Zadanie projektowe - sprawozdanie

Kacper Zuba, nr indeksu 184322
Politechnika Rzeszowska im. Ignacego Łukasiewicza

Listopad 2025

Spis treści

1 Problematyka

1.1 Treść zadania

Dla zadanej tablicy liczb całkowitych znajdź maksymalny iloczyn dwóch elementów znajdujących się w tablicy.

Wejście: nieposortowana tablica liczb całkowitych
Wyjście: wszystkie pary liczb, które po pomnożeniu dadzą maksymalny iloczyn (spośród wszystkich zawartych w tablicy elementów).

1.2 Analiza problemu

Otrzymujemy potencjalnie nieposortowaną tablicę liczb całkowitych (o długości minimum 2) Spośród nich musimy znaleźć wszystkie dwuelementowe wariacje liczb, których iloczyn będzie największy spośród wszystkich takich wariacji.

Po wstępnej analizie problemu widzimy, że skuteczne może okazać się podejście kombinatoryczne w którym faktycznie sprawdzimy wszystkie możliwe wariacje liczb. Łatwo policzyć, że wtedy dla tablicy wejściowej o długości n będziemy musieli wykonać $\frac{n!}{(n-2)!}$ iteracji, wynik to $n^2 - n$ iteracji. Ewentualnie jeżeli zoptymalizujemy algorytm by nie sprawdzał dwa razy tych samych kombinacji (np. [1,2] oraz [2,1] dadzą ten sam wynik, bo mnożenie jest przemienne) i przejdziemy tym samym na kombinacje to wykonamy $\binom{n}{2}$ iteracji i zredukujemy ich liczbę do $\frac{n^2-n}{2}$, dalej jednak otrzymując algorytm o złożoności $O(n^2)$.

1.3 Wybór podejścia

Zacznę od implementacji algorytmu opisanego w poprzednim podrozdziale, coraz to bardziej modyfikując go i wyciągając wnioski z jego niedoskonałości by dojść do najbardziej optymalnej implementacji.

2 Propozycje implementacji algorytmu

Na początku przygotowano dwie uniwersalne funkcje do pracy na tablicach, które na pewno przydadzą się przy opracowywaniu coraz to kolejnych implementacji algorytmu. Użycie przy ich pisaniu wbudowanego w c++ mechanizmu szablonów funkcji sprawia, że będą one reużywalne nawet przy zmianie typów danych na jakich program będzie operował.

Funckja **wypelnijTabliceLiczbamiPseudolosowymi** przyjmuje jako argumeny tablicę, jej długość oraz dolny oraz dolny zakres z którego ma losować liczby. Korzysta z wbudowanej w C++ biblioteki do cstdlib, służącej m.in. do generowania liczb pseudolosowych.

Funckja **wypiszTablice** jak sama nazwa wskazuje wypisuje do konsoli przekazaną w argumencie tablicę w czytelnej formie do konsoli.

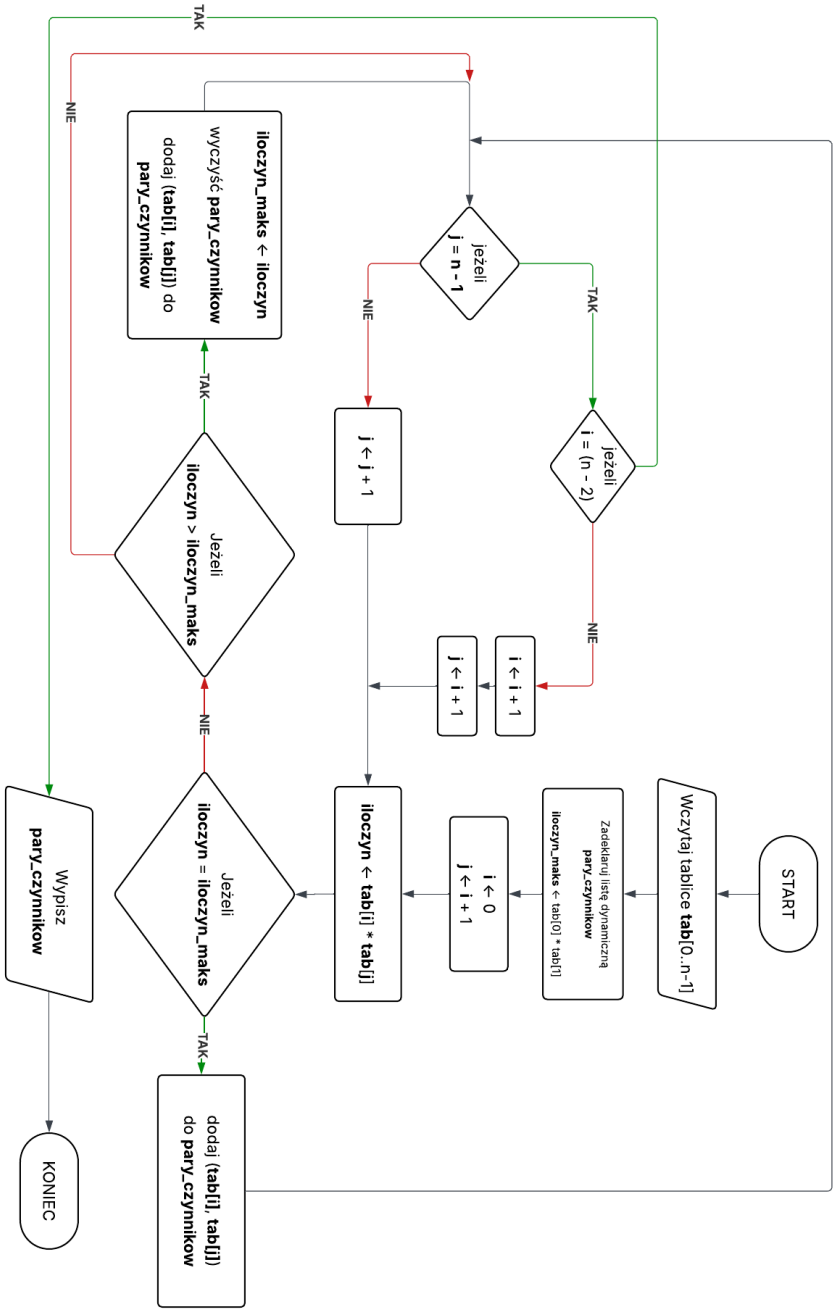
Z racji na użycie wskaźników przy obu tych funkcjach, będą one użyteczne dla wszystkich typów danych używających indekserów, a te - z racji uzycia template'ów - mogą też one przechowywać dowolne typy danych.

2.1 Algorytm brute-force

Zajmijmy się implementacją algorytmu brute-force na ten moment godząc się z jego potencjalną nieoptymalnością. Poniżej schemat blokowy tego rozwiązania.

```
#include <cstdlib>
#include <ctime>
template <typename T>
void wypelnijTabliceLosowymiLiczbamiZPrzedzialu(
    T* tab, int dlugosc_tablicy,
    T dolna_granica, T gorna_granica
    ) {
    srand(time(NULL));
    for(int i = 0; i < dlugosc_tablicy; i++) {
        tab[i] = dolna_granica + (
        rand() % (gorna_granica - dolna_granica + 1)
        );
    }
}


template <typename T>
void wypiszTablice(T* tablica, int dlugosc_tablicy) {
    cout<<'[';
    for(int i = 0; i < dlugosc_tablicy; i++) {
        cout<<tablica[i];
        if(i < (dlugosc_tablicy - 1)) {
            cout<<',';
        }
    }
    cout<<']';
}
```

Kod 1: funkcje do pracy na tablicach



Rysunek 1: Schemat blokowy algorytmu brute-force

2.1.1 Implementacja w C++

Na podstawie przygotowanego wcześniej schematu blokowego (rys. 2.1) przygotowano kod programu.

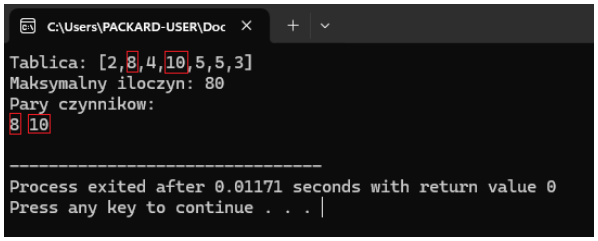
```cpp
#include <iostream>
#include <utility>
#include <vector>

using namespace std;

int main() {
    int n = 10;

    if (n < 2) {
        cerr << "Tablica musi miec co najmniej 2 elementy\n";
        return 1;
    }

    vector<int> tab(n);
    wypelnijTabliceLosowymiLiczbamiZPrzedzialu(
        tab.data(), n,
        0, 20
    );

    cout<<"Tablica: ";
    wypiszTablice(tab.data(), n);
    cout<<endl;

    int iloczyn_maks = tab[0] * tab[1];

    vector<pair<int, int>> pary_czynnikow;

    for (int i = 0; i < dlugosc_tablicy - 1; ++i) {
        for (int j = i + 1; j < dlugosc_tablicy; ++j) {
            int iloczyn = tab[i] * tab[j];

            if (iloczyn > iloczyn_maks) {
                iloczyn_maks = iloczyn;
                pary_czynnikow.clear();
                pary_czynnikow.push_back({tab[i], tab[j]});
            } else if (iloczyn == iloczyn_maks) {
                pary_czynnikow.push_back({tab[i], tab[j]});
            }
        }

    }

    cout << "Maksymalny iloczyn: " << iloczyn_maks << "\n";
    cout << "Pary czynnikow:\n";
    for (const auto &p : pary_czynnikow) {
        cout << p.first << " " << p.second << "\n";
    }


    return 0;
}
```

Kod 2: algorytm brute-force

W kodzie użyto dwóch wbudowanych w C++ typów, jakimi są wektor oraz para. Wektor to rodzaj tablicy dynamicznej, która z racji korzystania z możliwości odczytywania indekserów jest kompatybilna z napisanymi wcześniej funkcjami (kod 1). Para to po prostu struktura danych przetrzymująca dwie wartości danego typu, więc nadaje się idealnie do przechowywania par czynników.

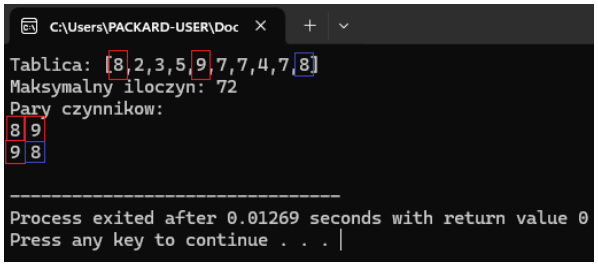
2.1.2 Wyniki

Pierwszych uruchomienia programu dają poprawne wyniki.



Rysunek 2: Wynik uruchomienia programu z algorytmem brute force

Zaobserwowano także pewne niedociągnięcie. W przypadku kiedy w liście powtarza się jeden z czynników, to wpis z tymi samymi czynnikami pojawia się wielokrotnie.



Rysunek 3: Problem z powtarzającymi się rekordami w algorytmie brute force

Napisałem w tym celu funckję, która zwraca wartość logiczna mówiącą, czy para zawiera dwa takie same elementy. Nie możena tutaj użyć wbudowanego porównania par, gdyż ono bierze pod uwagę kolejność ([9,4] to coś innego niż [4, 9]). Przedrostek const użyty przy argumentach funkcji oznacza

```
bool czyParyZawierajaTeSameElementy
    (const pair<int,int>& a, const pair<int,int>& b) {
    return (a.first == b.first && a.second == b.second) ||
           (a.first == b.second && a.second == b.first);
}
```

Kod 3: funkcja do porównywania par

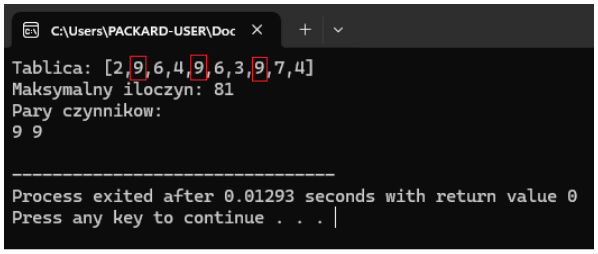
że przekazujemy niemodyfikowalną wartośc i jest wymagany do działania programu (ma to związek z tworzeniem par przez składnie C++ w wersji używanej przeze mnie).

Następnie zmodyfikowane pętle w algorytmie, by sprawdzała, czy para o tych elementach została już zawarta.

Po sprawdzeniu widać, że problem został rozwiązany i pomimo wielokrotnego wystąpienia czynnika, para jest wypisana jednokrotnie. Przeprowadziłem test również dla liczb ujemnych i bez zaskoczeń -

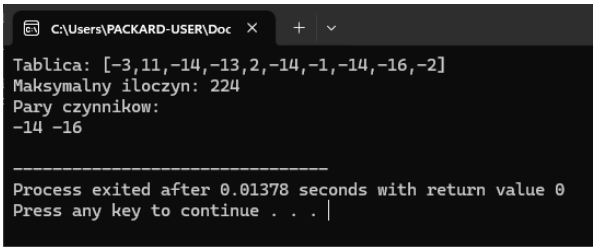
```
for (int i = 0; i < dlugosc_tablicy - 1; ++i) {
    for (int j = i + 1; j < dlugosc_tablicy; ++j) {
        int iloczyn = tab[i] * tab[j];

        if (iloczyn > iloczyn_maks) {
            iloczyn_maks = iloczyn;
            pary_czynnikow.clear();
            pary_czynnikow.push_back({tab[i], tab[j]});
        } else if (iloczyn == iloczyn_maks) {
            pair<int,int> nowa_para = {tab[i], tab[j]};

            bool czy_para_zostala_juz_zawarta = false;
            for (const auto& p : pary_czynnikow) {
                 if (czyParyZawierajaTeSameElementy
                      (p, nowa_para)) {
                    czy_para_zostala_juz_zawarta = true;
                    break;
                }
            }

            if (!czy_para_zostala_juz_zawarta) {
                pary_czynnikow.push_back(nowa_para);
            }
        }
    }
}
```

Kod 4: modyfikacja pętli w algorytmie, by wyeliminować powtórzenia



Rysunek 4: Wynik po poprawce

algorytm działa również dla tego zestawu danych.



Rysunek 5: Wynik działania programu z algorytmem brute force przy rozszerzeniu zestawu danych na liczby ujemne

2.1.3 Obserwacje

Taka implementacja daje poprawne rezultaty, ale dalej reprezentuje kwadratową złożoność obliczeniową. Zaobserwowano, że przy rozszerzeniu zestawu na liczby ujemne, że zawsze pary liczb stanowiące czynniki największego iloczynu to:

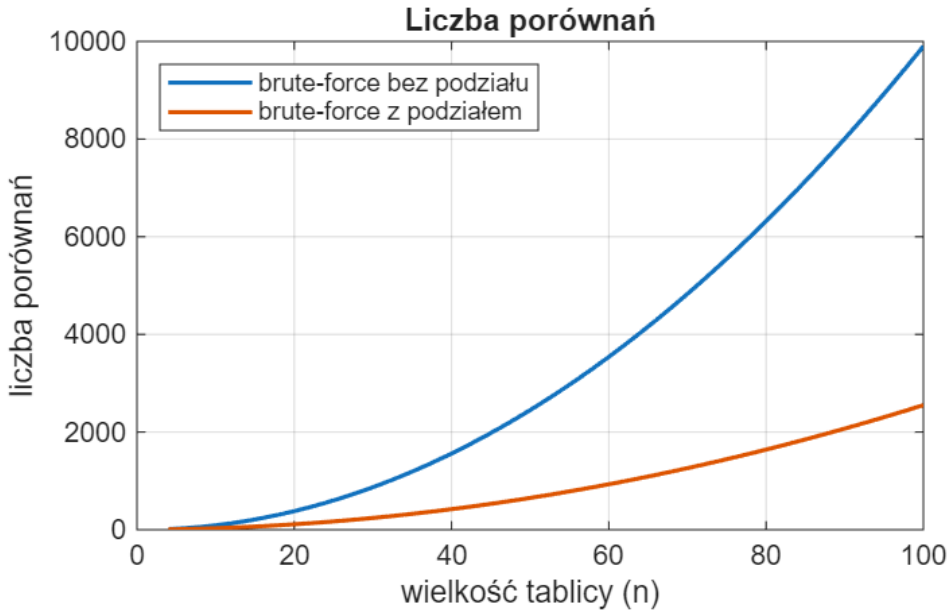
- dwie liczby dodatnie,
- dwie liczby ujemne

Nie powinno to dziwić, bo iloczyn liczby dodatniej z ujemną to liczba ujemna, a to na wyjściu już zmniejsza jej szansę bycia największym iloczynem. Należy jedna zastanowić się, czy w takim razie rozpoczęcie algorytmu od podziału tablic na dwie: jedna z ujemnymi, a druga z dodatnimi, a następnie zajmowanie się nimi osobno, mogłoby przynieść zmniejszenie liczby iteracji (dalej jednak nie obniżając złożoności $O(n^2)$) dla większych zestawów danych wejściowych.

Dla przykładu: 100-elementowy zestaw danych wejściowych a w tym 50 dodatnich i 50 ujemnych, Załóżmy, że dane zostaną posortowane do dwóch tablic, wykonując 100 porównań. Następnie każda z nich zostaje przeszukana metodą brute force. Na samym końcu wystarczy tylko zobaczyć, z która tablica dała większy iloczyn, wykonując jedno dodatkowe porównanie. Zgeneralizujmy liczbę porównań tego podejścia dla tablicy rozmiaru n przy optymistycznym założeniu, że dokładnie połowa liczb będzie ujemna.

$$n + 2 * \binom{n/2}{2} + 1$$

Złożoność - wciąż kwadratowa, ale patrząc na to ze perspektywy przyrostu wielkosci danych widzimy znaczną różnicę w liczbie iteracji.

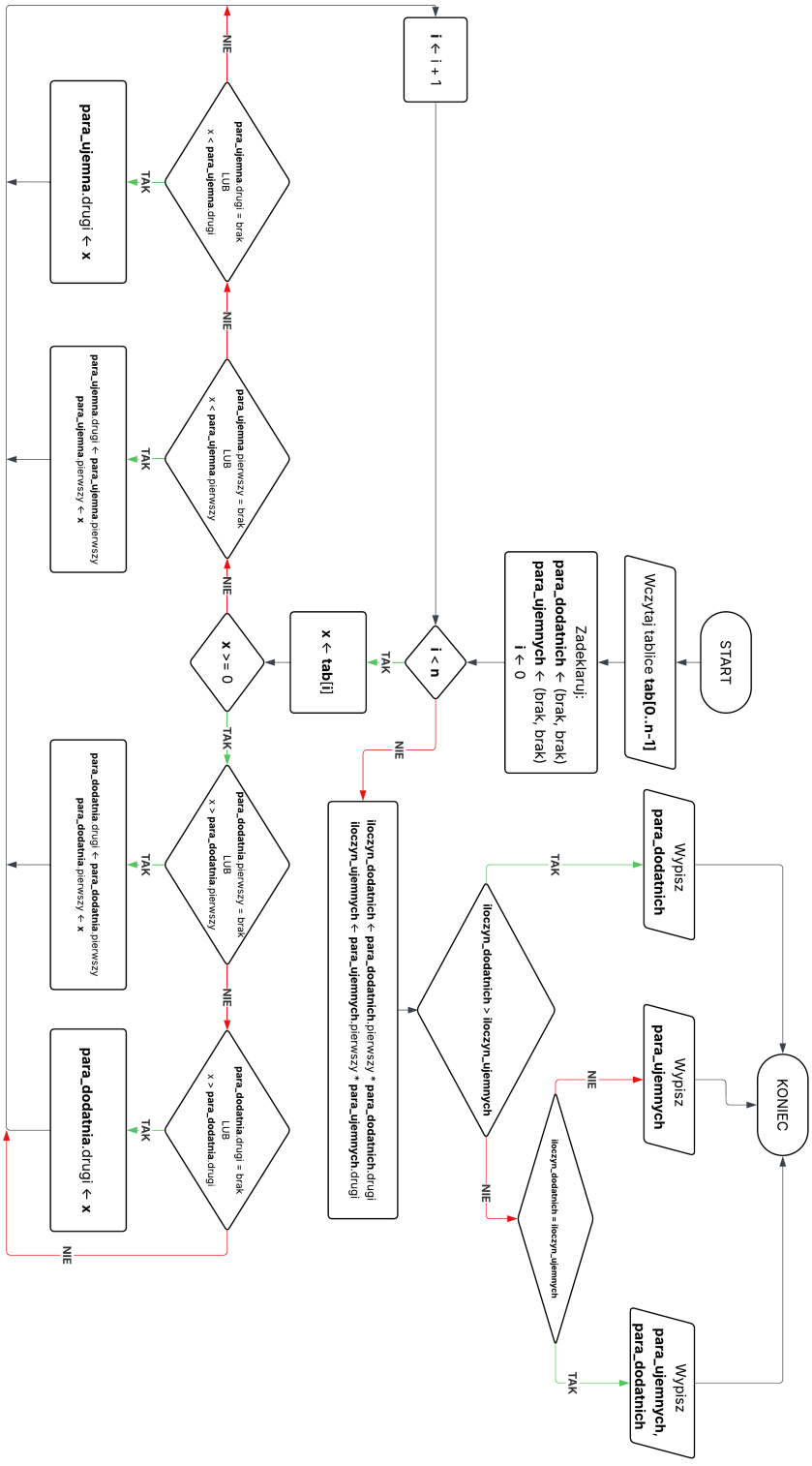


Rysunek 6: Liczba iteracji przy obu podejściach

Jest jednak jeszcze jedna, ciekawsza obserwacja, bo - jak się okazuje - nie trzeba porównywać ze sobą wszystkich par. Wystarczy znaleźć parę dwóch największych dodatnich oraz (ewentualnie) dwóch najmniejszych ujemnych liczb, bo jedynie iloczyn tych par może dać największy iloczyn.

2.2 Algorytm optymalny (liniowy)

Zgodnie z obserwacją opracowuję algorytm, który wyszukuje interesujące nas pary. Na końcu porównujemy iloczyny, które dają i wypisujemy odpowiedni wynik.



Rysunek 7: Schemat blokowy algorytmu po optymalizacji

Dzięki temu podejściu otrzymujemy algorytm o liniowej złożoności obliczeniowej, bo jedynie raz musimy przejść po wszystkich elementach tablicy i umieścić je w odpowiedniej grupie. Ta optymalizacja do złożoności $O(n)$ będzie oczywiście najbardziej zauważalna przy dużych zbiorach danych. Najpierw zajmijmy się implementacją programistyczną.

2.2.1 Implementacja w C++

```
if(tab.size() == 2) {
    cout<<"Maksymalny iloczyn: "<<tab[0] * tab[1]<<endl;
    cout<<"["<<tab[0]<<", "<<tab[1]<<"]";
    return;
}

pair<int,int> para_dodatnia = {INT_MIN, INT_MIN};
pair<int,int> para_ujemna = {INT_MAX, INT_MAX};

for (int x : tab) {
    if (x > 0) {
        if (x > para_dodatnia.first) {
            para_dodatnia.second = para_dodatnia.first;
            para_dodatnia.first = x;
        } else if (x > para_dodatnia.second) {
            para_dodatnia.second = x;
        }
    }

    if (x < 0) {
        if (x < para_ujemna.first) {
            para_ujemna.second = para_ujemna.first;
            para_ujemna.first = x;
        } else if (x < para_ujemna.second) {
            para_ujemna.second = x;
        }
    }
}

int iloczyn_dodatnie = (para_dodatnia.second != INT_MIN
                      ? para_dodatnia.first * para_dodatnia.second
                      : INT_MIN);

int iloczyn_ujemne = (para_ujemna.second != INT_MAX
                      ? para_ujemna.first * para_ujemna.second
                      : INT_MIN);

cout<<"Maksymalny iloczyn: "<<(iloczyn_dodatnie > iloczyn_ujemne
    ? iloczyn_dodatnie
    : iloczyn_ujemne)<<endl;

cout << "Pary czynnikow:\n";
if(iloczyn_dodatnie >= iloczyn_ujemne) {
    cout<<"["<<para_dodatnia.first<<", "<< para_dodatnia.second << "]\n";
}

if(iloczyn_ujemne >= iloczyn_dodatnie) {
    cout<<"["<<para_ujemna.first<<", "<< para_ujemna.second << "]\n";
}
```

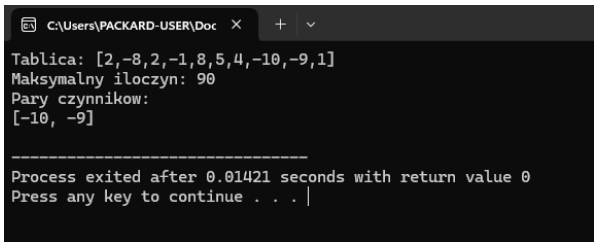
Kod 5: algorytm zoptymalizowany

Ta implementacja jest właściwie bezpośrednim przełożeniem schematu blokowego z jedną zmianą.

Instrukcja warunkowa na samym początku sprawdza, czy długość tablicy to dokładnie 2, bo to jedyna sytuacja, w której możemy nie znaleźć pary liczb dodatnich lub ujemnych (sytuacje z listą jednoelementową lub pustą zostały wykluczone wcześniej).

2.2.2 Wyniki

Ponownie, wyniki produkowane przez program są poprawne. Tutaj, w przeciwieństwie do poprzedniej implementacji, nie ma możliwości by pary liczb wypisały się wielokrotnie, więc nie trzeba niczego dodatkowo filtrować.



Rysunek 8: Wynik działania programu ze zoptymalizowanym algorytmem

3 Podsumowanie

Zadanie zostało wykonane, ale można wyciągnąć jeszcze z niego dodatkowe wnioski.

3.1 Benchmark algorytmów

Z pomocą modelu językowego stworzyłem funkcję pozwalającą zmierzyć czas wykonania funkcji.

```
#include <chrono>
template <typename F, typename... Args>
long long zmierz_czas_wykonania(F&& funkcja, Args&&... args){
    auto start = chrono::high_resolution_clock::now();

    std::forward<F>(funkcja)(std::forward<Args>(args)...);

    auto stop = chrono::high_resolution_clock::now();

    return chrono::duration_cast<chrono::milliseconds>(stop - start).count();
}
```

Kod 6: Kod funkcji do pomiaru czasu egzekucji funkcji

Dzięki niej byłem w stanie dokładnie zmierzyć czas wykonania obu algorytmów dla rosnących zestawów danych wejściowych. Wyniki nie są zaskakujące. Dla rosnącego zestawu danych odnotowujemy znacznie szybszy wzrost czasu wykonania dla niezoptymalizowanej wersji algorytmu.

Wielkość danych wejściowych	100	1000	10000
Czas wykonania algorytmu zoptymalizowanego [ms]	0	1	1
Czas wykonania algorytmu brute force [ms]	1	3	133

Tabela 1: Wyniki pomiarów czasu wykonania algorytmów

O ile do dwóch pierwszych odczytów można wysnuć wątpliwości, to przy trzecim dla zestawu danych składającego się z 10 tysięcy elementów nie ma mowy o błędzie pomiarowym. Czas wykonania wersji brute force jest kolosalny w porównaniu do wersji alternatywnej.

3.2 Wnioski

Pomimo, że zadanie z początku wydawało się banalne, na drodze do rozwiązania pojawiało się wiele pomniejszych problemów, które przy odpowiednim podejściu udawało się proceduralnie eliminować. Wnikliwa obserwacja oraz analiza matematyczna pozwoliła również znaleźć zależność, której wykorzystanie z kolei doprowadziło do zoptymalizowania algorytmu od $O(n^2)$ do $O(n)$.

Spis rysunków