AISD lista 2

Dominik Szczepaniak

July 10, 2024

1 Zadanie 2

Mamy n odcinków z punktami końcowymi $< p_j, k_j >$ i mamy ułożyć algorytm który zmaksymalizuje ilość wybranych odcinków, tak, żeby żaden się nie przecinał ze sobą.

Linie leżą na osi OX

Ten problem jest izomorficzny z problemem przydziału prac - praca zaczyna się o czasie p_j a kończy o czasie k_j. Od tej pory tak będę opisywał ten problem

Algorytm:

1. Sortujemy prace po czasie zakończenia, a później po czasie startu

```
lambda (a, b) -> a.1 < b.1 || (a.1 == b.1 && a.0 < b.0)
```

2. Wybieramy po kolei prace idąc od początku listy i wstawiamy tylko te które możemy (czas zakończenia poprzedniej pracy jest mniejszy równy od czasu startu obecnie rozpatrywanej pracy).

Poniżej przedstawię dowód, który na pierwszy rzut oka wydaje się poprawny, ale nie jest:

> Załóżmy nie wprost, że istnieje jakiś wynik który jest większy od naszego. Czyli istnieje jakaś praca w naszym zbiorze którą można usunąć i w jej miejsce wstawić dwie inne prace.
>
> No ale to jest niemożliwe, ponieważ jedna z tych prac musiałaby się konczyć wcześniej niż ta praca którą mamy w naszym zbiorze, więc zostałaby ona wybrana szybciej przez algorytm zachłanny, także nasz algorytm jest poprawny.

Dlaczego powyższy dowód nie jest poprawny? Ponieważ zakładamy, że lepszy dobór prac istnieje po usunięciu tylko jednej pracy z naszego zbioru, a może być tak, że trzeba usunąć k prac, aby wstawić k+1 prac.

Poprawny dowód powinien uwzględniać zbiory prac:

> Weźmy zbiór A, który będzie naszym zbiorem prac który wybrał algorytm. Oznaczmy B jako najlepszy zbiór prac.
>
> Załóżmy nie wprost, że $|B| > |A|$.
>
> Weźmy więc pierwszy odcinek który różni się w zbiorze B od odcinka w zbiorze A.
>
> Mamy następujące możliwości:
>
> 1. **Koniec odcinka z B jest mniejszy niż koniec odcinka z A**
>
>   Ten przypadek nie jest możliwy, ponieważ jeśli koniec odcinka z B jest mniejszy niż koniec odcinka z A to odcinek z B zostałby wybrany przez algorytm zachłanny zamiast odcinka z A.
>
> 2. **Koniec odcinka z B jest większy niż koniec odcinka z A**
>
>   To dopasowanie może być albo takie samo albo gorsze. Dlaczego? Ponieważ potencjalnie kolejny odcinek w A może się zacząć szybciej niż odcinek z B się skończy, więc zabierzemy miejsce. Jeśli B nie zabierze miejsca to znaczy, że kolejny z odcinków wybranych w B może być taki sam jak te wybrane przez A.
>
> 3. **Koniec odcinka z B jest równy końcowi odcinka z A**
>
>   W tym przypadku możemy iść indukcyjnie, ponieważ jeśli końce są takie same to możemy usunąć ten odcinek z B i wstawić odcinek z A, a wynik się nie zmieni. Wtedy jeśli natkniemy się na kolejny odcinek który jest różny to możemy zastosować jeden z dwóch poprzednich przypadków lub znowu zamienić odcinek z B na odcinek z A jeśli kończą się w tym samym miejscu.
>
> W każdym z przypadków mamy sprzeczność, że zbiór B jest lepszy od zbioru A, więc nasz algorytm jest poprawny.

2 Zadanie 3

Na wejściu mamy dwie liczby naturalne a i b.

Chcemy przedstawić ten ułamek jako sumę **różnych** ułamków o postaci 1/x, gdzie $x \in \mathbb{N}$.

Mamy udowodnić, że algorytm zachłanny zawsze daje rozwiązanie oraz zastanowić się czy zawsze daje optymalne rozwiązanie.

1. **Algorytm zachłanny zawsze daje rozwiązanie**

Wybierając jakikolwiek ułamek który jest większy od 0 oraz mniejszy od $\frac{a}{b}$ będziemy zmniejszać licznik o jakąś niezerową wartość. Jeśli nasz ułamek który odejmiemy przedstawimy jako $\frac{1}{x} = \frac{b}{b*x}$, a nasz początkowy ułamek jako $\frac{a}{b} = \frac{ax}{bx}$ a później odejmiemy, to dostaniemy:

$\frac{ax-b}{bx}$, co jest mniejszą wartością niż $\frac{a}{b}$, ponieważ b jest większe od 0. Także w skończonej liczbie kroków dojdziemy do 0, więc nasz algorytm się zakończy.

2. **Czy to rozwiązanie jest zawsze optymalne?**

$\frac{9}{20} = \frac{1}{3} + \frac{1}{9} + \frac{1}{180}$

$\frac{9}{20} = \frac{1}{4} + \frac{1}{5}$

Nie jest.

3 Zadanie 4

Teza: Kazda liczbe naturalna mozna przedstawic jako suma liczb fibonaciego, gdzie kazda jest uzyta co najwyzej raz i nie używamy żadnych dwóch kolejnych liczb fibonnaciego.

Dowod przez indukcje:

n = 1:

1 mozemy przedstawic jako 1:

Krok:

Załóżmy, że dla n teza zachodzi.

Pokażmy też, że zachodzi dla n+1.

Mamy dwie możliwości:

1. **Do liczby n użyliśmy jednej jedynki**

   Jeśli użyliśmy jednej jedynki to nie użyliśmy lub użyliśmy 2.

   (a) Jeśli nie użyliśmy dwójki to dokładamy jedynkę i z dwóch jedynek robimy dwójkę. Jeśli użyliśmy trójkę to łączymy w kolejną liczbę.

Robimy tak indukcyjnie, aż nie będzie użyta kolejna liczba fibbonaciego (wtedy teza zachodzi).

Przykładowo:

1, 3, 8, 21 -> 1, 1, 3, 8, 21 -> 2, 3, 8, 21 -> 5, 8, 21 -> 13, 21 -> 34

(b) Jeśli użyliśmy dwójkę to możemy złączyć 2 i nową jedynkę w 3 i później znowu kolejno łączyć następujące po sobie liczby, aż założenie będzie się zgadzać

1, 2, 5, 13 -> 1, 1, 2, 5, 13 -> 1, 3, 5, 13 -> 1, 8, 13 -> 1, 21

2. **Do liczby n nie użyliśmy ani jednej jedynki**

   Wtedy używamy jednej jedynki i dostajemy n+1.

Stąd możemy ułożyć algorytm zachłanny który będzie brał kolejne liczby Fibonnaciego (od największej) i odejmował ją od naszej wartości n i z tego dostaniemy odpowiedź. Wiemy, że tak można zrobić, ponieważ za pomocą liczb fibonnaciego można przedstawić każdą liczbę.

4 Zadanie 5

Jeśli będziesz zachłannie wybierał do i-tego dnia i i-tego dnia ci brakuje kasy to musimy jakoś wytrzasnąć te pieniądze. Możemy zasymulować wybór w przeszłości.

Kiedy się najbardziej opłaca? Gdy kara była najmniejsza

Wystarczy brać modulo 100 wszystko - wiadomo czemu.

Załóżmy, że cofamy się w przeszłość. Wiemy, że kiedyś tam wydaliśmy X monet. Chcemy dostać nowy zastrzyk gotówki w jedynkach. W takim razie musimy zapłacić 100, i wtedy dostaniemy 100 - X jedynek. No ale skoro zapłaciliśmy te X, to teraz mamy je w kieszeni (bo zamiast płacić X zapłaciliśmy jednak 100), no i dostaliśmy 100 - X, więc łącznie mamy 100-X + X = 100 jedynek.

Algorytm:

Czyli to co wystarczy zrobić, to iść po kolei.

Jeśli nie ma kasy, wybieramy najmniejsze W spośród poprzednich (priority queue) i usuwamy je (więcej nie możemy go użyć, bo kasa nam wyda 100, a nie jedynki)

Jeśli jest kasa to idziemy dalej.

Czy się kończy?

Mamy dwa przypadki:
a) stać nas na przejście dnia.
b) dobieramy "z przeszłości"
W obu przypadkach przejdziemy przez następny dzień. Indukcyjnie dojdziemy do końca.

Daje optymalny wynik?
Załóżmy, że mamy jakiś wynik A, który jest mniejszy od naszego B.
Czyli istniał dzień w którym A dobrało, a my nie dobraliśmy.
No ale skoro A dobrało jedynki, a my nie dobraliśmy, to z założenia algorytmu A musiało dobrać w dniu w którym W nie jest minimalne, czyli wynik musiał być większy od naszego.
W takim razie mamy sprzeczność, ponieważ wynik jest większy, a miał być niższy.
Miejsca można zapisywać, więc znamy w których miejscach musimy pytać o i-tego mina na przedziale
Most basic rozwiązanie to jest priority queue

5 Zadanie 6

Teza: Ścieżka prosta może zawierać co najwyżej dwa liście.
Dowód: Załóżmy nie wprost, że ścieżka prosta zawiera więcej niż dwa liście. Jeśli mamy więcej niż dwa liście, to dla przynajmniej dwóch liści musieliśmy z nich wyjść. W jednym z nich mogliśmy zacząć, ale do drugiego musieliśmy wejść, a w drzewie prowadzi tylko jedna droga do liścia, więc jeśli z niego wyszliśmy to odwiedziliśmy ten sam wierzchołek dwa razy, co jest sprzeczne z definicją ścieżki prostej, stąd teza jest prawdziwa.

Algorytm:
Jeśli k jest nieparzyste to odejmujemy 1 od k i dodajemy 1 do wyniku.
Przechodzimy DFS który liczy głębokość dla poddrzew. Jeśli głębokość poddrzewa jest mniejsza równa k/2 to dodajemy ten wierzchołek do wyniku.

Teza: Dla parzystych możemy pomalować wszystkie wierzchołki których depth od liścia wynosi k/2, czyli możemy pomalować wszystkie wierzchołki których dla k-2 wszyscy sąsiedzi byli pomalowani.

Jeśli mamy dowolną ścieżkę to musi się ona kończyć w liściach. W takim razie jeśli są pokolorowane tylko wierzchołki których depth poddrzewa jest <= k/2 to po obu stronach ścieżki mamy pokolorowanych tylko k/2 wierzchołków, więc sumuje się to do k.

Teza: Jeśli mamy k nieparzyste to możemy pokolorować tylko jeden wierzchołek.

Dowód: Weźmy dowolną ścieżkę $< v1, v2, ..., v_{n-1}, v_{n-2} >$. Wiemy, że dla k-1 pomalowanych jest (k-1)/2 wierzchołków z początku i końca. Jeśli pomalujemy dwa wierzchołki to będziemy musieli pomalować $v3$ i v_{n-2} w tej ścieżce, co przekroczy dozwoloną ilość kolorów. Także nie możemy pomalować dwóch wierzchołków, stąd maksymalna liczba to 1.

6 Zadanie 7

Algorytm:

Usuńmy krawędź e z grafu i puśćmy dfs z jednego z końców tej krawędzi odwiedzając tylko krawędzie o wadze mniejszej niż waga tej krawędzi. Jeśli odwiedzimy wierzchołek po drugiej stronie to krawędź nie należała do MST tego grafu, ponieważ mogliśmy dotrzeć do wierzchołka krawędzią o mniejszym koszcie z jakiegoś innego wierzchołka.

Jeśli waga krawędzi e należącej do cyklu C jest większa od pozostałych wag krawędzi, to ta krawędź nie może należeć do MST. Załóżmy, że e nie jest maksymalną krawędzią na żadnym cyklu w grafie G i nie należy do MST. Przypadki:

1. Krawędź e nie leży na żadnym cyklu, stąd musi należec do MST.
2. krawędź e leży na cyklu C: Weźmy więc MST i dołóżmy do niego krawędź e -> z tego tworzy sie cykl, bo e nie należało do MST. W tym drzewie rozpinającym musi istnieć jakaś krawędź maksymalna e'. Z założenia e nie jest maksymalną krawędzia na jakimś cyklu, więc e'

ma większą wagę od e. Po usunięciu e' otrzymamy MST o mniejszej wadze, więc dochodzimy do sprzeczności, że e nie należało do MST.

7 Zadanie 8

Osobny plik zad8.pdf

8 Zadanie 9

gdzie-chce = []
gdzie-jest = []
for i in pi:
gdzie-jest[pi[i]] = i
for i in sigma:
gdzie-chce[sigma[i]] = i
Fakt 1.

Jeśli idąc po kolei w sigma (gdzie poprzednie elementy już naprawiliśmy) napotkamy element który nie jest na swoim miejscu to chce on iść gdzieś na prawo.

Dowód:

Jeśli naprawiliśmy wszystkie poprzednie elementy, to są one na swoich pozycjach, a obecna pozycja nie jest pozycją na którą chce iść liczba która stoi na tej pozycji, więc musi ona chcieć iść gdzieś na prawo.

Przykład algorytmu:
pi = 1, 5, 2, 3, 4
sigma = 2, 1, 3, 4, 5
1. Idziemy forem po sigmie
Trafiamy na 2:
Na miejscu 2 jest 1
Czy 1 chce się zamienić z 2? Nie, bo 2 stoi dalej niż 1 chce iść.
1 przekieruje 2 na 5.

Czy 5 chce się zamienić z 2? Tak, bo 5 chce iść dalej niż stoi 2, więc jest jej to bez różnicy.
pi = 1, 2, 5, 3, 4
sigma = 2, 1, 3, 4, 5

Czy teraz 1 chce się zamienić z 2? Tak, bo 2 jest na miejscu gdzie 1 chce dotrzeć.

Skończyły nam się liczby, więc zwiększamy fora.

Mamy:

pi = 2, 1, 5, 3, 4

sigma = 2, 1, 3, 4, 5

Trafiamy na 1:

1 jest na swoim miejscu więc idziemy dalej.

Trafiamy na 3:

Na miejscu 3 stoi 5.

Czy 5 chce się zamienić z 3? Tak, bo 3 stoi bliżej niż chce iść 5.

Zamieniamy

Mamy:

pi = 2, 1, 3, 5, 4

sigma = 2, 1, 3, 4, 5

Idziemy dalej forem:

Trafiamy na 4:

Na miejscu 4 stoi 5.

Czy 5 chce się zamienić z 4?

Tak, bo 4 stoi bliżej miejsca 5 niż obecnie stoi 5.

Zamieniamy 4 i 5.

4 jest na swoim miejscu, dalej for.

Trafiamy na 5:

5 stoi na swoim miejscu - idziemy dalej.

Kończy się for - kończy się algorytm.

czyli algos:

```
gdzie-chce = []
gdzie-jest = []
for i in pi:
    gdzie-jest[pi[i]] = i
for i in sigma:
    gdzie-chce[sigma[i]] = i
wynik = 0
for i in sigma:
    if(pi[i] == sigma[i]):
        continue
```

```
obecny = pi[i]
odwiedzone = stack()
odwiedzone.push(pi[i])
#odwiedzone musi chciec isc w prawo, wynika to z faktu 1
while(!odwiedzone.empty()):
    rozwazamy = odwiedzone.top()
    if(gdzie-chce[rozwazany] >= gdzie-jest[obecny]):
        wynik += abs(gdzie-jest[obecny] - gdzie-jest[rozwazany])
        odwiedzone.pop()
        swap(pi[rozwazany], pi[obecny])
        swap(gdzie-jest[rozwazany], gdzie-jest[obecny])
    else: #jesli rozwazany chce skonczyc gdzies blizej niz jest obec
        odwiedzone.push(gdzie-chce[rozwazany])
print(wynik)
```

Dowód:

1. Czy się skończy?

   Zauważmy, że jedyna sytuacja w której nasz algorytm się nie skończy, to gdy odwiedzone nigdy nie będzie pusty w jakimś momencie.

   Zastanówmy się czy to możliwe.

   Nasz algorytm dla jakiejś liczby X chcę się zamienić z liczbą Y która jest na jego miejscu. Jeśli liczba Y nie chce się zamienić z liczbą X (bo liczba X stoi dalej niż liczba Y musi przejść), to przekierowuje liczbę X do zamiany z jakąś liczbą Z która stoi na polu liczby Y.

   No ale skoro Z stoi na polu liczby Y, to musi stać gdzieś pomiędzy liczbami Y i X (bo pole Y było bliżej niż stoi obecnie X - dlatego Y nie zgodziło się na wymiane). W takim razie możemy pójść sobie indukcyjnie i za każdym razem będziemy skracać przedział między Y i X, aż dojdziemy do sytuacji w której potencjalnie X może przesunąć się tylko jedno miejsce w prawo, ale późniejsze wymiany wszystkie będą przebiegać pomyślnie, ponieważ jakaś liczba którą wcześniej rozważaliśmy wskazała na to pole, więc będzie się chciała wymienić. W takim razie

nigdy nie dojdzie do sytuacji, że stack odwiedzone nigdy się nie zwolni.

2. Czy zwraca optymalny wynik?

   Załóżmy, że mamy jakiś wynik A, który jest lepszy od naszego wyniku B.

   Jeśli wynik A jest lepszy od wyniku B to musiała nastąpić jakaś wymiana między liczbami X i Y w A która była tańsza niż wymiana między liczbami X i Y w B.

   Zauważmy, że wymiana X i Y w B zawsze będzie przybliżać obie liczby do swojego miejsca docelowego. W takim razie każda z liczb Z w B przejdzie dokładnie dystans który dzieli ją między miejscem w którym się znajduje a miejscem docelowym.

   W takim razie mamy sprzeczność, że istnieje wynik A który jest lepszy od wyniku B.

Jaka złożoność?
Nasza złożoność wynosi O(n). Dlaczego?
Zauważmy, że dla każdej pozycji w forze chcemy umieścić liczbę docelową na to miejsce.
Rozważmy przypadki

1. Liczba X i Y chętnie się ze sobą wymieniają.

   W takim razie dla liczby X wykonujemy tylko jedną operację - więc dla wszystkich liczb wykonamy ich n - stąd złożoność O(n).

2. Liczby X i Y nie chcą się wymienić.

   W tym przypadku musimy użyć kolejnej liczby - Z. Jeśli X i Z chcą się wymienić w takim razie zamienimy je, a później zamienimy X i Y i w ten sposób będziemy mieć gotowe dwie liczby - X oraz Y, a wykonaliśmy tylko dwie zamiany - Z i X oraz X i Y.

Zauważmy, że punkt drugi możemy rozszerzać indukcyjnie, tj. jeśli a1 nie chce się zamienić z a2, oraz a3 (liczba wskazywana przez a2) nie chce się zamienić z liczbą a1, to możemy przejść do indukcji, w której a1 = X oraz a2 = Y i wtedy musimy znaleźć sobie kolejnego Z. Wiemy, że kiedyś się to

skończy i wtedy mamy, że X i Z się zamienią, później zamieni się jakieś Y_k, później Y_{k-1}, ..., Y_1, więc zrobimy poprawnie k liczb, czyli dokładnie tyle zamian ile chcieliśmy.

Co dowodzi, że algorytm działa liniowo.

9 Zadanie 10

Algorytm z wykładu wybiera te rodziny które dają najniższą cene za pokrycie jednego elementu.

Załóżmy, że mamy uniwersum U o n elementach, gdzie n jest potęgą liczby 2, t.j. $n = 2^k$ dla pewnego k.

Budujemy rodziny w następujący sposób:

1. Dla każdego elementu $u \in \mathbb{U}$ tworzymy podzbiór S_u który zawiera tylko ten element. $c(S) = n$ dla każdego z tych zbiorów.

2. Tworzymy podzbiór który zawiera wszystkie elementy z U za wyjątkiem jednego, a koszt ustalamy na logn. Będzie n takich podzbiorów, po jednym dla każdego elementu z U.

3. Tworzymy zbiór który zawiera wszystkie elementy z U i dajemy koszt równy $logn^2$.

Algorytm wybierze dowolny ze zbiorów z punktu 2, bo koszt jest bardzo niski. Następnie wybierze punkt 3.

Czyli koszt algorytmu to logn + $logn^2$.

Optymalny koszt to $logn^2$

Są blisko logn gorsze tak jak chciało zadanie.

==========

tworzymy zbiory o wielkości 1, gdzie każdy ma rozmiar 1/n, 1/(n-1), 1/(n-2)..., 1/1

i zbior o wielkosci U z kosztem 1 + E

1 + E / n > 1/n

Wybierzemy wiec wszystkie zbiory jednoelementowe. Koszt laczny:

1/n + 1/(n-1) + 1/(n-2) + ... + 1/1 logn

$\frac{1+E}{1+E+1/n+1/(n-1)+...+1/1}$ logn c.n.w

10 Zadanie 11

Liczba 1/|E| nie ma znaczenia, zawsze będzie taka sama, bo MST musi mieć tyle samo wierzchołków, więc krawędzi mamy też tyle samo.

Chcemy wybrać takie drzewo rozpinające w którym wagi są jak najbliżej siebie, t.j. różnica między najmniejszą a największą wagą w drzewie rozpinającym jest minimalna.

1. Posortujmy krawędzie po wadze.

2. Dla kazdego i po kolei rozwazamy mst zaczynajace sie w tej krawędzi.

- Dla danego i idziemy jednocześnie w prawo i lewo po posortowanych krawędziach i dodajemy je do drzewa rozpinającego jeśli możemy (możemy dodać jeśli jeden z końców krawędzi nie był jeszcze nigdzie dodany)

- Kończymy gdy będzie wystarczająca ilość krawędzi w drzewie rozpinającym.

a) Priorytezujemy krawędź z lewej strony jeśli krawędź z prawej strony ma większą różnicę do krawędzi startowej. I na odwrót.

3. Dla kazdego utworzonego MST liczymy jego funkcje i zapisujemy w jakies zmiennej wynik.

4. Zwracamy wynik

zlozonosc:

O(|E|*log|E|) - sortowanie

O(|E|) - przejscie po krawedziach

O(|E|) - dodawanie krawedzi dla kazdego przejscia po krawedziach

O(log|E|) - find and union

łącznie:

$O(|E| * log|E| + |E| * |E| * log|E|) = O(|E|^2 * log|E|)$

Dlaczego to działa?

Chcemy uzyskać wynik który ma jak najmniejszą różnicę między maksymalną a minimalną krawędzią, stąd chcemy brać krawędzie jak najbliższe tej którą wybraliśmy początkowo.

Załóżmy, że nasz program nie znalazł optymalnego wyniku.

Wtedy istnieje jakiś zbiór krawędzi który dał lepszy wynik.

Załóżmy, że nasz program wybrał krawędzie $A =< A_1, A_2, ..., A_n >$, a optymalny wynik wybrał krawędzie $B =< B_1, B_2, ..., B_n >$.

Wybierzmy krawędź z B która jest jak najbliżej średniej B.

Niech to będzie nasza krawędź startowa.

Ponieważ chcemy brać krawędzie które są jak najbliżej średniej, to będziemy brać te które są jak najmniej oddalone od średniej. Czyli w optymalnym

wyniku będziemy się rozszerzać w prawo i lewo i brać ten wynik który leży bliżej średniej.

No ale to robi dokładnie nasz algorytm, czyli nasz algorytm jest optymalny.

vector<int> Parent(MAX+5);
void unite(int x, int y)
Parent[Parent[x]] = y;
int find(int v)
if (v == Parent[v])
return v;
return Parent[v] = find(Parent[v]);
bool same(int x, int y)
return find(x)==find(y)?true:false;
for i in 0..n
Parent[i] = i;
Algorytm:

```
vector<int> parent, rank;

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
```

```
            rank[a]++;
        }
    }

    struct Edge {
        int u, v, weight;
        bool operator<(Edge const& other) {
            return weight < other.weight;
        }
    };

    int n;
    vector<Edge> edges;

    int costMin = INF;
    parent.resize(n);
    rank.resize(n);

    sort(edges.begin(), edges.end());
    vector<Edge> wynik;
    for(int i = 0; i<edges.size(); i++){
        //wyczysc dane z find and union
        for (int j = 0; j < n; j++)
            make_set(j);
        int lewy = 0;
        int prawy = 0;
        int ilosc_krawedzi = 0;
        vector<Edge> result;
        int nasza_waga = edges[i].weight;
        while(ilosc_krawedzi < n){
            if(find_set(lewy.u) == find_set(lewy.v)){
                lewy--;
                continue;
            }
            if(find_set(prawy.u) == find_set(prawy.v)){
                prawy++;
                continue;
            }
```

```
            if(nasza_waga - edges[lewy].weight > edges[prawy].weight - nasza
                union_sets(edges[prawy].u, edges[prawy].v);
                result.push_back(edges[prawy]);
                ilosc_krawedzi++;
                prawy++;
            }
            else{
                union_sets(edges[lewy].u, edges[lewy].v);
                result.push_back(edges[lewy]);
                ilosc_krawedzi++;
                lewy--;
            }
        }
        int cost_total = 0;
        for (Edge e : result) {
            cost_total += e.weight;
        }
        double srednia = cost_total / (n-1);
        int roznica = 0;
        for (Edge e : result) {
            roznica += abs(double(e.weight) - srednia);
        }
        if(roznica < costMin){
            costMin = roznica;
            wynik = result;
        }
    }
```