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

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 $\frac{n!}{(n-2)!}$, czyli $n^2 - n$ porównań. Ewentualnie jeżeli zoptymalizujemy algorytm by nie sprawdzał dwukrotnie tych samych par (np. (1,2) oraz (2,1) dadzą ten sam wynik, bo mnożenie jest przemienne) i przejdziemy tym samym na rozważanie kombinacji, to wykonamy $\binom{n}{2}$, czyli $\frac{n^2-n}{2}$ porównań, dwukrotnie zmniejszając ich liczbę względem poprzedniej koncepcji, 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 przygotowałem 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.

```
template <typename T>
void wypelnijTabliceLosowymiLiczbamiZPrzedzialu(
    T* tab, int dlugosc_tablicy,
    T dolna_granica, T gorna_granica
    ) {
        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: kod funkcji do pracy na tablicach

Funkcja **wypelnijTabliceLiczbamiPseudolosowymi** przyjmuje jako argumenty tablicę, jej długość oraz zakres liczb spośród których mają zostać (pseudo)wylosowane elementy tablicy. Korzysta z wbudowanej w C++ biblioteki cstdlib, służącej m.in. do generowania liczb pseudolosowych. Należy również wspomnieć, że przed użyciem tej funkcji należy uprzednio zainicjalizować ziarno generatora liczb pseudolosowych, wywołując srand(time(NULL)). Nie mogę tego zrobić wewnątrz tej funkcji, bo mogłoby to skutkować powtarzaniem generowanych wartości w przypadku wielokrotnego wywołania tej funkcji w krótkim czasie.

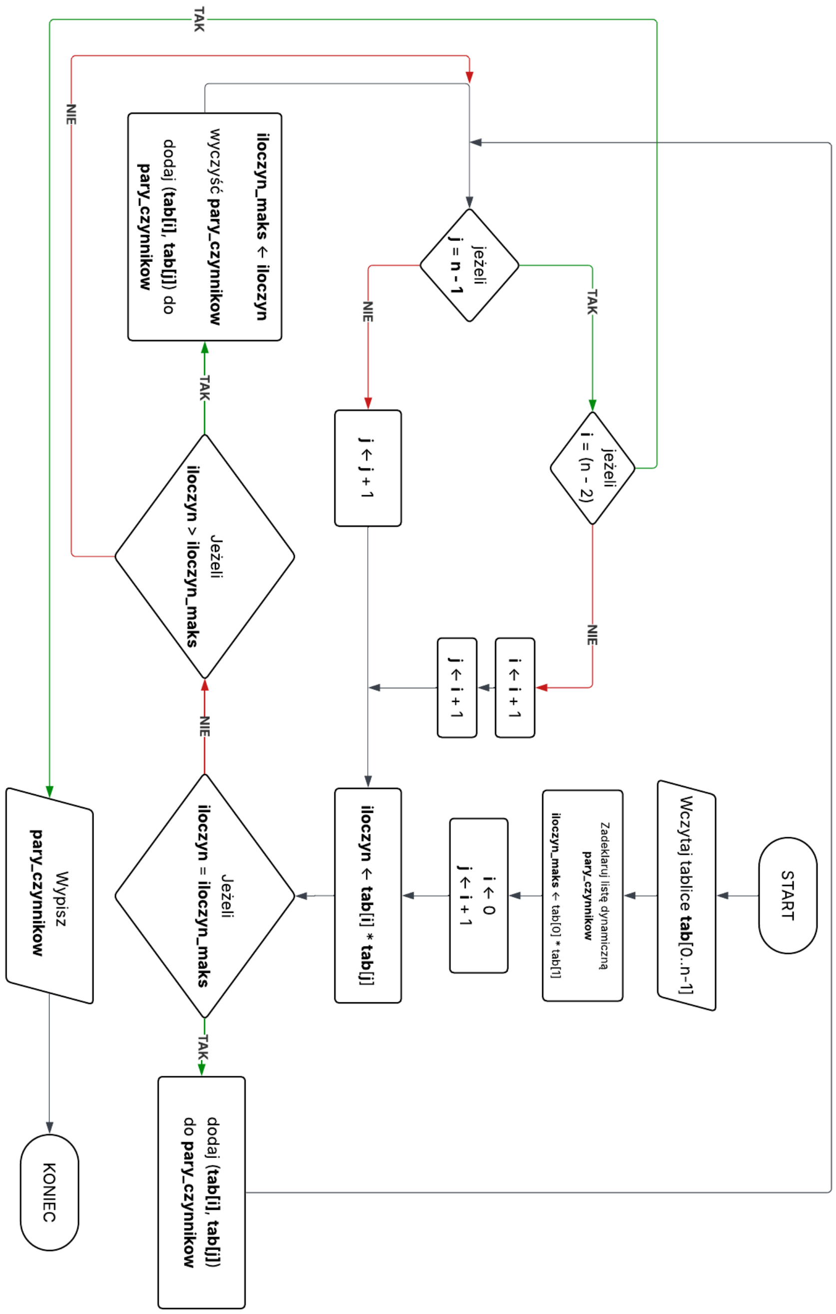
Funkcja **wypiszTablice** jak sama nazwa wskazuje wypisuje przekazaną w argumencie tablicę w czytelnej formie do konsoli.

Z racji użycia 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 (złożoność kwadratowa)

2.1.1 Schemat blokowy

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



Rysunek 1: Schemat blokowy algorytmu brute-force

2.1.2 Ręczne sprawdzenie poprawności działania algorytmu

Sprawdzam ręcznie działanie algorytmu dla przykladowych danych wejściowych: $[0, 4, -5, 8, -8]$ by wyłapać potencjalne błędy. Dla danej tablicy wejściowej poprawny algorytm powinien wypisać parę liczb (-5, -8).

i	j	tab[i]	tab[j]	iloczyn	iloczyn_maks	pary_czynnikow (po iteracji)
0	1	0	4	0	0	$\{(0, 4)\}$
0	2	0	-5	0	0	$\{(0, 4), (0, -5)\}$
0	3	0	8	0	0	$\{(0, 4), (0, -5), (0, 8)\}$
0	4	0	-8	0	0	$\{(0, 4), (0, -5), (0, 8), (0, -8)\}$
1	2	4	-5	-20	0	$\{(0, 4), (0, -5), (0, 8), (0, -8)\}$
1	3	4	8	32	32	$\{(4, 8)\}$
1	4	4	-8	-32	32	$\{(4, 8)\}$
2	3	-5	8	-40	32	$\{(4, 8)\}$
2	4	-5	-8	40	40	$\{(-5, -8)\}$
3	4	8	-8	-64	40	$\{(-5, -8)\}$

Tabela 1: Prześledzenie działania algorytmu brute force (ze schematu blokowego)

Wynik końcowy: `iloczyn_maks` $= 40$, `pary_czynnikow` $= \{(-5, -8)\}$. Algorytm działa poprawnie.

2.1.3 Implementacja w C++

Na podstawie przygotowanego wcześniej schematu blokowego (rys. 2.1.1) przygotowano kod programu.

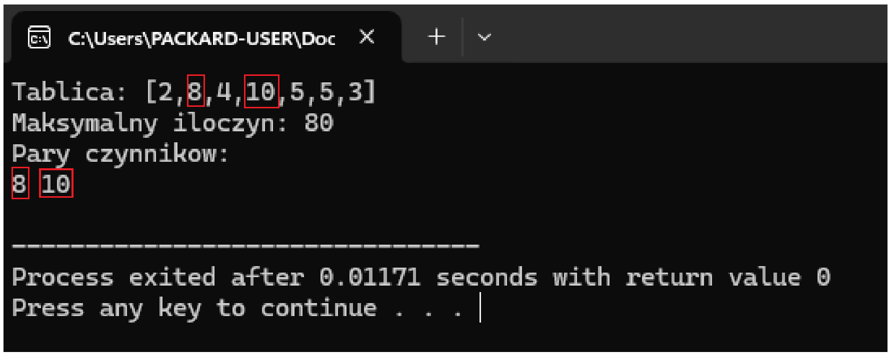
```cpp
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

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.

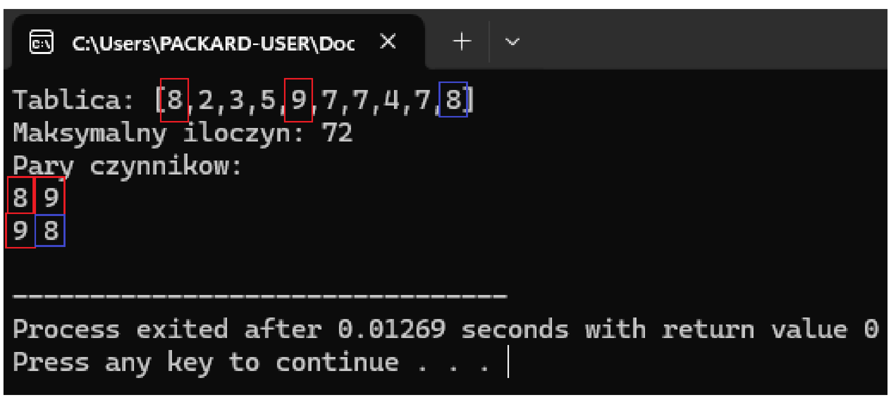
2.1.4 Wyniki

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, tzn. (9,4) to coś innego niż (4,9).

```
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

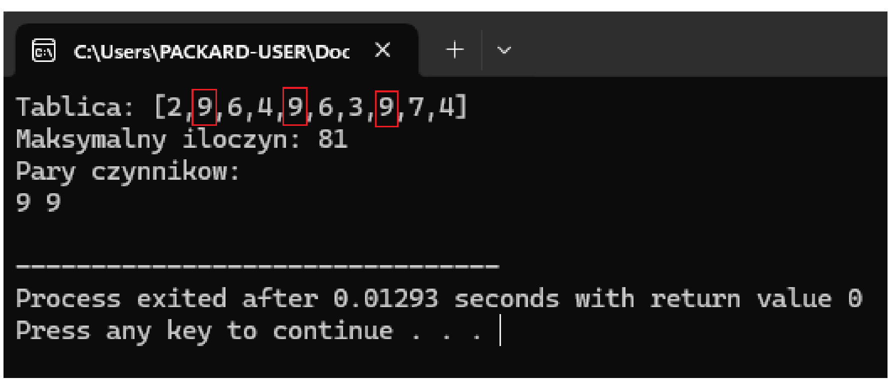
Przedrostek const użyty przy argumentach funkcji oznacza że przekazujemy niemodyfikowalną wartość i jest wymagany do działania programu (ma to związek z tworzeniem par przez składnie C++ w wersji używanej przeze mnie).

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

```
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);
            }
        }
    }
 }
```

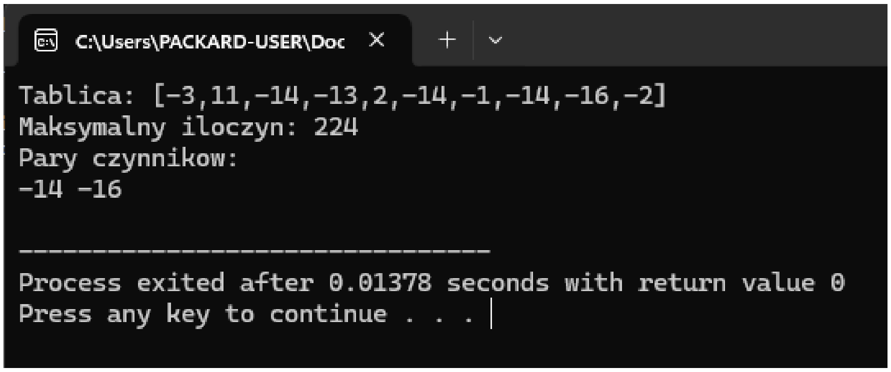
Kod 4: Modyfikacja pętli w algorytmie, by wyeliminować powtórzenia

Po sprawdzeniu widać, że problem został rozwiązany i pomimo wielokrotnego wystąpienia czynnika, para jest wypisana jednokrotnie.



Rysunek 4: Wynik po wprowadzeniu poprawki eliminującej powtórzenia

Przeprowadziłem test również dla liczb ujemnych i bez zaskoczeń - algorytm działa również dla tego zestawu danych.



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

2.1.5 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:

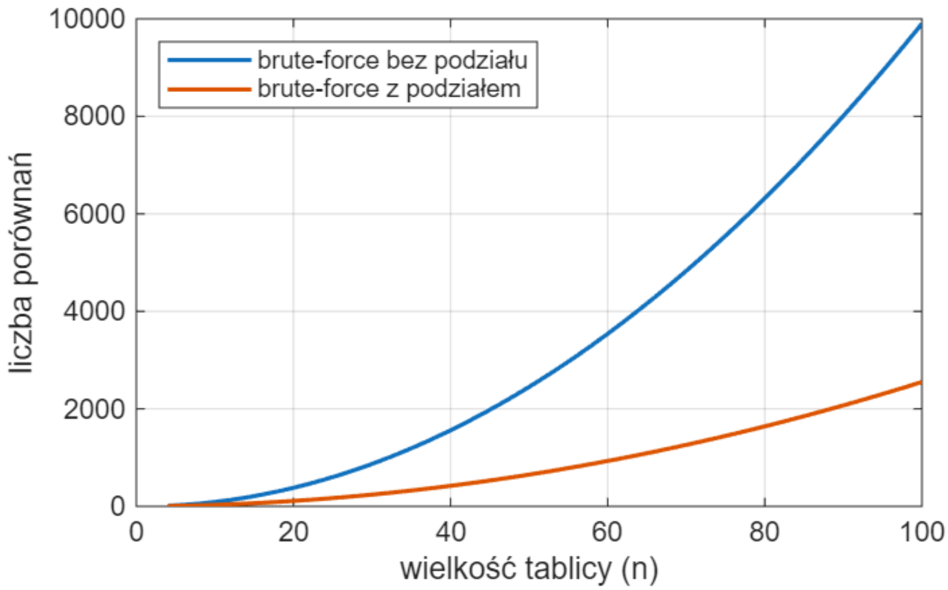
- dwie liczby dodatnie,
- dwie liczby ujemne

Następuje zależność, że jedyną sytuacją w której największy iloczyn w tablicy będzie ujemny, będzie przekazanie do wejścia dwuelementowej tablicy: pierwszy element dodatni, a drugi ujemny. Dodanie kolejnego elementu sprawi, że w tablicy znajdzie się przynajmniej jedna para elementów tego samego znaku, a ich iloczyn dla liczbę dodatnią. Z tej zależności wynika, że największy iloczyn w tablicy mogą dać jedynie dwie dodatnie lub dwie ujemne liczby prowadzi do dalszej optymalizacji. Należy 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 dokłądnie połowa liczb będzie ujemna.

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

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



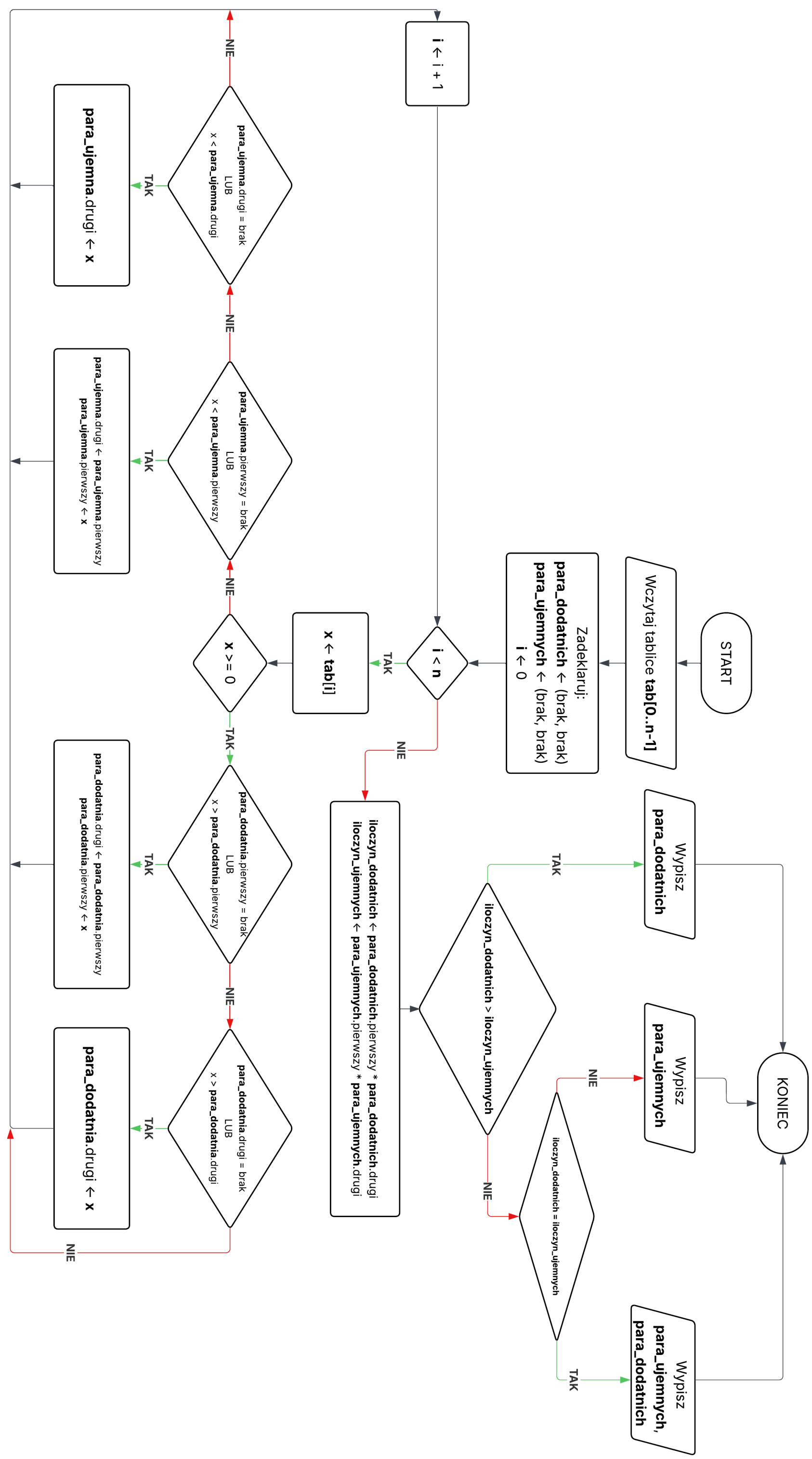
Rysunek 6: Wykres przedstawiajacy liczbę iteracji przy obu podejściach

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 (złożoność liniowa)

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

2.2.1 Schemat blokowy oraz pseudokod



Rysunek 7: Schemat blokowy algorytmu po optymalizacji

```
JESLI rozmiar(tab) = 2:
   iloczyn <- tab[0] * tab[1]
   WYPISZ iloczyn
   WYPISZ pare (tab[0], tab[1])
   STOP


para_dodatnia <- (BRAK, BRAK) // (najwieksza_dodatnia, druga_najwieksza_dodatnia)
para_ujemna <- (BRAK, BRAK) // (najbardziej_ujemna, druga_najbardziej_ujemna)


DLA kazdego x w tab:
   JESLI x > 0:

      JESLI para_dodatnia.element1 = BRAK LUB x > para_dodatnia.element1:
         para_dodatnia.element2 <- para_dodatnia.element1
         para_dodatnia.element1 <- x

      W PRZECIWNYM RAZIE JESLI para_dodatnia.element2 = BRAK LUB x > para_dodatnia.element2:
         para_dodatnia.element2 <- x


   JESLI x < 0:

      JESLI para_ujemna.element1 = BRAK LUB x < para_ujemna.element1:
         para_ujemna.element2 <- para_ujemna.element1
         para_ujemna.element1 <- x

      W PRZECIWNYM RAZIE JESLI para_ujemna.element2 = BRAK LUB x < para_ujemna.element2:
         para_ujemna.element2 <- x



najwiekszy_iloczyn <- BRAK
najlepsza_para <- (BRAK, BRAK)

JESLI para_dodatnia.element1 != BRAK ORAZ para_dodatnia.element2 != BRAK:
   iloczyn <- para_dodatnia.element1 * para_dodatnia.element2
   najwiekszy_iloczyn <- iloczyn
   najlepsza_para <- (para_dodatnia.element1, para_dodatnia.element2)


JESLI para_ujemna.element1 != BRAK ORAZ para_ujemna.element2 != BRAK:
   iloczyn <- para_ujemna.element1 * para_ujemna.element2

   JESLI najwiekszy_iloczyn = BRAK LUB iloczyn > najwiekszy_iloczyn:
      najwiekszy_iloczyn <- iloczyn
      najlepsza_para <- (para_ujemna.element1, para_ujemna.element2)


WYPISZ najwiekszy_iloczyn
WYPISZ pare najlepsza_para
```

Kod 5: Pseudokod algorytmu o złożoności liniowej

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.

2.2.2 Implementacja w C++

Na kanwie tego pseudokodu tworzę implementację komputerową. Umyślnie umieszczam tylko faktyczną logikę algorytmu (bez importów bibliotek oraz sygnatury funkcji głównej).

```cpp
if(tab.size() == 2) {
    return vector<pair<int, int>>{
        {tab.at(0), tab.at(1)}
    };
}

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

for (int x : tab) {
    // na potrzeby dzialania algorytmu traktujemy 0 jako liczbe dodatnia
    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);

if(iloczyn_dodatnie > iloczyn_ujemne) return vector<pair<int, int>>{ para_dodatnia };
else if(iloczyn_ujemne > iloczyn_dodatnie) return vector<pair<int, int>>{ para_ujemna };
else return vector<pair<int, int>> {para_dodatnia, para_ujemna};
```

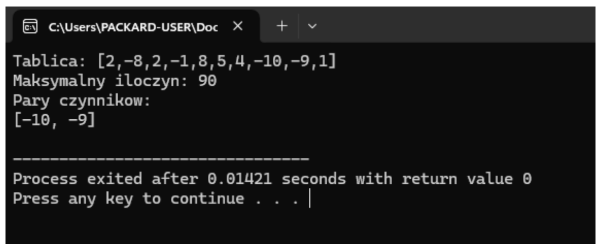
Kod 6: Kod komputerowej implementacji algorytmu o złożoności liniowej

Ta implementacja jest właściwie bezpośrednim przełożeniem schematu blokowego z dwiema drobnymi, acz znaczącymi zmianami.

- 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). Jeżeli natrafimy na taki przypadek, to największy iloczyn dają oczywiście jedyne dwa elementy.
- Traktujemy 0 jako liczbę dodatnią, by algorytm poprawnie identyfikował konkretne skrajne przypadki.

2.2.3 Wyniki

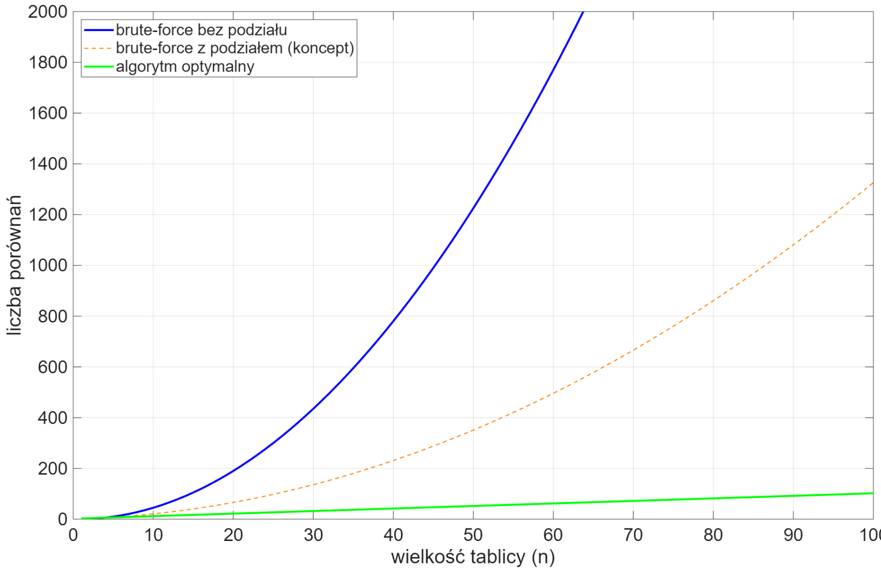
Ponownie, wyniki produkowane przez program są poprawne. Tutaj, w przeciwieństwie do poprzedniej implementacji, nie ma możliwości by pary liczb wypisały się wielokrotnie, więc nie trzeba niczego dodatkowo filtrować.



Rysunek 8: Wynik działania programu ze zoptymalizowanym algorytmem

3 Podsumowanie

Prezentuję wykres ukazujący kolosalną różnicę w złożoności czasowej algorytmów.



Rysunek 9: Wykres przedstawiajacy liczbę iteracji dla każdego z omawianych algorytmów

Świetnie obrazuje on, po co właściwie powinno się dążyć do optymalizacji algorymtów. Możemy sobie dalej wyobrazić jak dalej rosłaby liczba iteracji dla zestawu danych złożonego na przykład z setek tysięcy rekordów. Biorąc pod uwagę ogrom przetwarzanych współcześnie danych wcale nie jest to nierealny scenariusz.

3.1 Benchmarking algorytmów

Jeżeli moje obliczenia zwizualizowane na rysunku 9 są poprawne, różnice w czasie wykonania każdego z algorytmów powinny być widoczne gołym okiem (oczywiście przy odpowiednio dużym zestawie danych wejściowych), lecz oczywiście wygodniej będzie operować na konkretnych liczbach, więc z pomocą modelu językowego stworzyłem w języku C++ funkcję pozwalającą zmierzyć czas wykonania funkcji.

```cpp
template <typename T>
struct WynikBenchmarku {
    long long czas_ms;
    T wartosc;
};

template <typename F, typename... Args>
auto benchmark(F&& funkcja, Args&&... args)
-> WynikBenchmarku<typename std::result_of<F(Args...)>::type>
{
    using namespace std::chrono;
    using ReturnType = typename std::result_of<F(Args...)>::type;

    auto start = high_resolution_clock::now();

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

    auto stop = high_resolution_clock::now();

    return {
        duration_cast<milliseconds>(stop - start).count(),
        std::move(wynik)
    };
}
```

Kod 7: 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 odnotowano znacznie szybszy wzrost czasu wykonania dla niezoptymalizowanej wersji algorytmu.

Wielkość danych wejściowych	100	1000	10000	20000	50000
Czas wykonania algorytmu zoptymalizowanego [ms]	0	1	1	1	2
Czas wykonania algorytmu brute force [ms]	1	3	133	549	3284

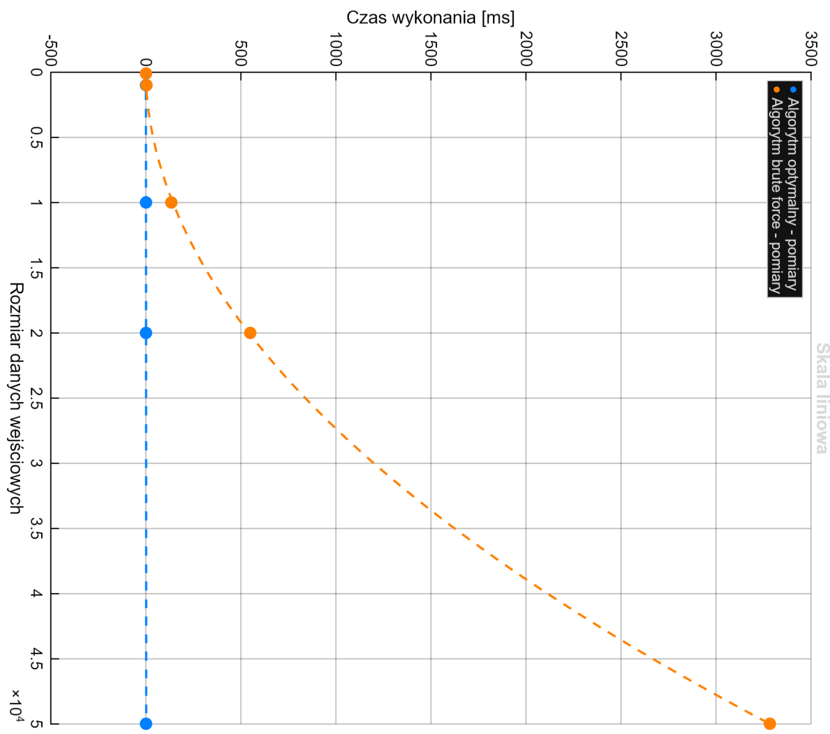
Tabela 2: Wyniki pomiarów czasu wykonania algorytmów

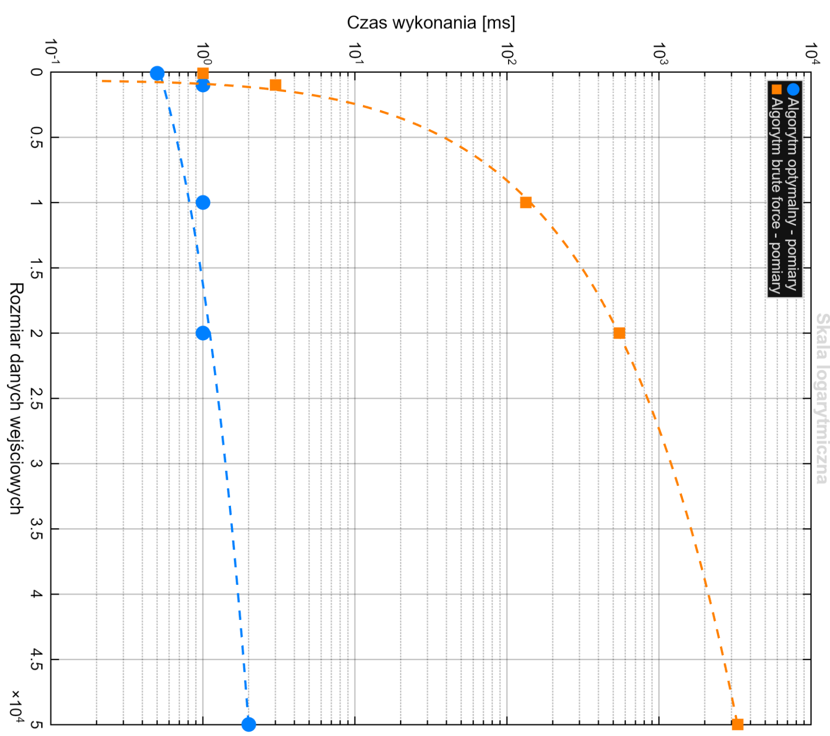
O ile dla dwóch pierwszych odczytów można snuć wątpliwości co do wpływu błędu pomiarowego, to od trzeciego - od zestawu danych składającego się z 10 tysięcy elementów - nie ma o nim mowy. Czas wykonania wersji brute force jest kolosalny w porównaniu do wersji alternatywnej.

Dokonane pomiary nanoszę na wykres i stosuję regresję do zwizualizowania wzrostu czasu wykonywania algorytmu w zależności od rozmiaru danych wejściowych. Moje dane eksperymentalne potwierdzają poprzednie założenia o tempie wzrostu czasu wraz ze wzrostem zestawu danych.

3.2 Wnioski i podsumowanie

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. Obserwacja oraz analiza matematyczna pozwoliła również znaleźć zależność, której wykorzystanie z kolei doprowadziło do zoptymalizowania algorytmu a tym samym zredukowania złożoności z $O(n^2)$ do $O(n)$.





Rysunek 10: Regresja czasu wykonania algorytmu na podstawie danych pomiarowych

Spis rysunków