WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

Wanesa Bułdyś

Wyszukiwanie największej możliwej liczby poprzez konstrukcję zadanych elementów

Projekt inżynierski

Opiekun pracy:

dr hab. inż. Mariusz Borkowski, prof. PRz

Rzeszów, 2025

Spis treści

1 Wstęp

Problem projektu polega na znalezieniu możliwie największej liczby, poprzez łączenie zadanych elementów (liczb) w sposób, który umożliwi utworzenie jak największej liczby. Elementy nie muszą być posortowane, jednak ich kolejność może wpłynąć na wynik działania programu, z tego względu ważna jest odpowiednia manipulacja kolejnością liczb, aby uzyskać największy możliwy wynik.

2 Szczegóły projektu

2.1 Opis problemu

W programie dany jest zbiór liczb w postaci tablicy, którego celem jest utworzenie największej możliwej liczby poprzez łączenie danych wejściowych, tak aby ich wynik był jak największy.

Dane wejściowe (liczby) mają być traktowane jako ciągi znaków, a nie liczby całkowite, ponieważ w kontekście tego zagadnienia mogłoby to doprowadzić do niewłaściwego wyniku końcowego.

2.2 Podstawy teoretyczne projektu

W celu rozwiązania powyższego problemu, należy porównać różne kombinacje liczb w sposób umożliwiający sprawdzenie, która z nich zwraca najwyższy wynik.

Głównym krokiem jest porównanie liczb w postaci ciągów znaków, aby ustalić, która z kombinacji daje wyższy wynik. Przykładowo dwie liczby a i b, porównujemy w dwóch możliwych układach:

- a+b
- b+a

gdzie znak "+", oznacza konkatenację łańcuchów.

Wybierany jest ten układ, który zwraca większy ciąg.

2.3 Szczegóły implementacji

1. Porównanie liczb.

   Zaimplementowana została funkcja porównująca liczby w postaci ciągów znaków. Jest to kluczowe do otrzymania prawidłowego rozwiązania powyższego programu.

2. Funkcja główna.

   Została zaimplementowana w celu wywołania odpowiednich operacji na danych wejściowych, sortowaniu ich oraz tworzeniu wyniku.

3. Testowanie programu.

   Program zawiera fukcję testującą poprawność działania algorytmu na różnych zestawach danych.

4. Odczyt i zapis do plików.

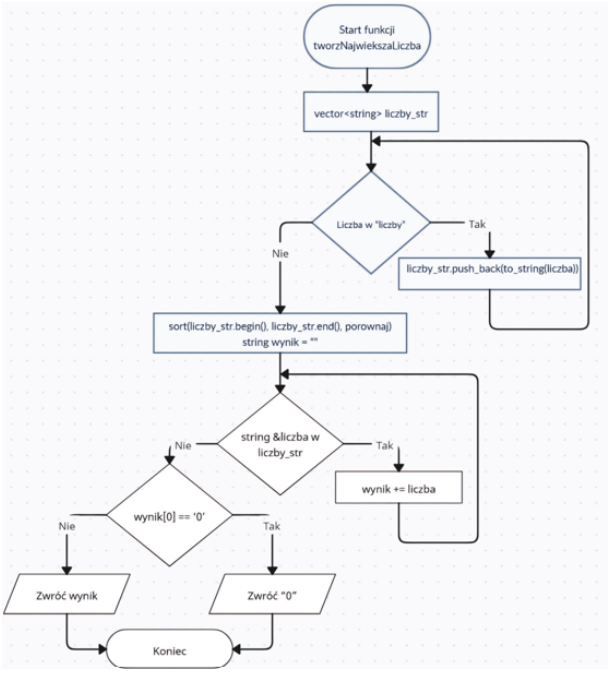
   Program umożliwia zarówno odczyt danych z plików, jak i zapis wyników do plików testowych.

5. Analiza wyników.

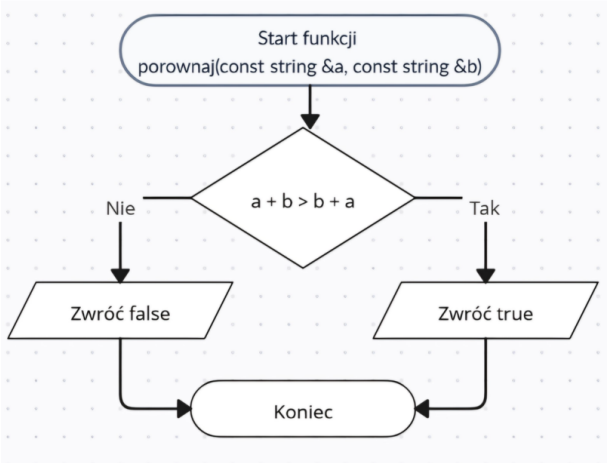
   Program zmierzy czas obliczeń w zależności od rozmiaru danych wejściowych.

3 Przykładowy zapis algorytmu

3.1 Schemat blokowy



Rysunek 1: Schemat blokowy funkcji tworzNajwiekszaLiczbe()



Rysunek 2: Schemat blokowy funkcji porownaj()

3.2 Pseudokod

```
1  Funkcja porownaj(a : string, b : string) → bool:
2      Return a + b > b + a
3  Funkcja tworzNajwiekszaLiczba(liczby: lista[int]) → string:
4      liczby_str ← nowa lista[string];
5      for każda liczba w liczby do
6          dodaj to_string(liczba) do liczby_str;
7      end
8      sortuj liczby_str według porownaj;
9      wynik ← "";
10     for każdy element w liczby_str do
11         wynik ← wynik + element;
12     end
13     if wynik[0] ==′ 0′ then
14         Return "0";
15     end
16     Return wynik;
```

4 Zoptymalizowany kod programu

4.1 Wprowadzone zmiany

1. Zarządzanie operacjami na plikach.

   Wszyskie dane wynikowe są przechowywane w pamięci oraz zapisywane do pliku jednorazowo na końcu programu, dzięki czemu minimalizujemy liczbę operacji otwierania i zamykania pliku.

2. Zarządzanie ciągami w funkcji.

   Konwertowanie liczb na ciągi odbywa się w miejscu, podczas ich sortowania. Dzięki temu zabiegowi unikamy tworzenia dodatkowego wektora, co redukuje zużycie pamięci.

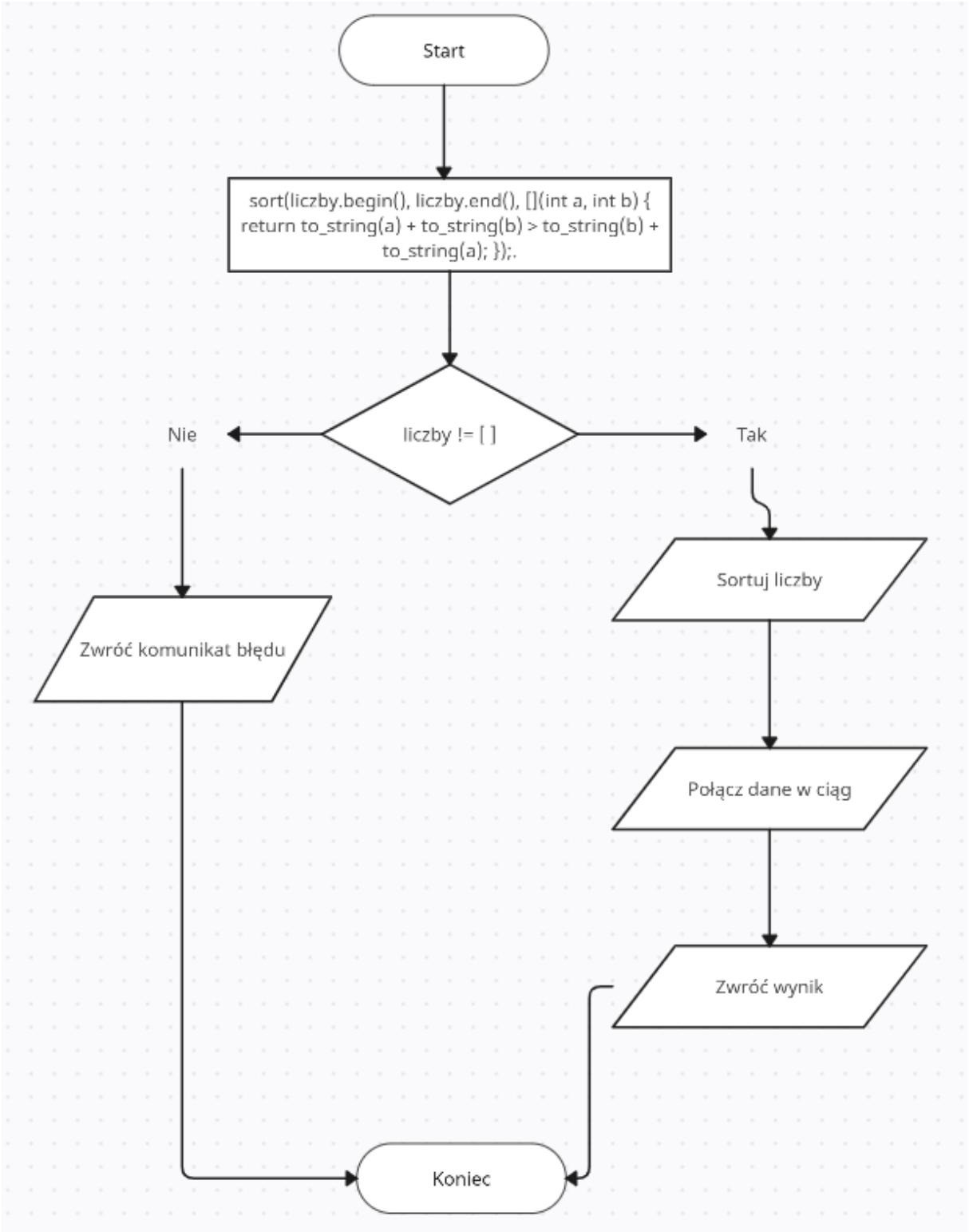
3. Lepsza obsługa błędów.

   Dodane zostały szczegółowe komunikaty o błędach, w przypadku braku danych wejściowych program kończy działanie i wyświetla odpowiedni komunikat.

4.2 Pseudokod

```
1  Funkcja sortujLiczby(liczby):
2      Posortuj liczby według następującego kryterium:;
3      for dwóch liczb a i b do
4          if to_string(a) + to_string(b) > to_string(b) + to_string(a) then
5              a jest przed b;
6          end
7          else
8              b jest przed a;
9          end
10     end
11 Funkcja polaczLiczby(liczby):
12     wynik ← Pusty ciąg znaków;
13     for każda liczba w liczby do
14         Dodaj to_string(liczba) do wynik;
15     end
16     Return wynik;
```

4.3 Schemat blokowy



Rysunek 3: Schemat blokowy zoptymalizowanego programu

5 Testy i analiza wyników

W obu programach zostały przeprowadzone testy na różnych zestawach danych, ich rezultaty zostały zapisane w poniższych tabelach.

Lp.	Dane wejściowe	Czas obliczeń [s]	Wynik
1.	10 81 9	0.005984	98110
2.	3 131 96 2	0.05795	9632131
3.	51 00 7 41 6	0.007051	7651410
4.	321 74 2	0.00547	743212
5.	723 46 12 2 2 9	0.006688	9723462212

Tabela 1: Testy pierwotnego kodu na różnych zestawach danych

Lp.	Dane wejściowe	Czas obliczeń [s]	Wynik
1.	10 81 9	0.000004156	98110
2.	3 131 96 2	0.000004832	9632131
3.	51 00 7 41 6	0.000005075	7651410
4.	321 74 2	0.000009270	743212
5.	723 46 12 2 2 9	0.000009858	9723462212

Tabela 2: Testy zoptymalizowanego kodu na różnych zestawach danych

W obu tabelach są identyczne wyniki dla takich samych danych wejściowych, co oznacza, że oba programy działają poprawnie. W przypadku pierwotnego kodu czas obliczeniowy jest zauważalnie wyższy, optymalizacja kodu wykazała znaczną poprawę wydajności, co umożliwiło zredukowanie czasu obliczeń o rzędy wielkości.

6 Złożoność czasowa i obliczeniowa

W projekcie przeanalizowano dwa podejścia do rozwiązania problemu tworzenia największej liczby z podanych elementów. Dwa różne kody zostały porównane pod względem złożoności czasowej i obliczeniowej, oraz ich wydajności w zależności od liczby elementów w danych wejściowych. Pierwotna wersja programu bazuje na standardowej metodzie przekształcania liczb na ciągi znaków, sortowania ich i łączenia w jedną dużą liczbę.

Główne operacje to:

1. Przekształcenie liczb na ciągi znaków – Złożoność: $O(n)$, gdzie n to liczba elementów.

2. Sortowanie ciągów znaków – Złożoność: $O(n \log n)$, ponieważ używamy standardowego sortowania.
3. Łączenie ciągów w jeden wynikowy ciąg – Złożoność: $O(n)$

Całkowita złożoność czasowa w przypadku pierwotnego kodu wynosi $O(nlogn)$, co jest głównie wynikiem operacji sortowania. Wykorzystanie pamięci zależy od liczby elementów i wymaga przechowywania liczb oraz ich reprezentacji w formie ciągów znaków.

W wersji zoptymalizowanej, zastosowano dodatkowe przechowywanie liczb w formie par (ciąg znaków, liczba), co pozwala na bardziej efektywne porównania w trakcie sortowania.

W tej wersji główne operacje to:

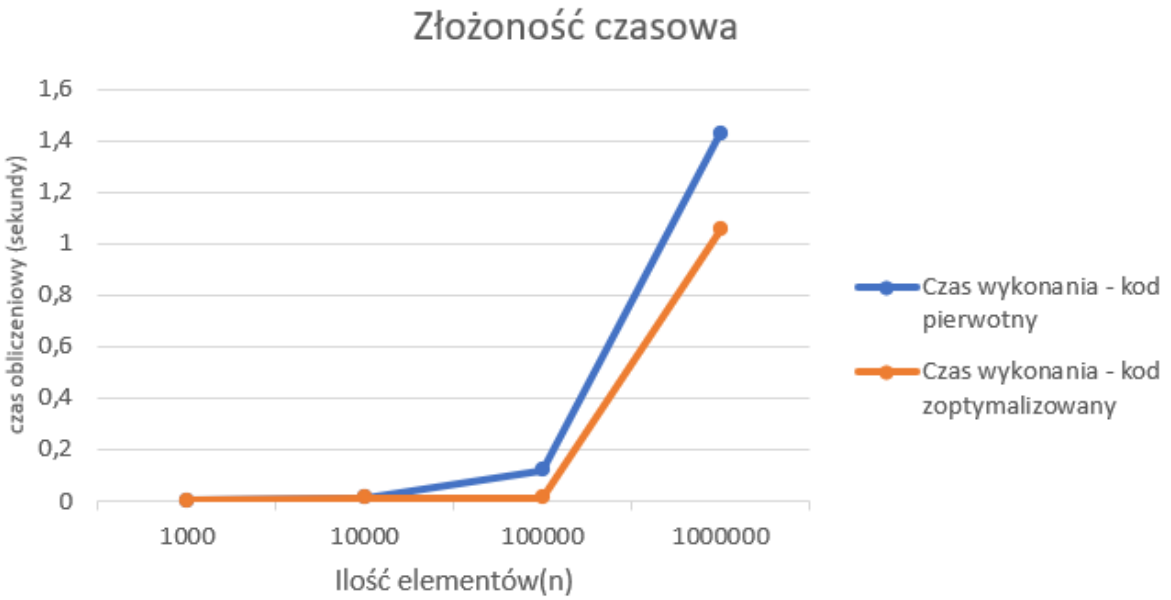
1. Przekształcenie liczb na ciągi znaków i przechowywanie ich w parze z oryginalnymi liczbami – Złożoność: $O(n)$.
2. Sortowanie par – Złożoność: $O(n \log n)$.
3. Łączenie ciągów znaków – Złożoność: $O(n)$.

Optymalizacja drugiego kodu przyczynia się do lepszego porównania ciągów znaków, ale jej wpływ na czas wykonania może być widoczny przy bardzo dużych danych ze względu na dodatkowe operacje wykonywane w 1000 powtórzeniach. Z tego względu pierwsza wersja może okazać się bardziej efektywna przy mniejszych zbiorach danych.

Aby porównać wydajność obu wersji kodów, przygotowano wykresy oraz tabele ilustrujące czas wykonania w zależności od liczby elementów w danych wejściowych. Pokazują, jak zmienia się czas wykonania w zależności od rozmiaru danych oraz jak oba kody różnią się pod względem efektywności. Analiza wyników umożliwia ocenę wpływu optymalizacji na czas wykonania oraz na porównanie wydajności obu podejść przy różnych rozmiarach danych.

Lp.	Liczba elementów	Czas wykonania (kod pierwotny)(sekundy)	Czas wykonania (kod zoptymalizowany)(sekundy)
1.	1000	0.0000997	0.000989
2.	10000	0.001097	0.12965
3.	100000	0.0118685	0.132648
4.	1000000	1.043077	1.53991

Tabela 3: Złożoność czasowa



Rysunek 4: Wykres złożoności czasowej dla obu programów

7 Wnioski

Po przeprowadzeniu testów i analizy obu wersji programu (pierwotnej i zoptymalizowanej), można zauważyć kilka istotnych różnic w ich wydajności oraz jakości wykonania.
Pierwotna wersja programu opierała się na prostym podejściu z sortowaniem liczb, które zostały przekonwertowane na ciągi znaków i posortowane według specjalnej funkcji porównawczej. Chociaż takie podejście jest wystarczające do rozwiązywania problemu w małych zbiorach danych, jego wydajność może być nieoptymalna dla dużych danych wejściowych.

Zoptymalizowana wersja znacząco poprawia wydajność, szczególnie przy dużych zbiorach danych. Zastosowanie par (pair<string, int>) pozwala na lepszą organizację danych, a dodatkowe operacje na ciągach znaków są bardziej efektywne. Dzięki temu, czas wykonania programu w przypadku dużej liczby powtórzeń (np. 1000 razy) może być zauważalnie krótszy, co jest szczególnie widoczne w testach. W obu wersjach programu zastosowano mechanizm pomiaru czasu, jednak w zoptymalizowanej wersji uwzględniono dodatkowe powtórzenia algorytmu (1000 razy), co pozwala uzyskać dokładniejsze wyniki pomiarów średniego czasu wykonania. Umożliwia to dokładniejsze porównanie czasów wykonania algorytmu oraz lepsze zrozumienie wpływu liczby powtórzeń na wydajność programu. Zoptymalizowana wersja programu nie tylko poprawia wydajność, ale także zapewnia lepszą organizację kodu, dzięki czemu jest bardziej czytelna i łatwiejsza do utrzymania. Dodatkowo, użycie funkcji do formatowania czasu oraz zapisywania wyników sprawia, że użytkownik ma dostęp do bardziej szczegółowych informacji o czasie wykonania i wynikach testów.

8 Podsumowanie

Projekt miał na celu stworzenie programu w języku C++, który pozwala na tworzenie jak największej możliwej liczby poprzez łączenie zadanych liczb w odpowiedniej kolejności. W ramach projektu przeanalizowano dwie wersje programu: pierwotną oraz zoptymalizowaną, dokonując porównań pod kątem wydajności, złożoności obliczeniowej, a także jakości wyników.

Testy wykazały, że zoptymalizowana wersja programu ma znaczną przewagę w przypadku większych zbiorów danych. Dzięki przeprowadzonym pomiarom czasu oraz zastosowaniu odpowiednich optymalizacji, program jest bardziej efektywny, zarówno pod względem zużycia czasu, jak i pamięci.

Ostatecznie, wnioski z projektu mogą być pomocne w przypadku dalszej optymalizacji algorytmów działających na dużych zbiorach danych, gdzie wydajność jest kluczowym czynnikiem.

Spis rysunków

Spis tabel