**P O L I T E C H N I K A R Z E S Z O W S K A**

**im. Ignacego Łukasiewicza**

**WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ**

Projekt z algorytmów i struktur danych  
studenta pierwszego roku studiów  
kierunku Inżynieria i analiza danych

Zero a Jeden 1.0

**Vitalii Morskyi**

Spis treści

[Opis problemu 2](#_Toc57411269)

[Analiza algorytmu 2](#_Toc57411270)

[Wczytywanie ciągów z plików 2](#_Toc57411271)

[Wyeliminowanie najprostszych przypadków 4](#_Toc57411272)

[Główny algorytm wyszukiwania podciągów 5](#_Toc57411273)

[Podstawowy algorytm wyszukiwania podciągów 5](#_Toc57411274)

[Optymalizowanie algorytmu wyszukiwania podciągów 8](#_Toc57411275)

[Porównanie algorytmów 13](#_Toc57411276)

[Dokumentacja z doświadczeń 17](#_Toc57411277)

[Wczytywania danych 17](#_Toc57411278)

[Wypisywania wyników 17](#_Toc57411279)

[Wnioski 17](#_Toc57411280)

[Legenda pseudokodów i schematów blokowych 18](#_Toc57411281)

# Opis problemu

W danym rozdziale chciałbym opisać zadanie, które otrzymałem do wykonania.

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. Ponieważ 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 podane trzy ciągi: „0,0,1,0,1,0,0”, „0010100” i „0 0 1 0 1 0 0”. Poprawny algorytm musi rozumieć, że one 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.

Ale oczywiście istnieją przypadki, kiedy chcemy generować dane wejściowe automatycznie. Dla tego byłoby dobrze, gdyby algorytm posiadał opcję automatycznego tworzenia plików wejściowych.

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

* jest równa ilość zer i jedynek;
* są wyłącznie zera lub wyłącznie 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 rozdziale wyjaśnię bardziej szczegółowo, w jaki sposób mój program wykonuje każdy z wcześniej wymienionych punktów.

# Analiza algorytmu

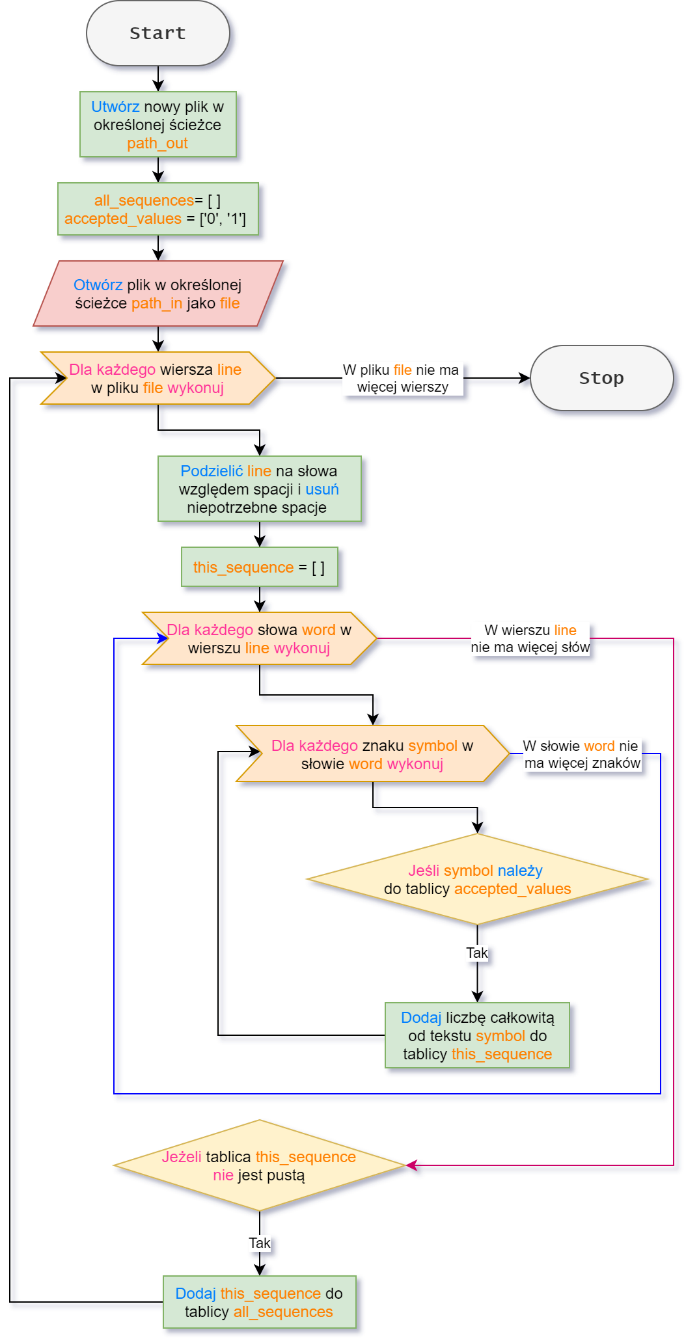
W danym rozdziale chciałbym szczegółowo wyjaśnić każdy ułamek mojego algorytmu.

## 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 dlatego, że działanie całego programu zależy od poprawności tej funkcji, to już lepiej zostawić tutaj taki algorytm, który będzie dawał poprawny wynik w największej ilości przypadków, zwłaszcza że jego czas wykonania zajmuje mniej niż 2 procent czasu wykonania całego algorytmu.

Pseudokod działania takiego algorytmu oraz schemat blokowy (Schemat blokowy 1) znajdują się poniżej. W kodzie źródłowym funkcja, która wykonuje te działania, nazywa się reading\_file i znajduje się w klasie binary\_sequences.

Schemat blokowy funkcji read\_file():



Schemat blokowy 1 read\_file()  
Źródło: opracowanie własne.

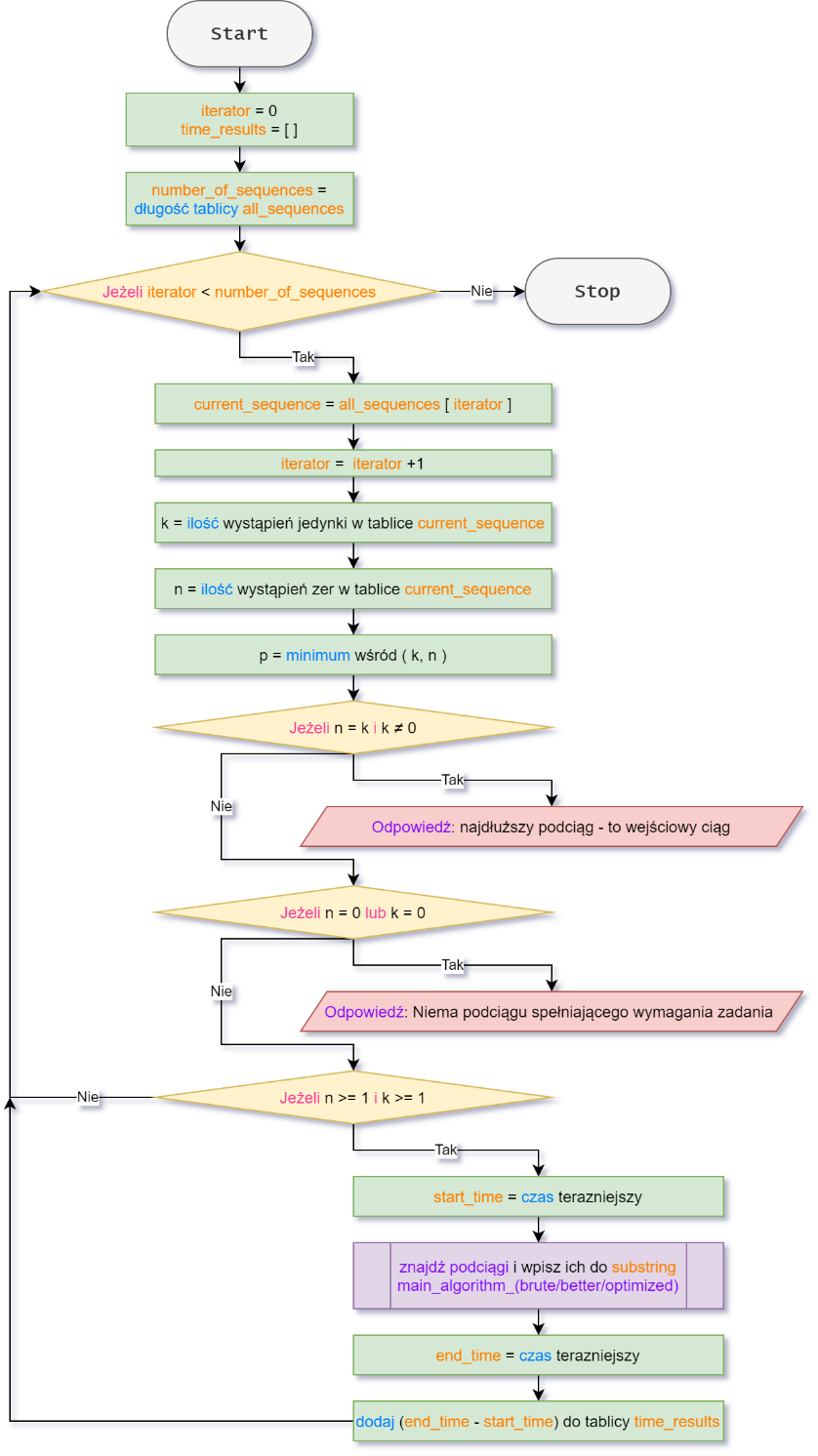
Pseudokod funkcji read\_file():

1. Utwórz nowy plik w określonej ścieżce path\_out // Tworzenie wyjściowego pliku
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 (Schemat blokowy 2) 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.

Schemat blokowy funkcji solve\_problem():



Schemat blokowy 2 solve\_problem()  
Źródło: opracowanie własne.

Pseudokod funkcji solve\_problem():

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 // Funkcja odpowiedź w kodzie źródłowym posiada nazwę give\_answer()
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 = main\_algorithm\_(brute/better/optimized) // Wybieramy algorytm zgodnie z decyzją użytkownika
17. | | end\_time = czas teraźniejszy
18. | | dodaj (end\_time - start\_time) do tablicy time\_results
19. Zakończ

## Główny algorytm wyszukiwania podciągów

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

### Podstawowy algorytm wyszukiwania podciągów

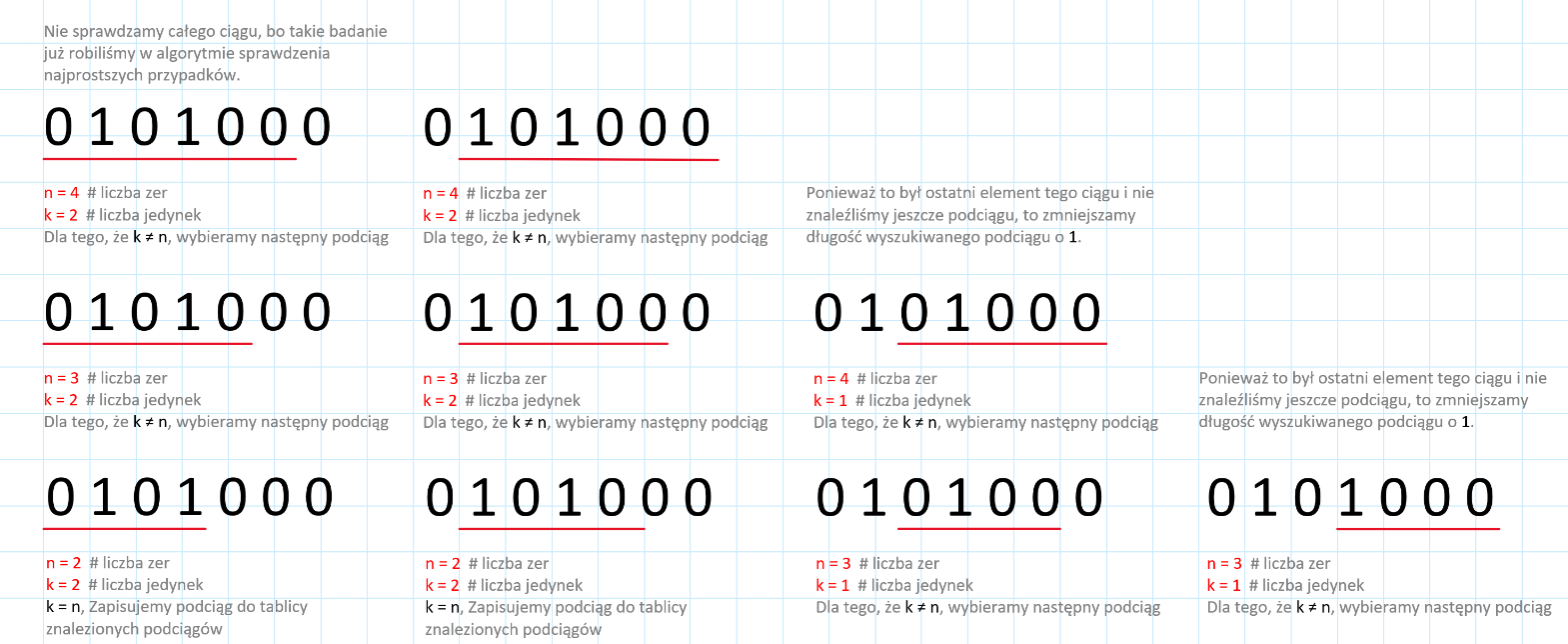
Nasz algorytm ma znaleźć różne najdłuższe podciągi, w których liczba zer i jedynek jest równa. Dlatego możemy sprawdzać wszystkie możliwe podciągi licząc ilości zer i jedynek w każdym. Ponieważ potrzebujemy odszukać najdłuższy podciąg, to będziemy zaczynać od najdłuższego i iść do najkrótszego możliwego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

Na przykład bierzemy ciąg:

Oczywistym jest fakt, że ma dwa najdłuższe podciągi:

oraz

Na schemacie działania mojego algorytmu (Schemat 1) widać, że program bada każdy podciąg. Tylko wtedy, kiedy ilość zer i jedynek jest równa, algorytm dopisuje podciąg do tablicy wyników. Tak jak funkcja #2 eliminuje możliwość nieistnienia podciągu, to taki algorytm zawsze potrafi znaleźć podciąg.

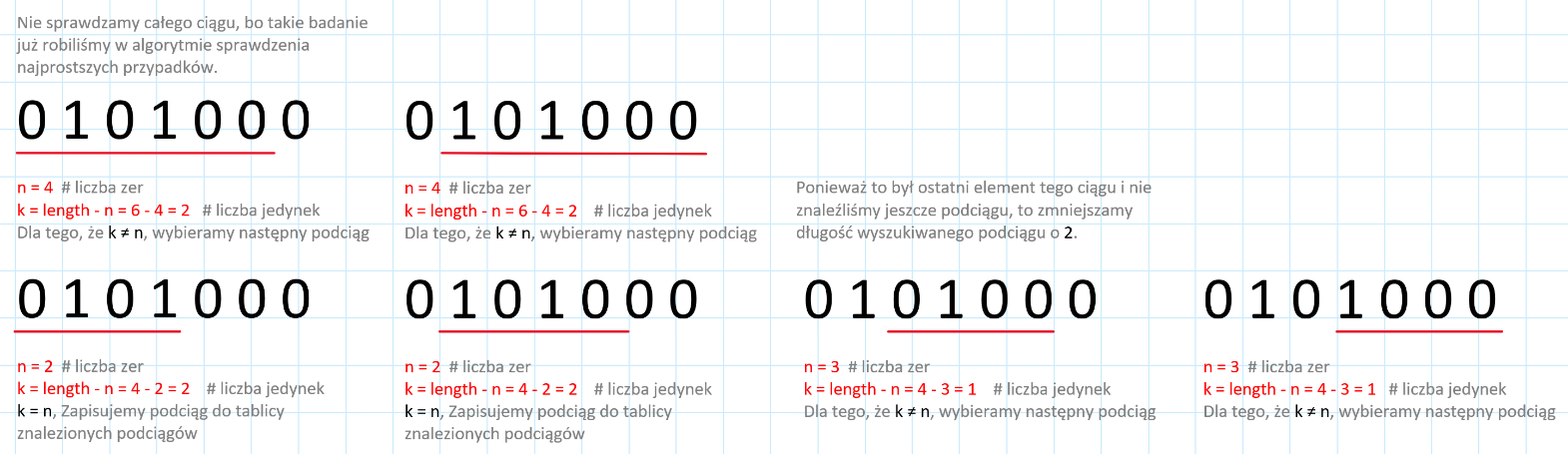


Schemat 1 Kroki działania aktualnego algorytmy dla ciągu 0101000.  
Źródło: opracowanie własne.

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

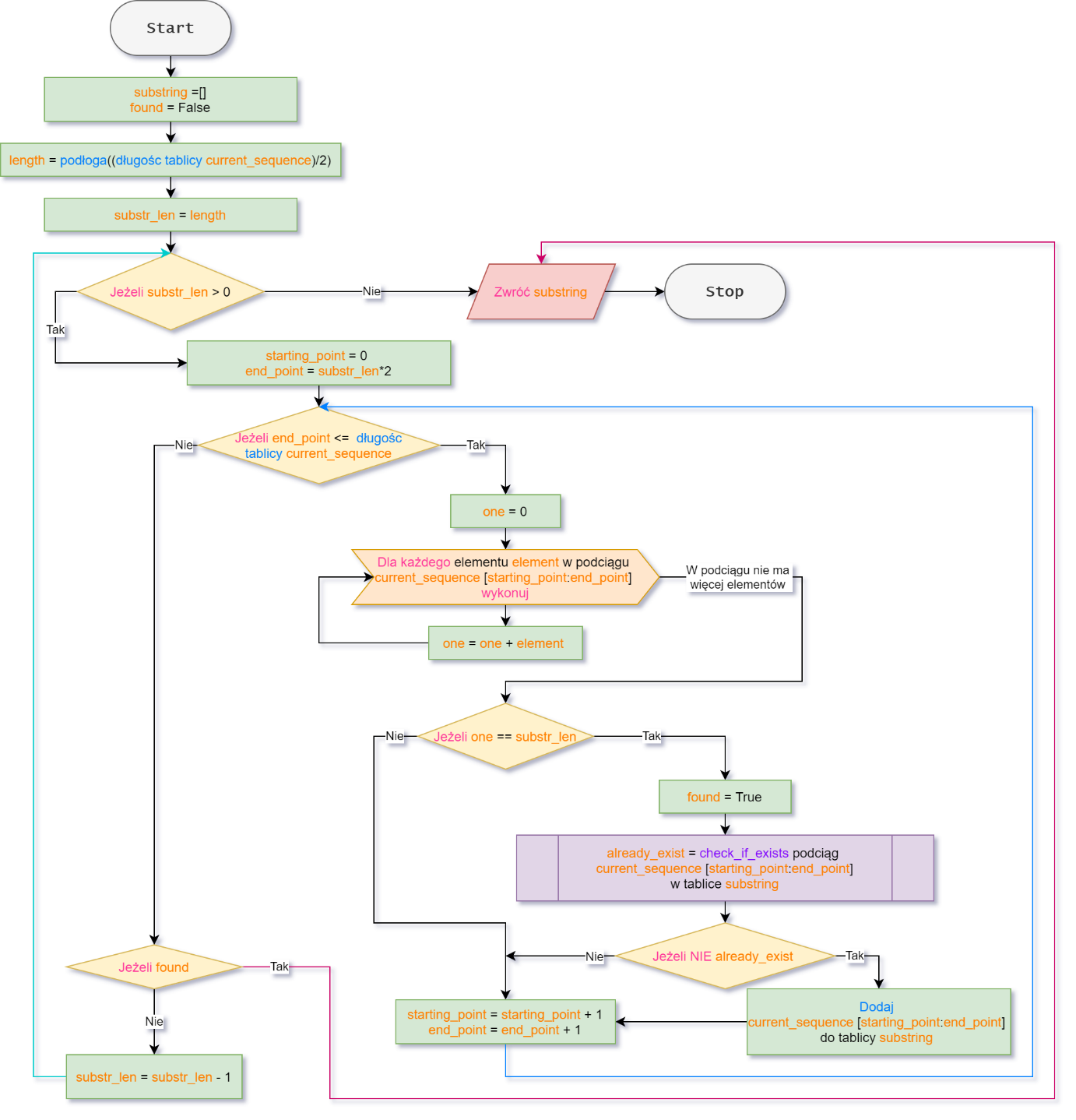
To znaczy, że możemy sprawdzać tylko parzyste długości podciągów, skracając w ten sposób czas potrzebny do wykonania programu. Skoro będziemy mieć tylko parzyste podciągi, to możemy zliczać tylko ilość któregoś jednego z elementów (0, 1) i porównywać ją z połową długości podciągu, co także zmniejsza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

Poszczególne etapy realizacji takiego algorytmu przedstawiono na poniższym schemacie (Schemat 2). Jego pseudokod oraz schemat blokowy (Schemat blokowy 3; Schemat blokowy 4) są podane poniżej; w kodzie źródłowym ten algorytm posiada nazwę main\_algorithm\_brute i znajduje się w klasie binary\_sequences.



Schemat 2 Kroki działania aktualnego algorytmy dla ciągu 0101000.  
Źródło: opracowanie własne.

Schemat blokowy funkcji main\_algorithm\_brute():



Schemat blokowy 3 main\_algorithm\_brute()  
Źródło: opracowanie własne.

Schemat blokowy funkcji check\_if\_exists():



Schemat blokowy 4 check\_if\_exists()  
Źródło: opracowanie własne

Pseudokod funkcji main\_algorithm\_brute():

1. substring =[]
2. found = False
3. length = podłoga((długośc tablicy current\_sequence)/2)
4. substr\_len = length
5. Dopóki substr\_len > 0 wykonuj K06:K24
6. | starting\_point = 0
7. | end\_point = substr\_len\*2
8. | Dopóki end\_point <= długośc tablicy current\_sequence wykonuj K09:K21
9. | | one = 0
10. | | Dla każdego elementu element w podciągu current\_sequence [starting\_point:end\_point] wykonuj K11:
11. | | | one = one + element
12. | | Jeżeli one == substr\_len wykonuj K13:K19
13. | | | found = True
14. | | | already\_exist = False
15. | | | Dla każdego elementu element w tablice substring wykonuj K16:K17 // Ta część pseudokodu (K14:K17) w schemacie blokowym, zarówno jak i w kodzie źródłowym, oznaczona przez funkcję „check\_if\_exists()”, bo powtarza się kilka razy w całym programu.
16. | | | | Jeżeli element == current\_sequence [starting\_point:end\_point] wykonuj K17
17. | | | | | already\_exist = True
18. | | | Jeżeli NIE already\_exist wykonuj K19
19. | | | | Dodaj current\_sequence [starting\_point:end\_point] do tablicy substring
20. | | starting\_point = starting\_point + 1
21. | | end\_point = end\_point + 1
22. | Jeżeli found wykonuj K23
23. | | break
24. | substr\_len = substr\_len – 1
25. Zwróć substring
26. Zakończ

### Optymalizowanie algorytmu wyszukiwania podciągów

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

Weźmiemy 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 (Schemat 3) 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.



Schemat 3 Kroki działania aktualnego algorytmy dla ciągu 10111111111111111010.  
Źródło: opracowanie własne.

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:

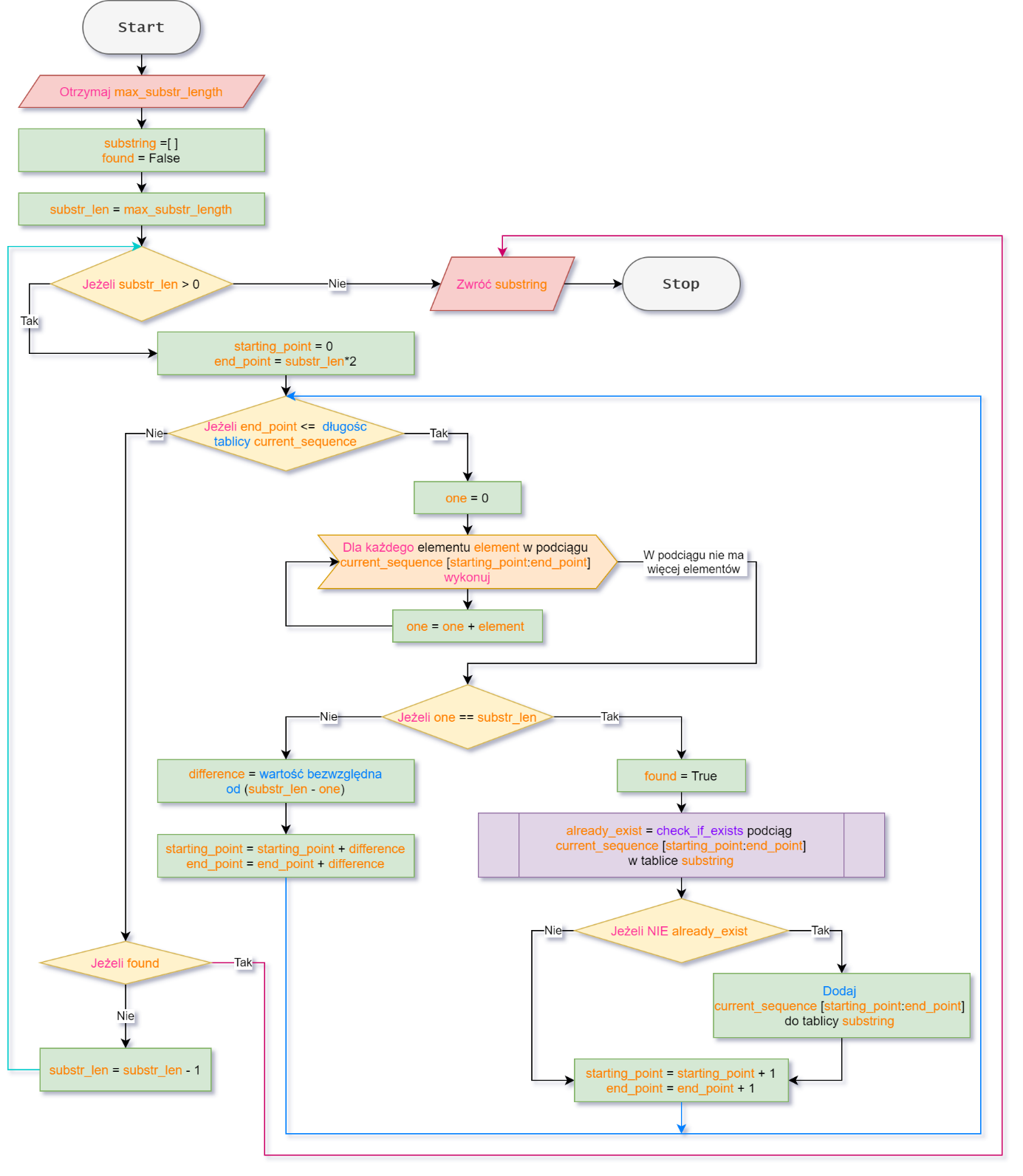


Schemat 4 Kroki działania aktualnego algorytmy dla ciągu 10111111111111111010.  
Źródło: opracowanie własne.

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 pokazany na Schemat 4.

Schemat blokowy (Schemat blokowy 5) i pseudokod tego algorytmu podaję poniżej, w programie ten algorytm nazywa się main\_algorithm\_better i znajduje się w klasie binary\_sequences.

Schemat blokowy funkcji main\_algorithm\_better():



Schemat blokowy 5 main\_algorithm\_better()  
Źródło: opracowanie własne.

Pseudokod funkcji main\_algorithm\_better():

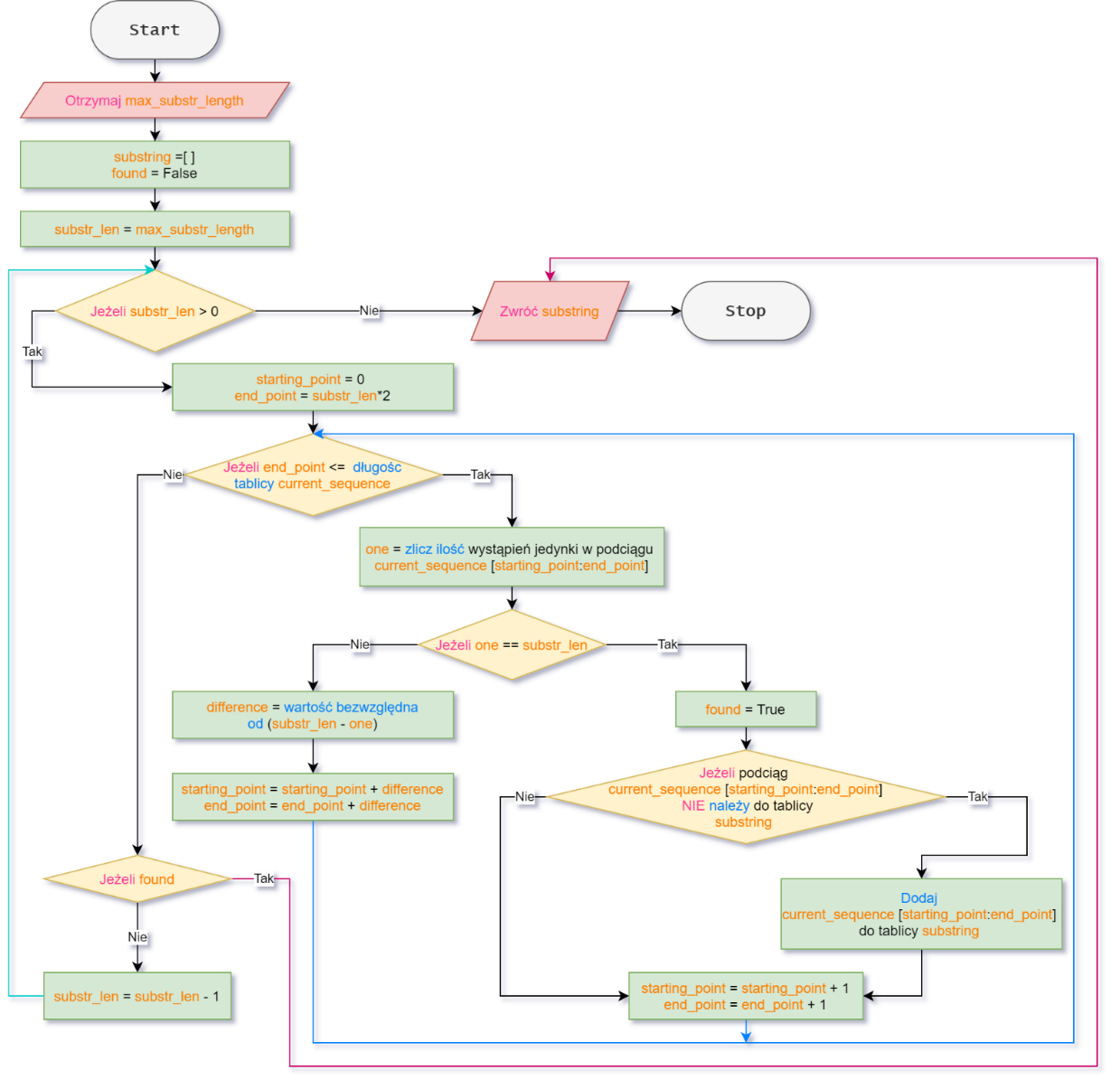
1. Otrzymaj max\_substr\_length // Funkcja przejmuje taki argument
2. substring =[]
3. found = False
4. substr\_len = max\_substr\_length
5. Dopóki substr\_len > 0 wykonuj K06:K28
6. | starting\_point = 0
7. | end\_point = substr\_len\*2
8. | Dopóki end\_point <= długośc tablicy current\_sequence wykonuj K09:K25
9. | | one = 0
10. | | Dla każdego elementu element w podciągu current\_sequence [starting\_point:end\_point] wykonuj K11:
11. | | | one = one + element
12. | | Jeżeli one == substr\_len wykonuj K13:K21
13. | | | found = True
14. | | | already\_exist = False
15. | | | Dla każdego elementu element w tablice substring wykonuj K16:K17 // Ta część pseudokodu (K14:K17) w schemacie blokowym, zarówno jak i w kodzie źródłowym, oznaczona przez funkcję „check\_if\_exists()”, bo powtarza się kilka razy w całym programu.
16. | | | | Jeżeli element == current\_sequence [starting\_point:end\_point] wykonuj K17
17. | | | | | already\_exist = True
18. | | | Jeżeli NIE already\_exist wykonuj K19
19. | | | | Dodaj current\_sequence [starting\_point:end\_point] do tablicy substring
20. | | | starting\_point = starting\_point + 1
21. | | | end\_point = end\_point + 1
22. | | W przeciwnym razie wykonuj K23:K25:
23. | | | difference = abs(substr\_len-one)
24. | | | starting\_point = starting\_point + difference
25. | | | end\_point = end\_point + difference
26. | Jeżeli found wykonuj K27
27. | | break
28. | substr\_len = substr\_len – 1
29. Zwróć substring
30. Zakończ

Ostatnia optymizacja, której możemy dokonać – to wykorzystywanie funkcji natywnych języka programowania dla takich działań jako zliczenie ilości wystąpień elementu w tablice i sprawdzenie czy istnieje element w tablice. Taki algorytm jest ustawiony jako domyślny w moim programie. Jego pseudokod oraz schemat blokowy (Schemat blokowy 6) są podane poniżej, w kodzie źródłowym ten algorytm posiada nazwę main\_algorithm\_optimized() i znajduje się w klasie binary\_sequences.

Pseudokod funkcji main\_algorithm\_optimized():

1. Otrzymaj max\_substr\_length // Funkcja przejmuje taki argument
2. substring =[]
3. found = False
4. substr\_len = max\_substr\_length
5. Dopóki substr\_len > 0 wykonuj K06:K22
6. | starting\_point = 0
7. | end\_point = substr\_len\*2
8. | Dopóki end\_point <= długośc tablicy current\_sequence wykonuj K09:K15
9. | | one=zlicz ilość wystąpień jedynki w podciągu current\_sequence[starting\_point:end\_point]
10. | | Jeżeli one == substr\_len wykonuj K11:K16
11. | | | found = True
12. | | | Jeżeli podciąg current\_sequence[starting\_point:end\_point] NIE należy do tablicy substring wykonuj K13
13. | | | | Dodaj current\_sequence[starting\_point:end\_point]do tablicy substring
14. | | | starting\_point = starting\_point + 1
15. | | | end\_point = end\_point + 1
16. | | W przeciwnym razie wykonuj K17:K19:
17. | | | difference = abs(substr\_len-one)
18. | | | starting\_point = starting\_point + difference
19. | | | end\_point = end\_point + difference
20. | Jeżeli found wykonuj K21
21. | | break
22. | substr\_len = substr\_len – 1
23. Zwróć substring
24. Zakończ

Schemat blokowy funkcji main\_algorithm\_optimized():



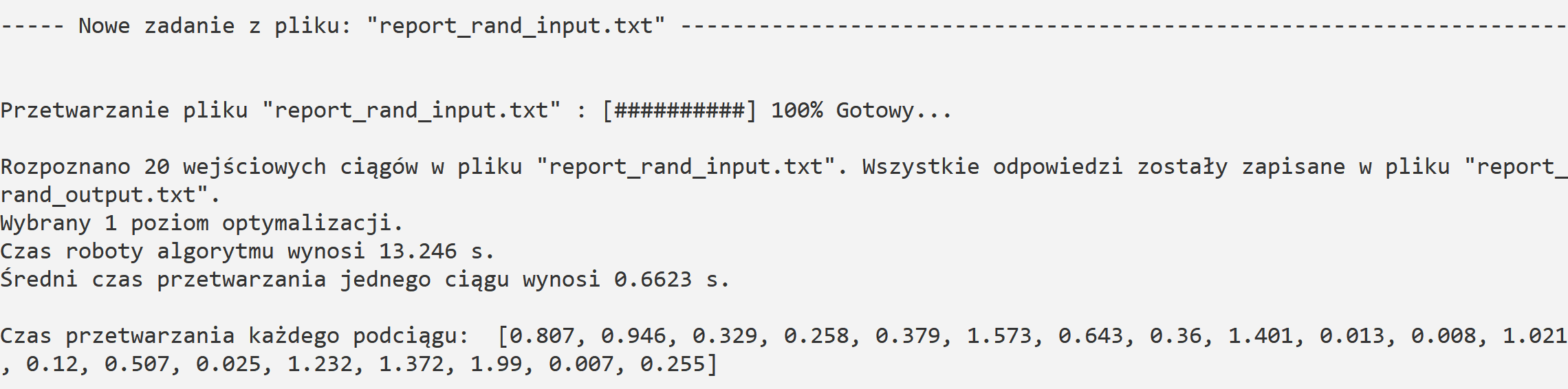
Schemat blokowy 6 main\_algorithm\_optimized()  
Źródło: opracowanie własne.

# Porównanie algorytmów

Skoro już stworzyliśmy aż 3 algorytmy i twierdzimy, że każdy kolejny jest lepszy od poprzedniego, to musimy jakoś to udowodnić. Aby to zrobić, utworzymy plik wejściowy z dużą liczbą znaków. Niech zawiera 20 losowych ciągów zer i jedynek, z których każdy będzie miał 1000 znaków. Aby to zrobić, użyjemy funkcji random\_sequence(), którą stworzyłem, aby generować losowe sekwencje zer i jedynek. Znajduje się ona w pliku pomocniczym assistant\_module.py. Nazwijmy nowo utworzony plik „report\_rand\_input.txt”. Wynik działania naszego algorytmu zostanie zapisany w pliku „report\_rand\_output.txt”.

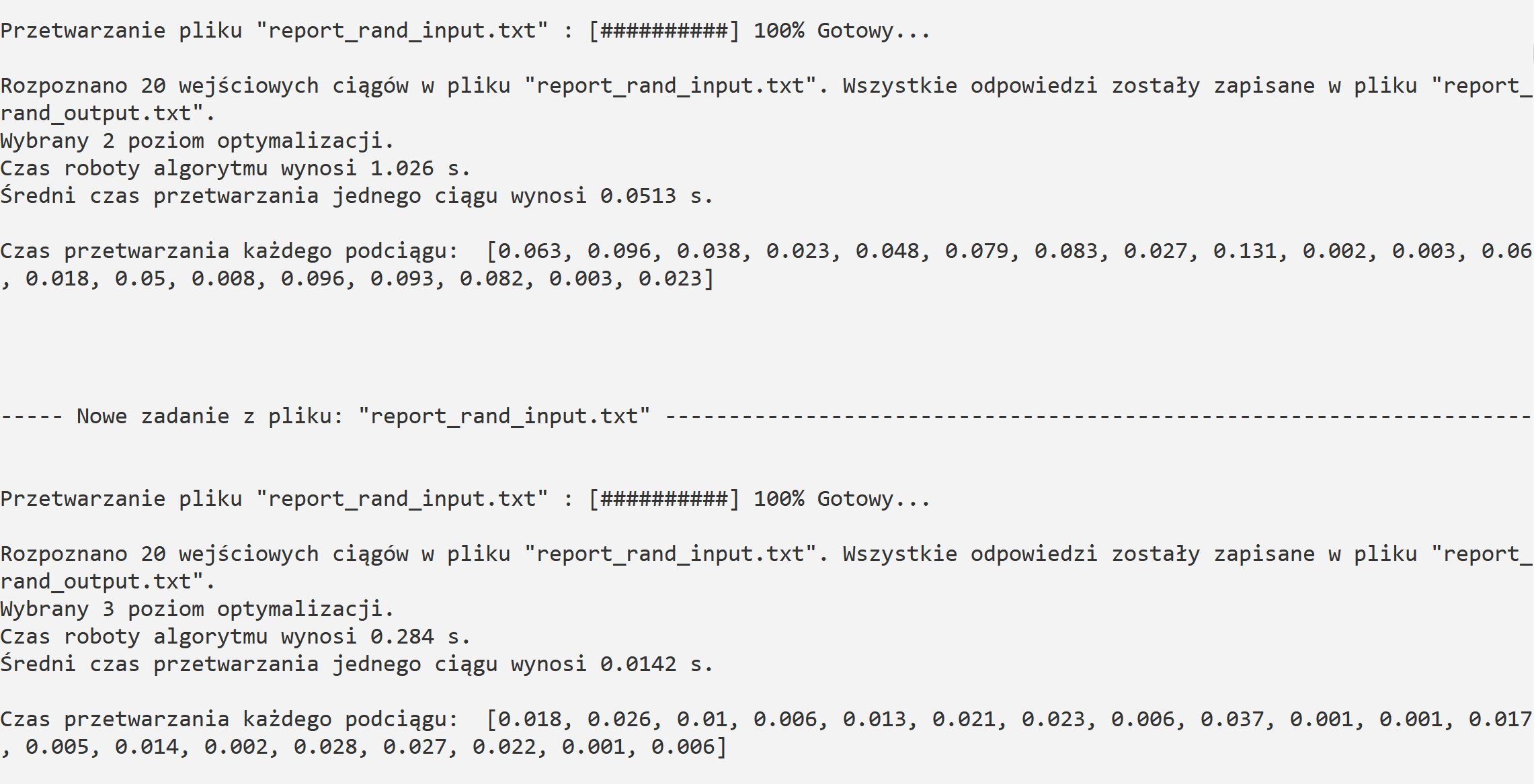
Zgodnie z przykładem podanym w funkcji main() utworzymy obiekt klasy „binary\_sequences” i nazwiemy go „test\_1”. Przypisując odpowiednie wartości do flag, możemy skorzystać z funkcji solve\_problem(), która znajdzie wszystkie podciągi i zapisze wyniki do plików źródłowych. Ponieważ mamy zamiar porównać nasze algorytmy, potrzebujemy informacji o wynikach czasu. Aby to otrzymać, używamy funkcji give\_time().

Po uruchomieniu programu otrzymujemy wyniki działania najmniej optymalizowanego algorytmu, podobne do tych, które są na Rysunek 1.



Rysunek 1 Wynik w konsoli po uruchomieniu najmniej optymalizowanego algorytmu.  
Źródło: opracowanie własne.

Widać, że mój komputer wykonał to zadanie w 13 sekund. Ale do porównania algorytmów potrzebujemy wyników co najmniej dwóch algorytmów. Aby uzyskać wyniki drugiego algorytmu zmienimy flagę wyboru algorytmu i ponownie rozpoczniemy wyszukiwanie podciągów za pomocą funkcji solve\_problem(). Aby nie pisać tych samych trzech wierszy po raz trzeci, wyniki trzeciego algorytmu możemy uzyskać za pomocą funkcji test().

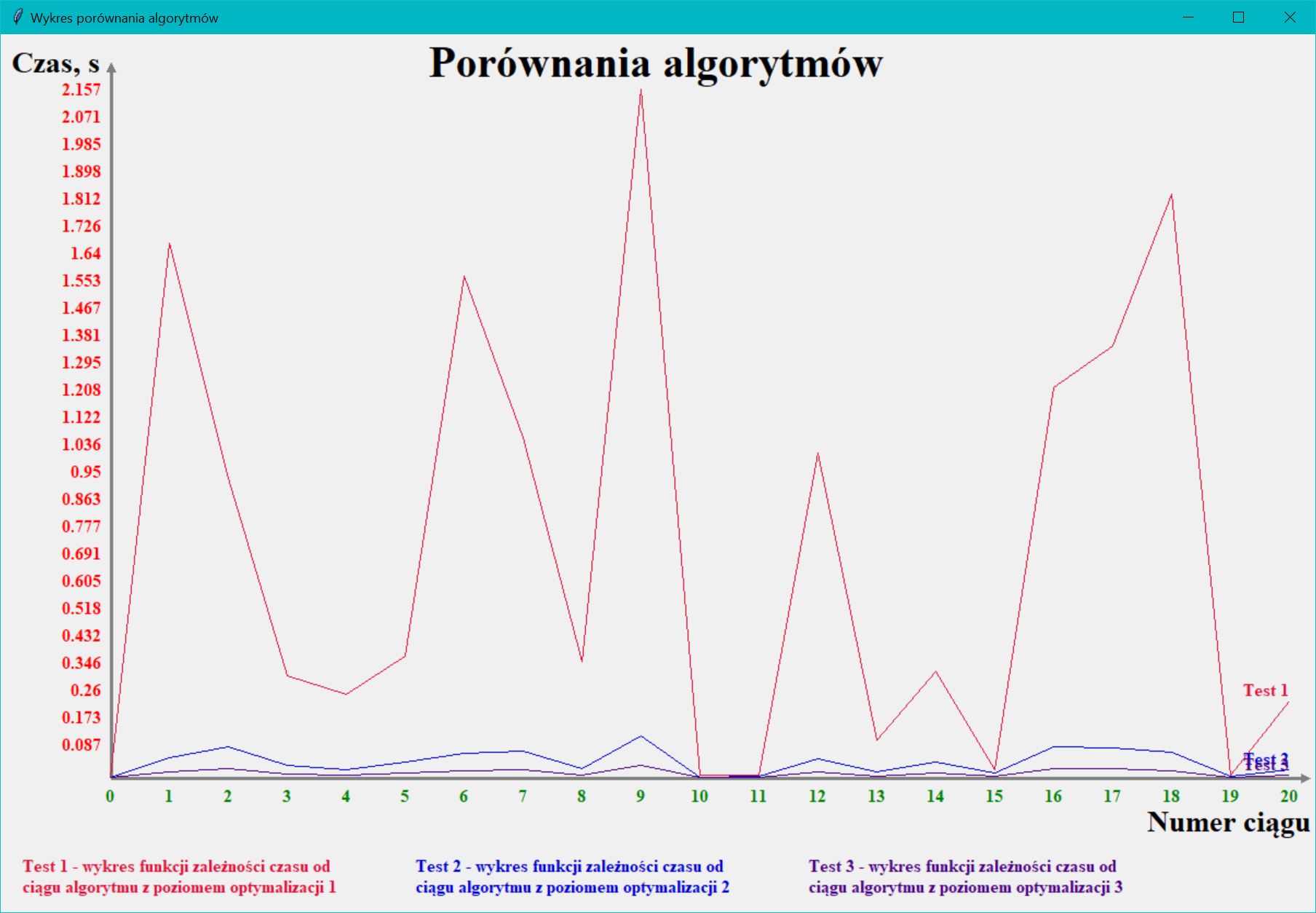


Rysunek 2 Wynik w konsoli po uruchomieniu drugiego i trzeciego algorytmów.  
Źródło: opracowanie własne.

Po uruchomieniu programu otrzymujemy wyniki działania trzech algorytmów. Wyniki drugiego i trzeciego algorytmów są podane na Rysunek 2.

Czasy wykonania drugiego i trzeciego algorytmu wynoszą odpowiednio 1 i 0,3 sekundy. W porównaniu do pierwszych 13 sekund widzimy globalną poprawę, więc zoptymalizowane algorytmy działają naprawdę szybciej. Teraz interesuje nas pytanie, o ile szybciej?

W rzeczywistości znacznie przyjemniej byłoby spojrzeć na wyniki jako wykres funkcji osi czasu. Aby to zrobić, możemy skorzystać z funkcji algorytm\_comparison(), którą napisałem specjalnie do takich celów używając modułu „tkinter”.

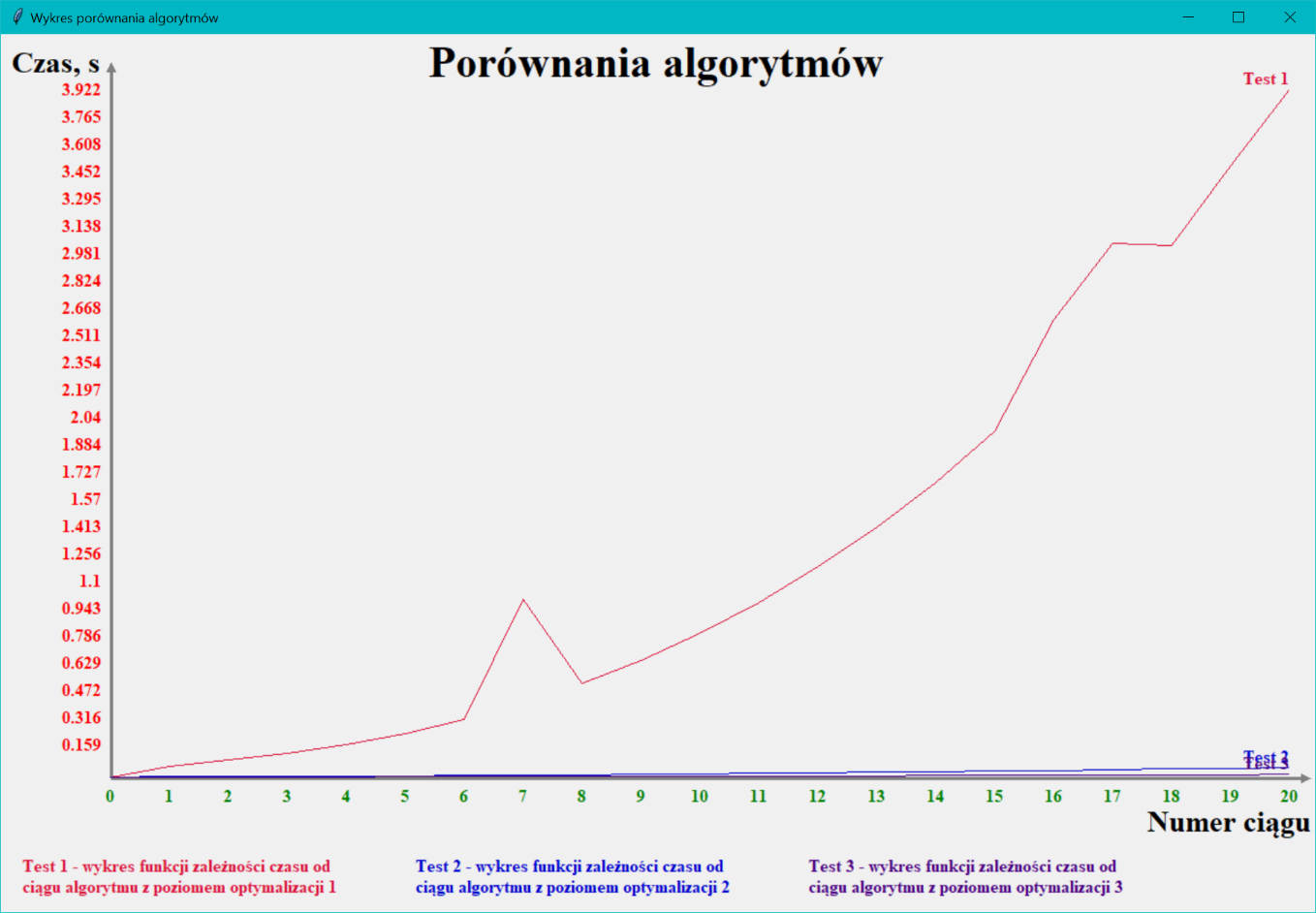


Wykres 1 Porównanie trzech algorytmów w trybie losowych danych wejściowych.  
Źródło: opracowanie własne.

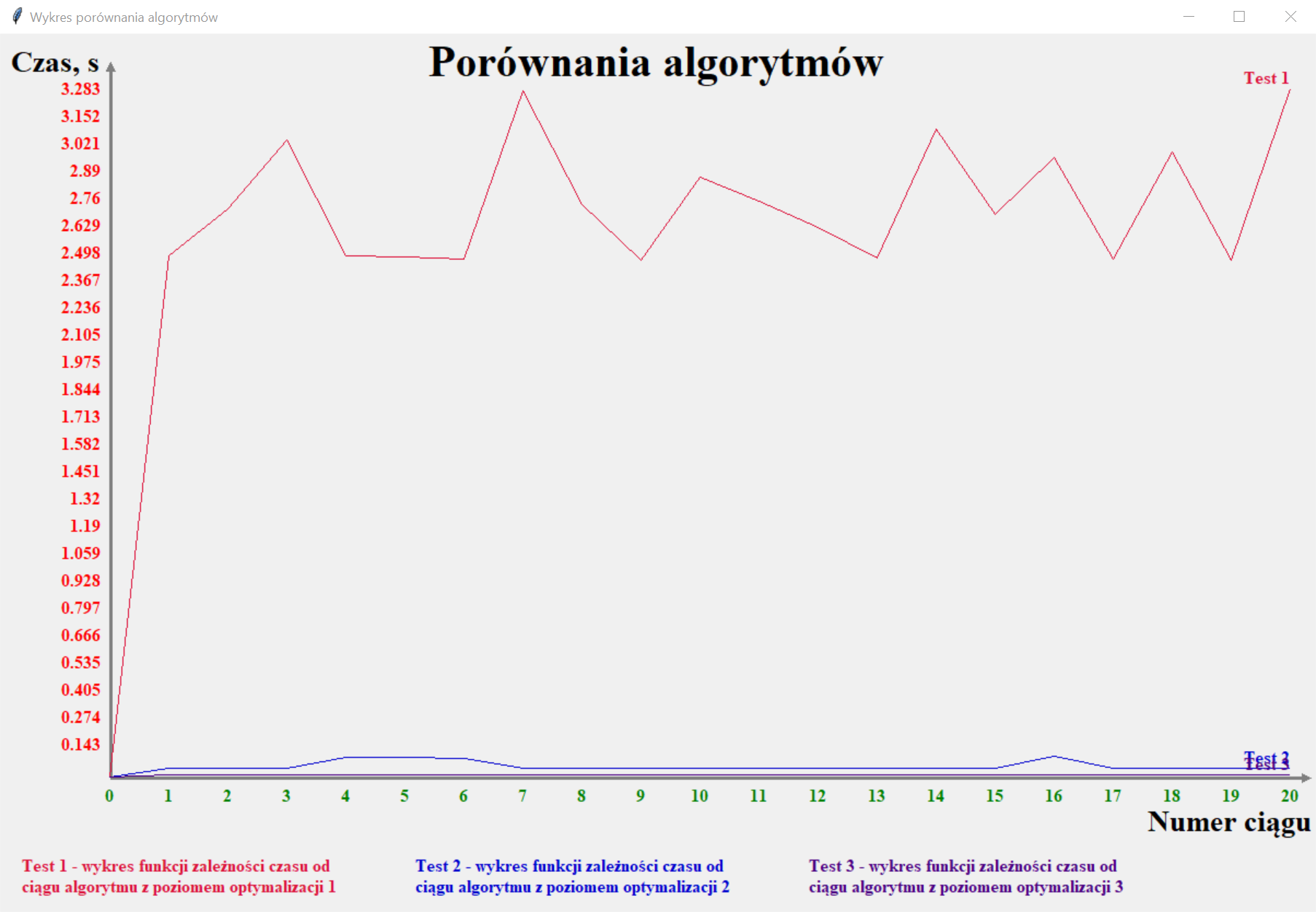
Ta funkcja daje nam wykres pokazany na Wykres 1. Widzimy, że chociaż wszystkie ciągi wejściowe mają identyczną długość, czas wykonania każdej z nich jest inny, ponieważ dane są losowe. Ponadto większość ciągów dążą do ideału, ponieważ dane nie są całkowicie losowe. Wynika to z faktu, że duża liczba liczb pseudolosowych dąży do idealnego podziału 1/1, co oznacza, że nasz algorytm będzie mógł bardzo łatwo znaleźć podciąg z taką samą liczbą jedynek i zer. Aby zobaczyć rzeczywisty czas wyszukiwania podciągów, sugeruję użycie stałego stosunku jedynek do zer, czyli 2/1. Zależność ta będzie równie trudna zarówno dla pierwszego algorytmu, jak i dla trzeciego, ponieważ jeśli będziemy dalej zwiększać stosunek (na przykład 3/1), to trzeci algorytm będzie mógł jeszcze łatwiej znaleźć podciąg w takich sekwencjach, a pierwszemu przeciwnie będzie bardziej trudno.

Aby otrzymać plik wejściowy o stałym stosunku zer do jedynek 2/1, należy skorzystać z funkcji worst\_sequence(), którą stworzyłem dla tych celów. Znajduje się ona w pliku pomocniczym assistant\_module.py. Utworzymy plik wejściowy zawierający 999 zer i jedynek z sekwencją „101101101 ...”. Nazwijmy go „report\_worst\_input.txt”, a plik dla wyniku działania naszego algorytmu - „report\_worst\_output.txt”.

Na wykresie (Wykres 3) otrzymujemy bardziej stabilne odczyty, gdzie wahania zależą głównie od tego, jak system operacyjny ładował procesora.



Wykres 2 Porównanie trzech algorytmów w trybie zwiększających się danych wejściowych.  
Źródło: opracowanie własne.



Wykres 3 Porównanie trzech algorytmów w trybie stałych danych wejściowych.  
Źródło: opracowanie własne.

Do tej pory pracowaliśmy ze stałą liczbą zer i jedynek w ciągu wejściowym. Spróbujemy stworzyć plik, w którym z każdym wierszem liczba jedynek i zer rośnie o 45, zaczynając od 100. Aby to zrobić, należy zmienić parametr „increment” na 15 („101”\*15 = 45 symbolów), a „start\_repeats” na 100 w poprzednio używanej funkcji worst\_sequence(). Otrzymujemy wykres gałęzi paraboli (Wykres 2).

# Dokumentacja z doświadczeń

## Wczytywania danych

Program akceptuje prawie każdy możliwy format danych wejściowych. Na przykład: „1, 0, 1, 0”; „1,0,1,0”; „1 0 1 0”; „1 \* 0 \* 1 \* 0”; „1, qwe0 e1 \* 0”; „1010”. Każde z tych wejść deklaruje następującą sekwencję: [1, 0, 1, 0] .W przypadku więcej niż jednego wejścia, należy je oddzielić nową linią. Na przykład to wejście:

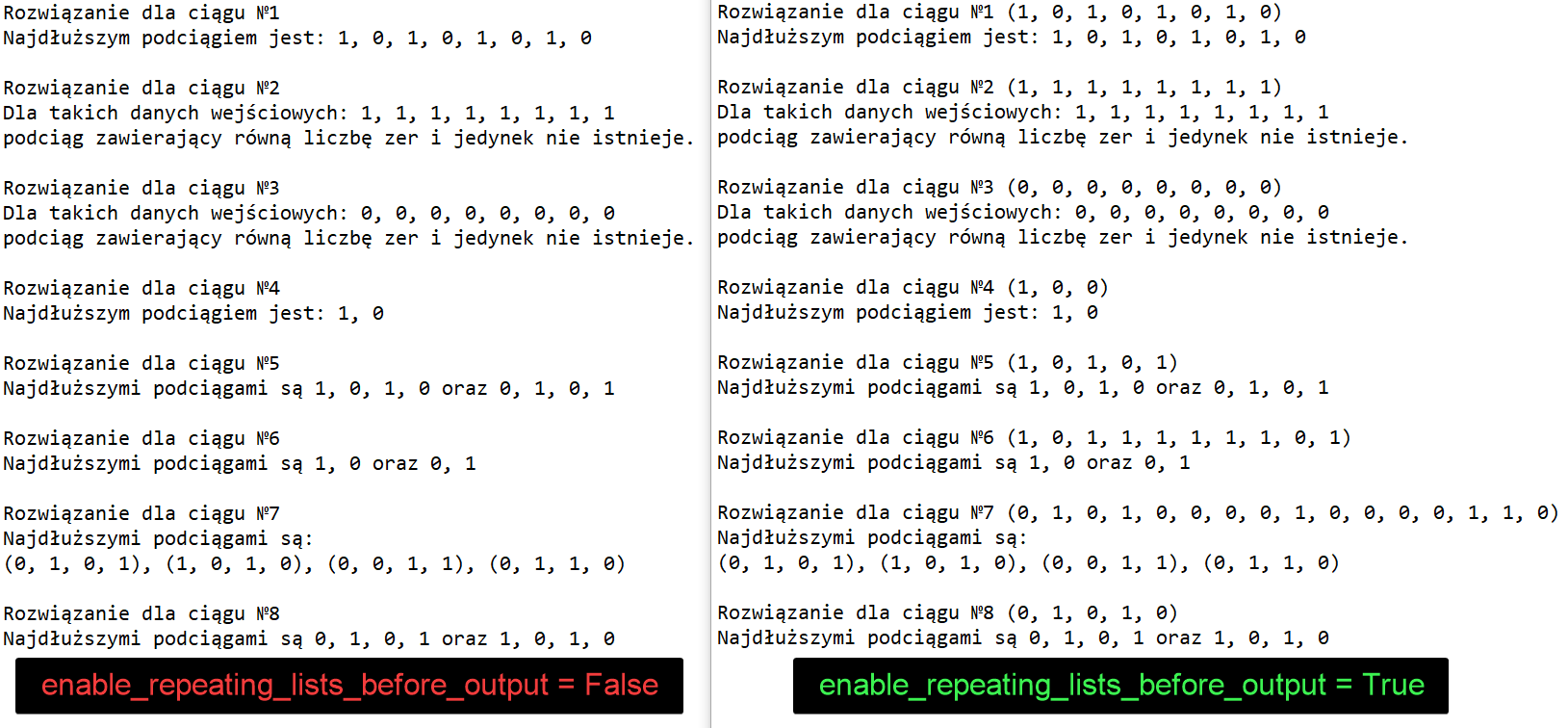
1111010111  
1101011010  
1011101010  
1111010101

będzie przyjęte jako 4 sekwencji:

[1, 1, 1, 1, 0, 1, 0, 1, 1, 1], [1, 1, 0, 1, 0, 1, 1, 0, 1, 0], [1, 0, 1 , 1, 1, 0, 1, 0, 1, 0], [1, 1, 1, 1, 0, 1, 0, 1, 0, 1]

## Wypisywania wyników

Istnieją dwie opcje zapisu podciągów wyjściowych w pliku: z powtarzaniem ciągów wejściowych i bez. Flaga „enable\_repeating\_lists\_before\_output” jest odpowiedzialna za wybranie odpowiedniej opcji. Przykład podano na rysunku poniżej (Rysunek 3).



Rysunek 3 Opcje zapisu pliku wyjściowego.  
Źródło: opracowanie własne.

# Wnioski

Opracowałem program, który znajduje wszystkie najdłuższe podciągi ciągu wejściowego, które mają tą samą liczbę zer i jedynek. Aby każdy sposób tworzenia pliku działał z moim programem, stworzono zaawansowaną funkcję czytnika plików, która akceptuje prawie każdy plik. Aby nie tworzyć danych za każdym razem ręcznie, napisano kilka funkcji pomocniczych, które mogą generować dane wejściowe w zależności od wprowadzonych ustawień. Algorytm został trzy razy zoptymalizowany i wszystkie trzy wersje zostały porównane ze sobą. Aby wyraźniej zademonstrować optymalizację algorytmów, stworzono kolejny program rysujący wykresy zależności czasu wykonania algorytmu od każdego podciągu. Wszystkie optymalizacje i ulepszenia zostały jasno przedstawione za pomocą graficznych diagramów głównego algorytmu. Schematy blokowe i pseudokody zostały stworzone dla funkcji odczytywania plików, odrzucania najprostszych przypadków i trzech głównych algorytmów.

## Legenda pseudokodów i schematów blokowych

## Spisy odwołań