# Opis problemu

Dla głównego zadania projektu program musi umieć wczytywać i wypisywać dane z plików tekstowych. Dane wejściowe są podane w postaci tablicy i przedstawiają sobą ciąg zawierający wyłącznie wartości 0 lub 1. Główna funkcja musi odnaleźć wszystkie najdłuższe podciągi, zawierające równą liczbę zer i jedynek.

Teraz już możemy określić najważniejsze zadania naszego algorytmu. Zaczniemy od początku - wczytywania danych. Skąd praktycznie nic nie wiemy o tym jak będzie wyglądał plik wejściowy, to musimy wykorzystać taki algorytm, żeby mógł wczytywać dane w poprawny sposób niezależnie od tego jak oni są zapisane. Na przykład, są dane trzy ciągi: „0,0,1,0,1,0,0”, „0010100” i „0 0 1 0 1 0 0”. Poprawny algorytm musze rozumieć, że oni są identyczne. Także prawdopodobnym wydaje się fakt, że będziemy chcieli używać tego algorytmu do wielu ciągów jednocześnie. Dla tego program, który mógł by wczytywać dużo tablic z jednego pliku byłby bardziej użyteczny.

Kolejnym zadaniem programu jest wyeliminowanie możliwości najprostszych przypadków, kiedy w podanym ciągu:

* jest równa ilość zer i jedynek;
* są wyłącznie zera lub jedynki.

Tylko po takim wyeliminowaniu możemy już zaczynać wyszukiwać podciągi.

I na koniec dobrze by było, gdyby nasz program był w stanie poinformować użytkownika o takich danych jak: jaki plik program próbował odczytać, w którym zapisał wyniki, ile ciągów znalazł w pliku wejściowym oraz ile czasu zajęło jego działanie.

Podsumowując, nasz algorytm musi być w stanie:

1. odczytać wejściowy plik tekstowy niezależnie od formy zapisu danych;
2. odczytywać dużą liczbę ciągów z jednego pliku;
3. wyeliminować najprostsze przypadki;
4. wyszukać najdłuższe podciągi w najbardziej szybki sposób;
5. poinformować użytkownika o przydatnych rzeczach.

W następnym podrozdziale wyjaśnię bardziej szczegółowo, w jaki sposób mój program wykonuje każdy z wcześniej wymienionych punktów.

# Analiza algorytmu

W danym rozdziale chciałbym szczegółowo wyjaśnić każdy ułamek mojego algorytmu.

## Wczytywanie ciągów z plików

Już wcześniej zaznaczyłem, że poprawny algorytm musi być w stanie wczytywać dane w poprawny sposób niezależnie od ich formy zapisu. Dla tego myślę, że najprostszym rozwiązaniem takiego problemu byłoby sprawdzanie czy każdy element jest jedynką lub zerem. Oczywiście taki sposób będzie działać troszeczku dłużej, niż identyczny sposób bez sprawdzania każdego elementu, ale skąd działanie całego programu zależy od poprawności tej funkcji, to już lepiej zostawić tutaj taki algorytm, który będę dawał poprawny wynik w największej ilości przypadków, zwłaszcza biorąc pod uwagę, że jego czas wykonania zajmuje mniej niż 2 procent czasu wykonania całego algorytmu.

Pseudokod działania takiego algorytmu, a oraz i schemat blokowy znajdują się poniżej. W kodzie źródłowym funkcja, która wykonuje te działania, nazywa się reading\_file i znajduje się w klasie binary\_sequences.

Funkcja 1: Wczytywanie ciągów z plików

Pseudokod:

1. Utwórz nowy plik w określonej ścieżce path\_out
2. all\_sequences= [ ]
3. accepted\_values = ['0', '1']
4. Otwórz plik w określonej ścieżce path\_in jako file
5. Dla *każdego wiersza line w pliku file* wykonuj:
6. | Podzielić line na słowa względem spacji
7. | Usuń niepotrzebne spacje z wierszu line
8. | this\_sequence = [ ]
9. | Dla *każdego słowa word w wierszu line* wykonuj:
10. | | Dla *każdego symbolu symbol w słowie word* wykonuj:
11. | | Jeśli *symbol należy do accepted\_values*, to
12. | | | Dodaj int(symbol) do tablicy this\_sequence
13. | Jeśli *this\_sequence ≠ [ ],* to
14. | | Dodaj this\_sequence do tablicy all\_sequences
15. Zakończ

## Wyeliminowanie najprostszych przypadków

Żeby wyeliminować takie ciągi, w których już od początku niema zer lub jedynek, a także ciągi, w których liczba zer i jedynek jest równa, potrzebujemy wiedzieć, ile jest jedynek i zer w ciągu. Poniżej znajdują się pseudokod i schemat blokowy funkcji, która wykonuje powyżej opisane działania. W kodzie źródłowym funkcja, która wykonuje te działania, nazywa się solve\_problem i znajduje się w klasie binary\_sequences.

Funkcja 2: Wyeliminowanie najprostszych przypadków

Pseudokod:

1. iterator = 0
2. time\_results = [ ]
3. number\_of\_sequences = długość (all\_sequences)
4. Dopóki *iterator < number\_of\_sequences* wykonuj:
5. | current\_sequence = all\_sequences [ iterator ]
6. | iterator += 1
7. | k = zlicz ilość wystąpień jedynki w tablice current\_sequence
8. | n = zlicz ilość wystąpień zer w tablice current\_sequence
9. | p = minimum wśród (k, n)
10. | Jeżeli *n = k i k ≠ 0*, to:
11. | | Odpowiedź: najdłuższy podciąg - to wejściowy ciąg
12. | W przeciwnym razie, jeśli *n = 0 lub k = 0*, to:
13. | | Odpowiedź: Niema podciągu spełniającego wymagania zadania
14. | W przeciwnym razie, jeśli *n >= 1 i k >= 1*, to:
15. | | start\_time = czas teraźniejszy
16. | | substring = znajdź podciągi
17. | | end\_time = czas teraźniejszy
18. | | dodaj (end\_time - start\_time) do tablicy time\_results

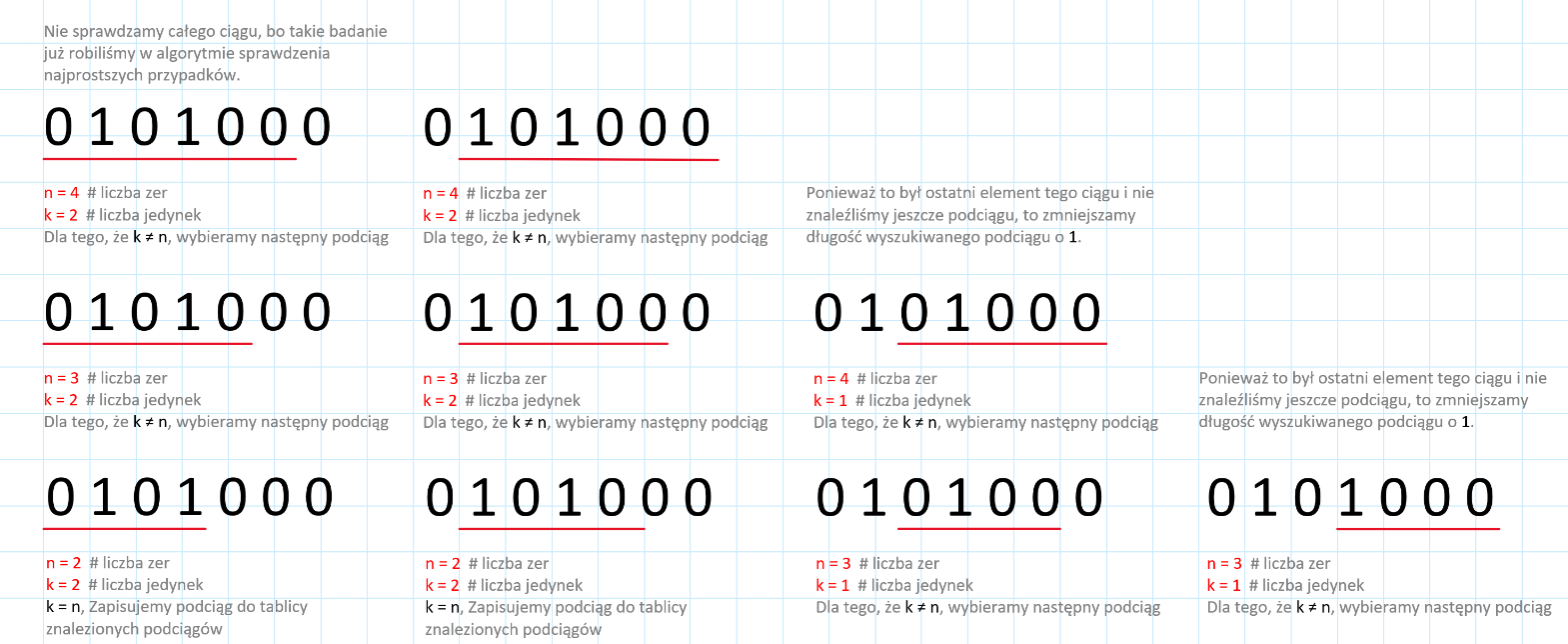
## Główny algorytm wyszukiwania podciągów

Starałem się maksymalnie zoptymalizować algorytm, więc teraz mam kilka wersji mojego algorytmu.

### Podstawowy algorytm wyszukiwania podciągów

Nasz algorytm ma znaleźć różne najdłuższe podciągi, w których liczba zer i jedynek jest równa. Dla tego możemy sprawdzać wszystkie możliwe podciągi licząc ilości zer i jedynek w każdym. Skąd potrzebujemy odszukać najdłuższy podciąg, to będziemy zaczynać od najdłuższego i iść do najkrótszego możliwego podciągu, żeby nie sprawdzać zbędne podciągi.

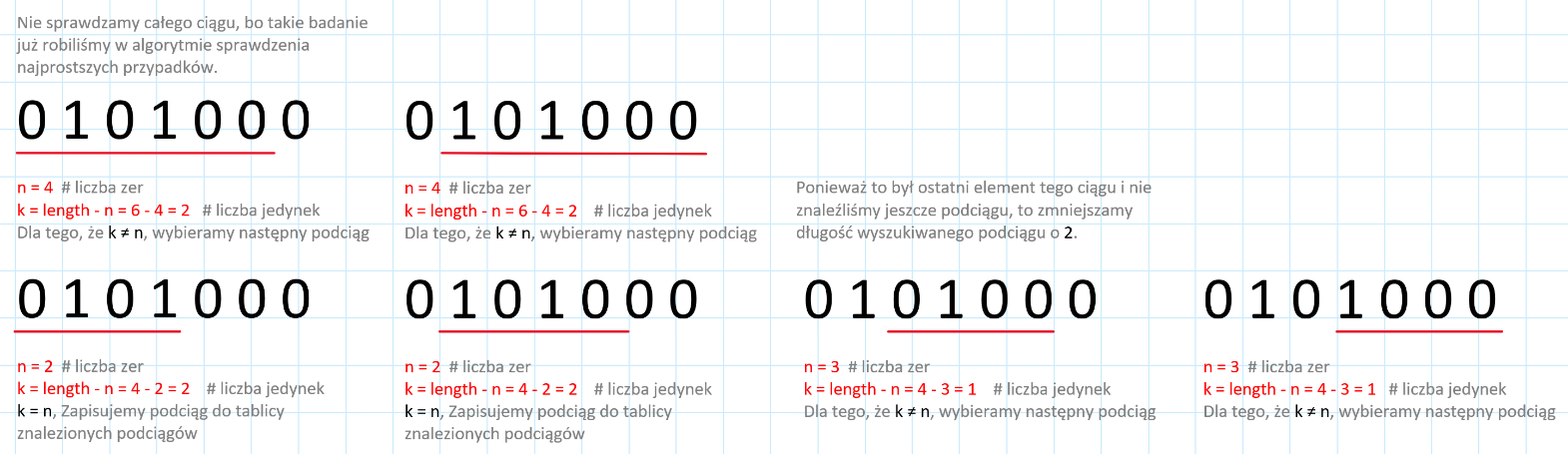
Na przykład bierzemy ciąg:

Oczywistym jest fakt, że ma dwa najdłuższe podciągi:

oraz

Na schemacie działania mojego algorytmu widać, że program bada każdy podciąg. Tylko wtedy, kiedy ilość zer i jedynek jest równa, algorytm dopisuje podciąg do tablicy wyników. Skąd funkcja #2 eliminuje możliwość nieistnienia podciągu, to taki algorytm zawsze potrafi znaleźć podciąg.

Wiemy, że liczba wystąpień jedynek ma być równą liczbie wystąpień zer. Czyli długość całego podciągu:

To znaczy, że możemy sprawdzać tylko parzyste długości podciągów, skracając w ten sposób czas potrzebny do wykonania programu. Skąd będziemy mieć tylko parzyste podciągi, to możemy zliczać tylko ilość któregoś jednego z elementów (0, 1) i porównywać ją z połową długości podciągu, co także zmniejsza złożoność algorytmu. Poszczególne etapy realizacji takiego algorytmu przedstawiono na poniższym schemacie.

### Optymalizowanie algorytmu wyszukiwania podciągów

Możemy zacząć od tego, że po prostu nie musimy sprawdzać wszystkich podciągów, a tylko te, których długość jest mniejsza lub równa dwukrotnej liczbie elementów, których jest mniej w ciągu. Czyli będziemy sprawdzać tylko te podciągi, długość których jest teoretycznie możliwa. Na przykład, w już wcześniej wspominanym ciągu maksymalna teoretyczna długość podciągu będzie zależała od ilości jedynek dlatego, że ich jest mniej. Czyli:

Weźmy na przykład inny ciąg, długość którego posiada 20 symboli:

W tym ciągu liczba zer , a liczba jedynek . Nasz algorytm już wie, że maksymalna teoretyczna długość podciągu wynosi:

Chociaż zaczynamy szukać podciągów o długości co najmniej 6 znaków, nadal wykonujemy wiele zbędnych obliczeń. Na przykład na tym schemacie widać, że co najmniej trzy razy obliczamy ilość jedynek w zielonym prostokącie chociaż od początku wiemy, że cztery jedynki nie mogą znajdować się w podciągu z sześciu elementów. Moim zdaniem w najlepszy sposób rozwiązuje ten problem następne ulepszenie: kiedy wiemy dokładnie ilość zer i jedynek w złym podciągu, to możemy zaprognozować, gdzie może znajdować się najbliższy poprawny podciąg za pomocy następnej formuły:

gdzie difference – to odległość pomiędzy początkami aktualnego i następnego podciągów. Inaczej mówiąc, gdy w aktualnym podciągu jest na elementów więcej niż powinno być, to znaczy, że musimy przenieść się przynajmniej na elementów po ciągu, żeby uzupełnić warunek równości zer a jedynek. Schemat działania kolejnego ulepszonego algorytmu pokazano na \_. Schemat blokowy i pseudokod tego algorytmu podaję poniżej.

Ostatnia optymalizacja której możemy dokonać – to wykorzystywania funkcja natywna