Jezyki skryptowe i ich zastosowania, zadanie 1

Piotr Sieński 184297

Kwiecień 2024

1 Implementowana funkcja

Wybrana funkcja jest `str.rfind(sub[, start[, end]])`. Funkcja ta w Pythonie wyszukuje ciag znaków `sub` wewnatrz ciagu, od którego jest wywoływana, przeszukujac od końca do poczatku. Zwraca najwyższy indeks, gdzie podciag `sub` został znaleziony, w zakresie indeksów od `start` do `end`. Jeżeli `sub` nie zostanie znaleziony, funkcja zwraca -1. Parametry `start` i `end` sa opcjonalne i służa do ograniczenia przeszukiwanego zakresu, gdzie `start` określa poczatkowy indeks, a `end` końcowy indeks przeszukiwania.

1.1 Założenia

Implementowana przeze mnie funkcja działa analogicznie do wersji bibliotecznej, z tym wyjatkiem że ciag znaków w którym wyszukiwany jest podciag jest podawany jako pierwszy argument funkcji, zamiast wywołania bezpośrednio na obiekcie ciagu znaków.

2 Implementacja Python

Listing 1: Implementacja funkcji rfind() w Pythonie

```
def custom_rfind(s, sub, start=None, end=None):
    # If start is None, set it to beggining of the string
    if start is None:
        start = 0

    # If end is None, set it to the length of the string
    if end is None or end > len(s):
        end = len(s)

    # Adjust negative indices
    if start < 0:
        start = max(0, len(s) + start)
    if end < 0:
        end = max(0, len(s) + end)

    # Check if search space is valid
    search_space_len = end - start
```

```
if search_space_len < 0:
    return -1

# Check if the substring is longer than the search space
sub_len = len(sub)
if sub_len == 0:
    return end
if sub_len > search_space_len:
    return -1

# Iterate over the string in reverse within the specified range
for i in range(end - sub_len, start - 1, -1):
    if s[i:i+sub_len] == sub:
        return i
return -1
```

Sposób działania funkcji:

1. Funkcja ustawia wartość `start` na 0, jeśli nie jest podana, co pozwala na rozpoczecie przeszukiwania od poczatku ciagu.
2. Jeśli wartość `end` nie jest określona lub przekracza długość ciagu, funkcja ustawia ja na długość tego ciagu, umożliwiajac przeszukiwanie do jego końca.
3. Dostosowanie indeksów ujemnych dla `start` i `end`
4. Weryfikacja, czy przestrzeń poszukiwań (`end - start`) jest poprawna (nieujemna), eliminuje sytuacje, w których zakres przeszukiwań byłby niewłaściwie zdefiniowany.
5. Gdy `sub` jest pustym ciagiem, funkcja zwraca wartość `end`, co jest zgodne z zachowaniem metody bibliotecznej.
6. Jeśli długość podciagu `sub` przekracza przestrzeń poszukiwań, funkcja zwraca -1, wskazujac na niemożność znalezienia podciagu.
7. Iteracja przez ciag od końca do poczatku w określonym zakresie i porównywanie podciagów umożliwia odnalezienie najwyższego indeksu wystapienia `sub`.
8. Funkcja zwraca -1 w przypadku nieznalezienia podciagu

3 Implementacja C++

Implementacja w C++ działa na tej samej zasadzie co w Pythonie, z ta różnica, że argument `start` obiera wartość poczatkowa 0, a `end` wartość -1, ze wzgledu na fakt że C++ jest jezykiem silnie typowanym, a użycie bardziej skomplikowanych konstrukcji typu `std::optional` wydaje sie tutaj nie uzasadnione. Funkcja przyjmuje referencje do typów `std::string` jako pierwsze 2 argumenty i typy `int64_t` jako kolejne 2 argumenty, zwraca również `int64_t`

Listing 2: Implementacja funkcji rfind() w C++

```
int64_t custom_rfind(const std::string& str, const std::string& sub,
                     int64_t start = 0, int64_t end = -1)
{
    // Adjust the end parameter
    int64_t str_len = str.length();
    if (end == -1 || end > str_len)
    {
        end = str_len;
    }

    // Adjust negative indices
    if (start < 0)
    {
        start = std::max<int64_t>(0, str_len + start);
    }
    if (end < 0)
    {
        end = std::max<int64_t>(0, str_len + end);
    }

    // Check if search space is valid
    int64_t search_space_len = end - start;
    if (search_space_len < 0)
    {
        return -1;
    }

    // Check if the substring is longer than the search space
    int64_t sub_len = sub.length();
    if (sub_len == 0)
    {
        return end;
    }
    if (sub_len > search_space_len)
    {
        return -1;
    }

    // Iterate over the string in reverse within the specified range
    for (int64_t i = end - sub_len; i >= start; --i)
    {
        if (str.substr(i, sub_len) == sub)
        {
            return i;
        }
    }
    return -1;
}
```

4 Generowanie danych

Dane testowe zostały wygenerowane przy pomocy skryptu napisanego w Pythonie. Skrypt tworzy zestaw unikatowych próbek danych, generujac losowe ciagi

znaków i odpowiadajace im podciagi, które moga być używane do testowania algorytmu wyszukiwania. Działa poprzez losowe ustalanie długości ciagu głównego (z zakresu od 100 do 1000 znaków) i podciagu (od 1 do 3 znaków), a nastepnie generowanie tych ciagów za pomoca zdefiniowanej funkcji generate_random_string, która tworzy losowe ciagi składajace sie z liter i cyfr. W niektórych przypadkach, z prawdopodobieństwem równym 0.1, do próbki dodawane sa również losowo wybrane ograniczenia zakresu przeszukiwania (start i end), co ma na celu przetestowanie działania algorytmu wyszukiwania w różnych warunkach. Wygenerowane próbki sa nastepnie zapisywane do pliku "input.txt", przy czym skrypt dba o to, by w pliku znajdowały sie tylko unikatowe wpisy. Generowane jest 10000 próbek.

Listing 3: Skrypt generujacy dane testowe

```
def generate_random_string(length):
    """Generate a random string of specified length."""
    return ''.join(random.choices(
        string.ascii_letters + string.digits, k=length))

if __name__ == "__main__":
    num_samples = 10000
    search_space_restriction_probability = 0.1
    max_substring_length = 3
    filename = "input.txt"

    samples = []

    for _ in range(num_samples):
        string_length = random.randint(100, 1000)
        substring_length = random.randint(1, max_substring_length)
        main_string = generate_random_string(string_length)
        search_string = generate_random_string(substring_length)

        if random.random() < search_space_restriction_probability:
            end = random.randint(1, substring_length)
            start = random.randint(0, end)
            samples.append(f"\"{main_string}\",
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\"{search_string}\",{start},{end}\n")
        else:
            samples.append(f"\"{main_string}\",\"{search_string}\"\n")

    with open(filename, 'w') as f:
        for sample in list(set(samples)):
            f.write(sample)
```

5 Test poprawności

Poprawność własnych implementacji funkcji przetestowano poprzez wczytanie wygenerowanych danych z pliku, zapisanie wników działania każdej z funkcji do osobnego pliku, a nastepnie porównanie zawartości każdego z plików.

Listing 4: Kod zapisujacy wyniki działania funkcji w Pythonie

```
def check_correctness(input_filename, output_filename, function):
```

```
    data = load_data(input_filename)
    with open(output_filename, 'w') as output_file:
        for str1, str2, start, end in data:
            ret = function(str1, str2, start, end)
            output_file.write(str(ret) + '\n')
```

Listing 5: Kod porównujacy zawartość plików wynikowych

```
with open(custom_output_filename, 'rb') as f1,
     open(builtin_output_filename, 'rb') as f2,
     open(cpp_output_filename, 'rb') as f3:
    content1 = f1.read()
    content2 = f2.read()
    content3 = f3.read()

    if content1 == content2 == content3:
        print('All␣files␣have␣exactly␣the␣same␣contents')
    else:
        print('All␣files␣DO␣NOT␣have␣exactly␣the␣same␣contents')
```

6 Pomiary czasu

Do pomiaru czasu wykorzystano funkcje `time.perf_counter_ns()` w Pythonie oraz klase `std::chrono::high_resolution_clock` w C++.

Rozdzielczość zegara w Pythonie została wyznaczona na 100ns według kodu zawartego w PEP 564.

Listing 6: Skrypt określajacy rozdzielczość pomiaru czasu w Pythonie

```
def get_clock_precision():
    LOOPS = 10 ** 6
    min_dt = [abs(time.perf_counter_ns() - time.perf_counter_ns())
              for _ in range(LOOPS)]
    min_dt = min(filter(bool, min_dt))
    return min_dt
```

Natomiast w C++ rozdzielczość zegara wyznaczono na 1ns w nastepujacy sposób :

Listing 7: Kod określajacy rozdzielczość pomiaru czasu w C++

```
double clock_precision_ns = static_cast<double>(
    std::chrono::high_resolution_clock::period::num) /
    std::chrono::high_resolution_clock::period::den * 1e9;
```

W obu jezykach czas mierzony jest w nanosekundach i zapisywany w typie całkowitoliczbowym, aby uniknać utraty precyzji spowodowanej przez użycie typów zmiennoprzecikowych.

7 Test wydajności

Czas wykonania testowanych funkcji jest mierzony poprzez wykonanie każdej z nich N razy dla różnych zestawów danych wejściowych. Na poczatku i na

końcu każdego zestawu testów rejestrowany jest precyzyjny czas za pomoca zegara o wysokiej rozdzielczości. Różnica miedzy czasem zakończenia a czasem rozpoczecia pozwala na obliczenie całkowitego czasu wykonania dla danego zestawu. Sumujac czasy wykonania wszystkich zestawów testowych, otrzymuje sie łaczny czas wykonania funkcji w nanosekundach. Dane wejściowe zostały wcześniej wczytane i mierzony jest jedynie czas wykonania funkcji.

Listing 8: Pomiar czasu wykonania w Pythonie

```
def get_execution_time(n, data, function):
    total_duration_ns = 0
    for _ in range(n):
        for str1, str2, start, end in data:
            start_time = time.perf_counter_ns()
            ret = function(str1, str2, start, end)
            end_time = time.perf_counter_ns()
            total_duration_ns += (end_time - start_time)
    return total_duration_ns
```

Listing 9: Pomiar czasu wykonania w C++

```
int64_t get_execution_time(int n, const std::vector<std::tuple<
    std::string, std::string, int64_t, int64_t>> data)
{
    // Use an integer to count the total duration in nanoseconds
    int64_t total_duration_ns = 0;
    for (const auto& [str1, str2, start, end] : data) {
        auto start_time = std::chrono::high_resolution_clock::now();

        for (int i = 0; i < n; i++)
        {
            int64_t ret = custom_rfind(str1, str2, start, end);
        }

        auto stop_time = std::chrono::high_resolution_clock::now();
        // Calculate the duration of this execution in nanoseconds
        auto duration_ns = std::chrono::duration_cast
            <std::chrono::nanoseconds>(stop_time - start_time).count();
        total_duration_ns += duration_ns;
    }

    return total_duration_ns;
}
```

Najniższe czasy wykonania osiagneła funkcja wbudowana. Implementacja w C++ była od niej około 14 razy wolniejsza. Implementacja w Pythonie była około 5 razy wolniejsza od implementacji w C++.

Tabela 1: Wyniki testów wydajności

Funkcja	**N**	**Czas wykonania (ns)**	**Błąd względny**	
Oryginalna funkcja	1	13 191 900	7.580	$\times 10^{-2}$
	10	95 603 400	1.045	$\times 10^{-2}$
	100	931 418 000	1.073	$\times 10^{-3}$
	1000	9 708 018 000	1.030	$\times 10^{-4}$
Własna implementacja w Pythonie	1	892 755 200	1.120	$\times 10^{-3}$
	10	6 552 441 000	1.526	$\times 10^{-4}$
	100	61 508 305 200	1.625	$\times 10^{-5}$
	1000	583 622 966 100	1.713	$\times 10^{-6}$
Własna implementacja w C++	1	150 179 700	6.658	$\times 10^{-5}$
	10	1 287 328 300	7.768	$\times 10^{-6}$
	100	13 134 006 100	7.613	$\times 10^{-7}$
	1000	147 745 955 900	6.768	$\times 10^{-8}$