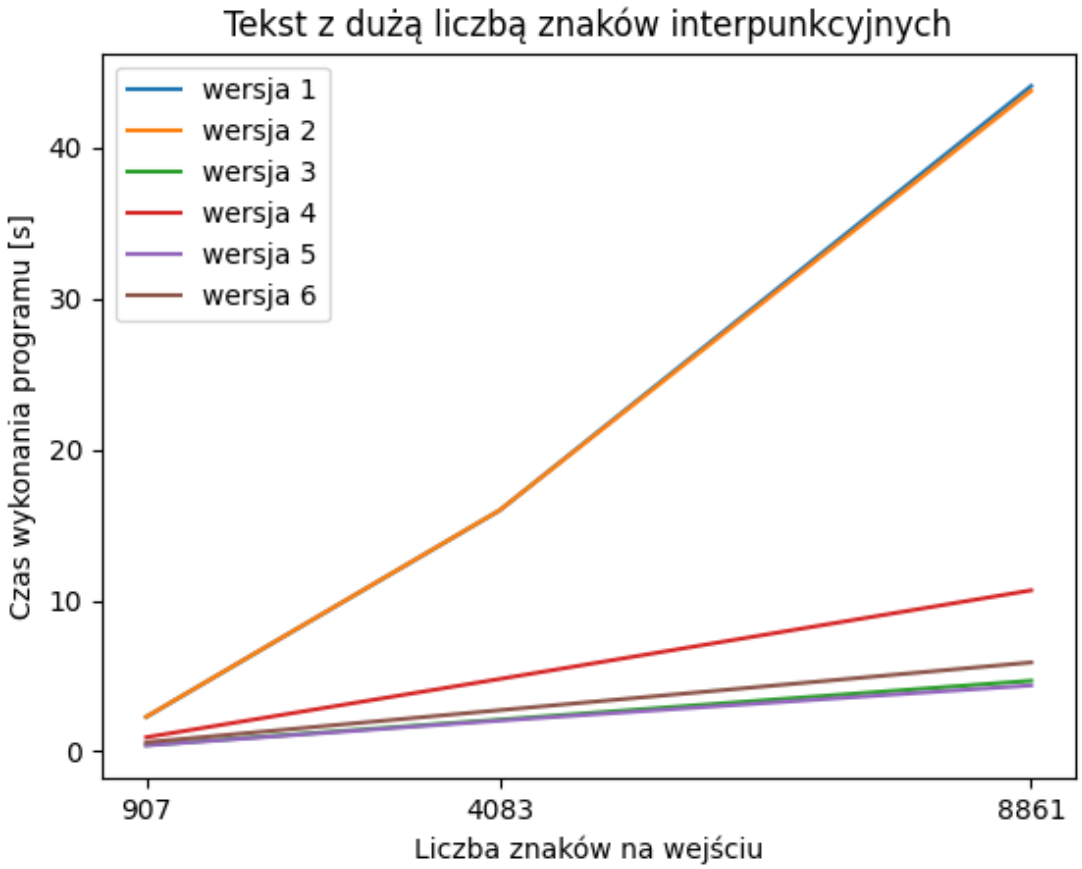
Dokładna długość tekstu będzie podana dla każdego przypadku. Tymczasem przejdźmy do testowania. Wyniki zostaną przedstawione na oddzielnych wykresach porównawczych dla każdej wersji programu, następnie uśrednione pomiędzy wszystkimi przypadkami.

3.2 Tekst z dużą liczbą białych znaków



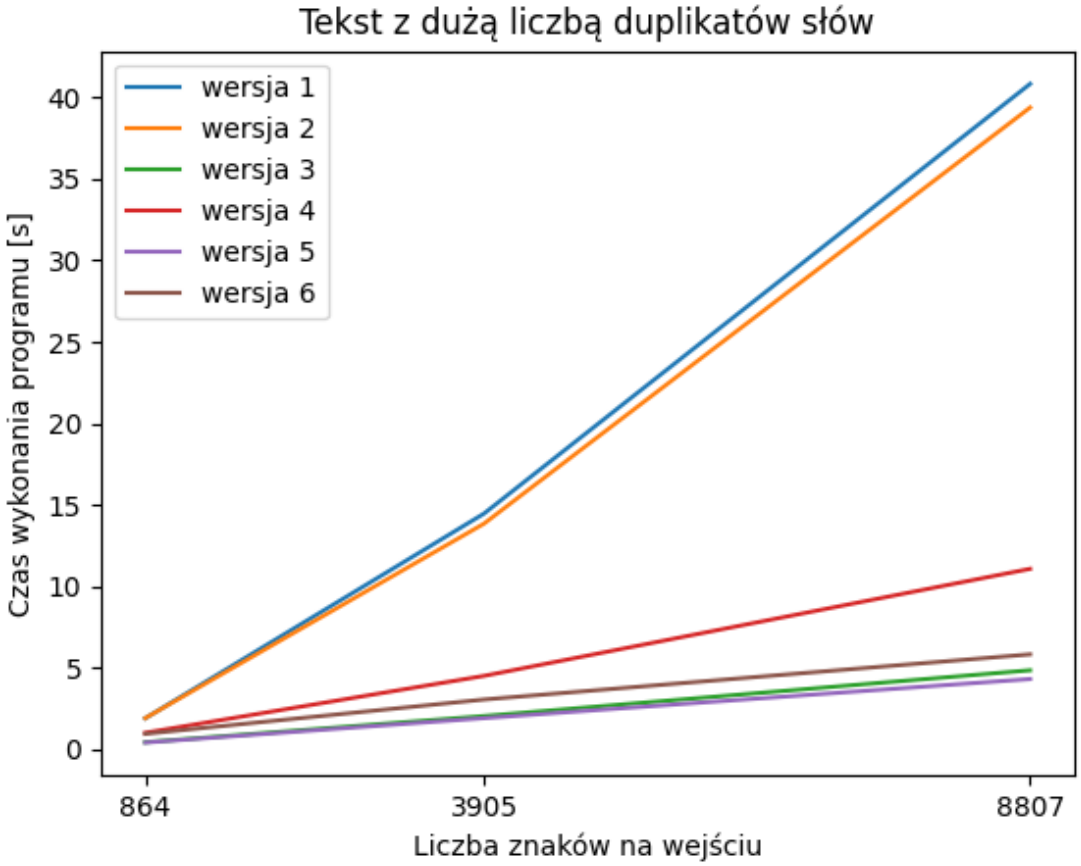
Wartości zaznaczone na osi poziomej to długości odpowiednio dla testu krótkiego, średniego i długiego. Podobnie będzie dla następnych wariantów testów.
Jak widać na wykresie, dwie pierwsze wersje programu mają bardzo podobne wyniki czasowe. O wiele lepiej niż we wcześniejszej części wypada wersja 4. (optymalizacja `-O2` zapewne na to wpłynęła), bowiem dla najdłuższego testu czas wynosi około 10 sekund. Pozostałe wersje mają jednak zauważalnie lepsze czasy. Co może wydawać się ciekawe, najlepsze są wersje 3. i 5., a wersja 6. (wielowątkowa) ma delikatnie gorszy czas - ale może to wynikać z faktu, że całości operacji nie udało mi się zrównoleglić. Dla najlepszych wersji czas spada poniżej 5 sekund (oczywiście dla najdłuższego testu).

3.3 Tekst z dużą liczbą znaków interpunkcyjnych



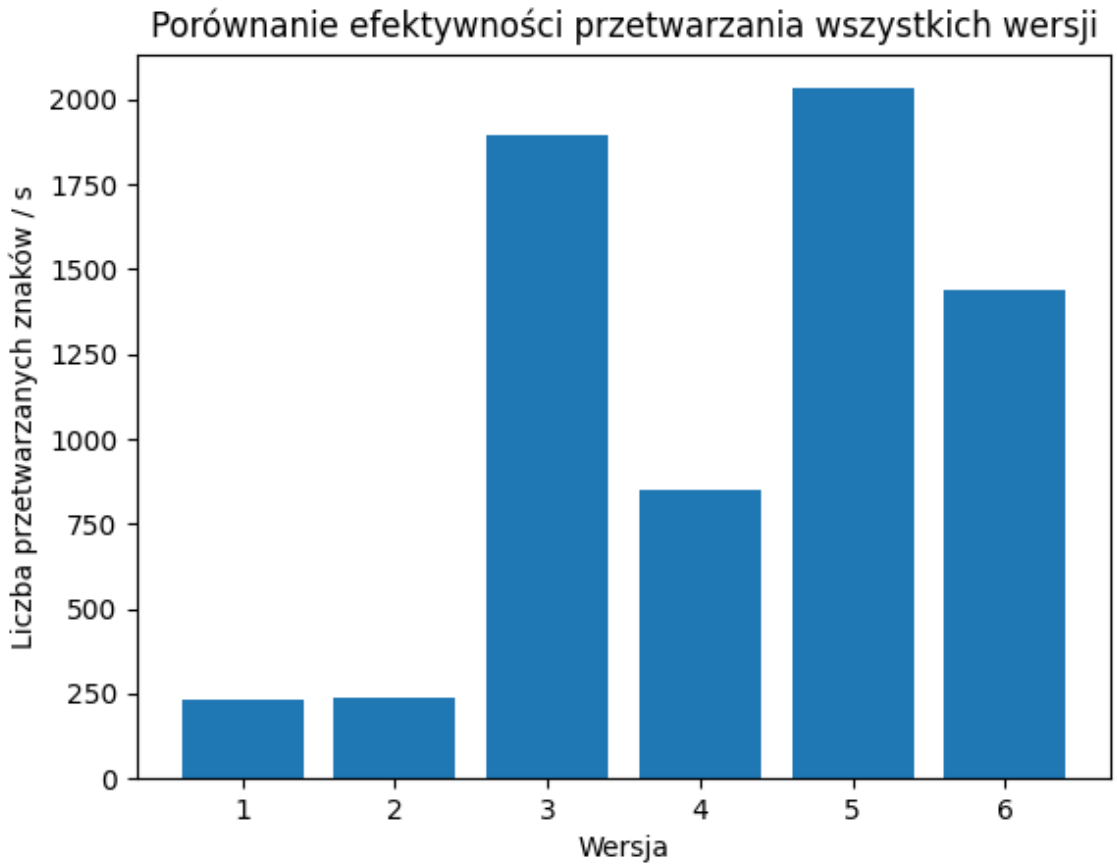
Jak widać, przebieg wykresów jest bardzo podobny jak dla wcześniejszych testów, chociaż każdy z testów jest nieco krótszy. Może to wynikać z faktu, że w przypadku sekwencji białych znaków do wyniku trafia tylko jeden z nich (zamieniony na spację), więc "efektywna" długość tekstu jest krótsza.
Osiągi czasowe poszczególnych wersji programu dla kolejnych testów są niemal identyczne jak w poprzednim przypadku.

3.4 Tekst z dużą liczbą duplikatów słów



W tym przypadku widać pewne różnice. Wersja 2. jest zauważalnie lepsza od wersji 1. (mimo że są to małe różnice). Długość kolejnych testów jest krótsza od wcześniejszych, ale tylko trochę - więc można je nazwać porównywalnymi. Podobnie jak w pierwszym wariancie, efektywna długość wyniku jest krótsza niż długość wejścia, jednakże samo wykrywanie duplikatów (poprzez porównywanie sąsiadujących słów) jest bardziej kosztowne - poniekąd się to "wyrównuje", bowiem czasy wykonania są podobne (trochę krótsze) oraz relacje między wersjami programu pod tym względem są takie same, jak w poprzednich przypadkach.

3.5 Uśrednione porównanie wszystkich wersji programu



Zsumowałem czasy wykonania dla każdego typu testu i każdej długości testu, jak i długości wszystkich testów. Podzielenie zsumowanej długości przez zsumowane czasy pozwala obliczyć "efektywność" każdej z wersji programu, czyli liczbę przetwarzanych znaków na sekundę.
Potwierdziły się wnioski, które można było wyciągnąć, analizując poprzednie punkty. Najlepiej radzi sobie wersja 5., która osiąga efektywność ponad 2000 znaków na sekundę. Tuż za nią plasuje się wersja 3., zauważalnie gorsza jest wersja 6. (poniżej 1500 znaków / s), natomiast wersja 4. wydawała im się aż tak nie odstawać, tymczasem jej efektywność wynosi poniżej 1000 znaków na sekundę. Naturalnie na końcu są dwie pierwsze wersje, które przetwarzają poniżej 250 znaków na sekundę.

W skrócie - **bare metal** wygrywa! Czy można było tak przewidywać? Faworytem mogła być też wersja równoległa. Warto też docenić, jak bardzo efektywność poprawia ograniczenie alokacji stringów.

4 Podsumowanie, wnioski

Zrealizowane zadanie było bardzo ciekawe, a w niektórych miejscach sprawiło większe problemy, przez co trzeba było przemyśleć implementację wymaganych funkcjonalności. Pomocne było to, że stanowiło tak naprawdę rozszerzenie zadania laboratoryjnego, więc "baza do zadania" była gotowa. Pozostało jedynie (i aż) dodać odpowiednie przetwarzanie znaków i wykrywanie duplikatów, co starałem się zrobić możliwie efektywnie.

Kolejne wersje programu, w których dodawałem coraz więcej optymalizacji względem prostego, jednakże niewydajnego, początkowego algorytmu pozwoliły wyciągnąć ważne wnioski. Mogłoby się wydawać, że czasy wykonywania programu powinny być lepsze wraz z każdą kolejną optymalizacją. Tymczasem tak jak druga optymalizacja przyniosła znaczną poprawę, tak kolejna wypadła na tym polu o wiele gorzej. Podobnie było pomiędzy dwiema ostatnimi wersjami - nastąpiło pogorszenie czasu wykonania.

To właśnie te wersje, które nie przyniosły poprawy względem poprzedniej, sprawiły mi więcej problemów niż pozostałe. Starałem się, żeby działanie algorytmu pozostało poprawne (i przede wszystkim równoważne) innym wersjom. Nie wykluczam, że mogłem popełnić błędy, które spowodowały, że nie osiągnąłem oczekiwanego rezultatu. Niemniej jednak napędziło to wnioski i skłoniło do głębszej analizy właśnie tych przypadków.