Lista 2

tags: `AiSD`, `lista2`

Zadanie 2

```
def solve(S):
    sorted <- tablica posortowanego zbióru S według prawych końców odcinków

    most_right = -infinity
    res = {}
        for s in sorted:
            if s.left > most_right:
                most_right = s.right
                res.add(s)

return res
```

Poprawność algorytmu

Pokażemy indukcyjnie równoliczność rozwiązania względem rozwiązania algorytmu optymalnego Opt.

Niech Opt = { ${o_1, o_2, ... , o_m}$ } oraz A to zbiór wynikowy naszego algorytmu. Załóżmy, że Opt również jest posortowany wg. końcowych krawędzi.

Indukcja T(n): Dla każdego n <= m istnieje ${a_i}$ należące do A, którego prawy koniec jest niemniejszy niż prawy koniec ${o_i}$

Baza(n=1) Algorytm wybiera pierwszy odcinek z brzegu w posortowanej tablicy. Zatem prawy koniec tego odcinka (jako, że jest najmniejszy w zbiorze) może byc co najwyżej równy lub mniejszy niż prawy koniec ${o_1}$

Krok (T(n) => T(n+1)) Z założenia indukcyjnego wiemy, że odcinek a_{n} kończył się przed o_{n}. Jako, że o_{n+1} kończy się później niż o_{n}, który z kolei kończy się później lub równo do a_{n}, to o_{n+1} nie należał do A kiedy wybierany był element a_{n+1}. Jeśli o_{n+1} nie został wtedy wybrany, to oznacza że wybrany odcinek a_{n+1} kończył się wcześniej lub w tym samym miejscu.

Złożoność Czasowa

- Sortowanie $O(n*logn)$
- Iteracja po odcinkach $O(n)$

Złożoność Pamięciowa

Działamy na tablicy i zbiorze o sumarycznym rozmiarze 2n, czyli złożoność pamięciowa to $O(n)$

Zadanie 3

Będziemy spamiętywać kolejne ułamki postaci $\frac{1}{\lceil b/a \rceil}$ i redukować nimi ułamek $\frac{a}{b}$ dopóki go nie wyzerujemy.

Zacznijmy od wyliczenia ułamka $\frac{a'}{b'}$ będącego wynikiem jednego kroku algorytmu.

$\frac{a}{b} - \frac{1}{\lceil b/a \rceil}= \frac{a\lceil b/a \rceil - b}{b\lceil b/a \rceil}$

Wiemy, że

1. $\lfloor -x \rfloor = -\lceil x \rceil$
2. n mod d $= n - \lfloor n/d \rfloor d$

Stąd otrzymujemy: $\frac{a\lceil b/a \rceil - b}{b\lceil b/a \rceil}=_1 \frac{-a\lfloor -b/a \rfloor - b}{b\lceil b/a \rceil}=_2 \frac{(-b)\bmod a}{b \lceil b/a \rceil} = \frac{a'}{b'}$

Pokażemy teraz, że algorytm się ewentualnie skończy. Zauważmy, że $a' = (-b)\bmod a < a$, czyli licznik w każdym kroku algorytmu będzie malał. Zauważmy też, że zarówno licznik jak i mianownik są nieujemnymi liczbami całkowitymi.

Pokażmy również, że odejmowany ułamek jest unikalny względem wszystkich poprzednich. W tym celu wykażemy poniższą nierówność (odejmowany ułamek jest ostro większy od wyniku odejmowania):

$\frac{a'}{b'} = \frac{(-b)\bmod a}{b \lceil b/a \rceil} < \frac{1}{\lceil b/a \rceil}$

Przeprowadźmy odpowiednie obliczenia: $\frac{(-b)\bmod a}{b} < 1$ $(-b)\bmod a < b$

Wiedząc, że $a < b$ (ponieważ mianownik jest niemalejący) oraz $(-b)\bmod a < a$ powyższa nierówność zachodzi. Czyli udowodniliśmy, że: $\frac{a'}{b'} < \frac{1}{\lceil b/a \rceil}$

W każdym kroku algorytmu odejmujemy ułamek niemniejszy niż ten, który otrzymalismy z poprzedniego kroku zatem:

$\frac{1}{\lceil b'/a' \rceil} \le\frac{a'}{b'} < \frac{1}{\lceil b/a \rceil}$

$\frac{1}{\lceil b'/a' \rceil} < \frac{1}{\lceil b/a \rceil}$

Czyli każdy kolejny odejmowany ułamek jest mniejszy od poprzedniego, a zatem jest też unikalny.

```
def solve(a, b):
    res = []

    while a != 0:
        c = ceil(b / a)
        a = a*c - b
        b = b*c

        res.append(c)

    return res
```

Złożoność obliczeniowa - $O(a)$

Czy algorytm jest optymalny?

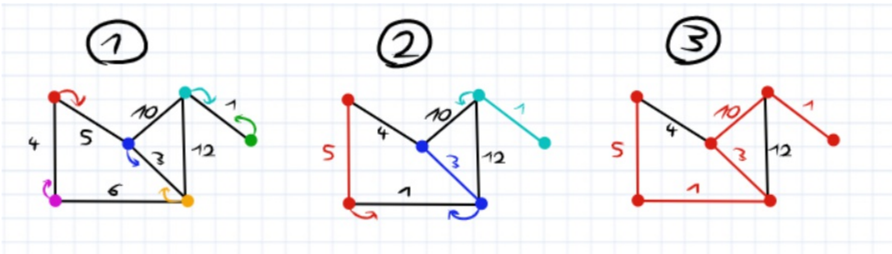
Nie jest. Przykładowo optymalny rozkład na ułamki dla $\frac{733}{4692}$ to $\frac{1}{12} + \frac{1}{23} + \frac{1}{34}$

W tym przypadku algorytm zwróciłby $\frac{1}{7} + \frac{1}{75} + \frac{1}{30412} + \frac{1}{1040470550}$

Zadanie 5

Algorytm Boruvki

1. Tworzymy graf superwierzchołków
2. Dla każdego superwierzchołka znajdujemy najlżejszą wychodzącą z niego krawędź
3. Superwierzchołki połączone ścieżką scalamy w jeden superwierzchołek
4. Powtarzamy aż zostanie tylko jeden superwierzchołek



Widzimy, że algorytm zawsze się kończy, jako że w każdym kroku łączymy ze sobą co najmniej połowę superwierzchołków (każdy wierzchołek dostaje krawędź, ale może ona być wspólna z innym superwierzchołkiem). Skoro z każdym krokiem algorytmu liczba superwierzchołków maleje o jakąś liczbę naturalną to algorytm kiedyś się zakończy.

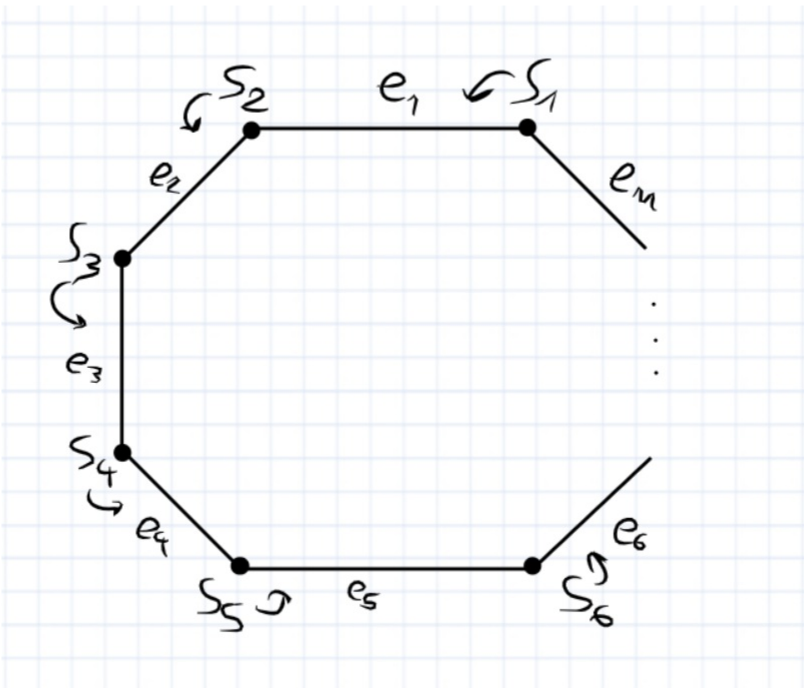
Zacznijmy od udowodnienia, że graf wynikowy działania Boruvki jest drzewem rozpinającym.

Spójność

Zaczynamy od pojedynczych wierzchołków, które są spójne i łączymy je krawędziami otrzymując wierzchołki spójne. Algorytm kończy się, gdy cały graf jest jednym superwierzchołkiem

Acykliczność

Załóżmy nie wprost, że w którymś kroku algorytmu powstał cykl. Rozważmy dowolny taki superwierzchołek przy którego tworzeniu powstał cykl – nazwijmy go S.



Widzimy, że każdy superwierzchołek wybiera inną krawędź (gdyby tak nie było to nie dostalibyśmy cyklu bo cykl złożony z n wierzchołków musi mieć n krawędzi, czyli jeśli któryś wierzchołek wybierze krawędź wspólnie z innym wierzchołkiem to cykl ma n-1 krawędzi)

Wiemy, że boruvka wybiera zawsze najlżejsze krawędzie zatem mamy $e_1 > e_2 > e_3 > e_4 > e_5 > e_6 > ... > e_n > e_1$ Otrzymujemy sprzeczność: $e_1 > e_n > e_1$

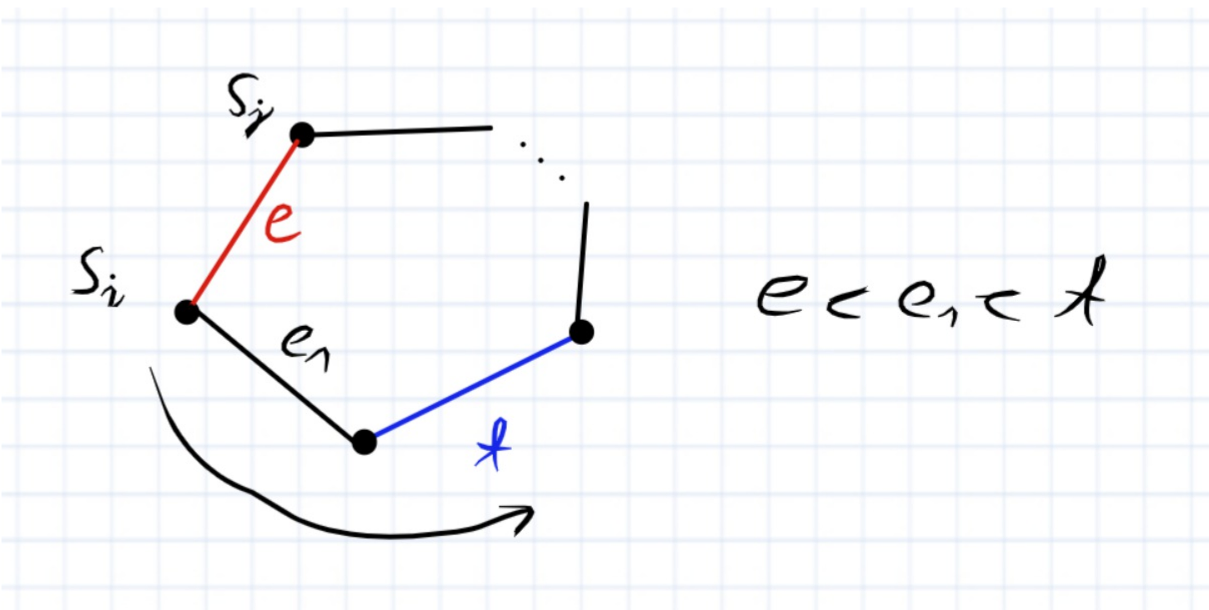
Drzewo rozpinające

Ponieważ w każdym kroku algorytmu zbiór superwierzchołków jest zbudowany ze wszystkich wierzchołków w grafie to graf wynikowy również zawiera wszystkie wierzchołki. Wiemy już, że ten graf jest spójny i acykliczny, czyli jest drzewem rozpinającym.

Graf wynikowy jest MST

Dowód indukcyjny W(k) := w każdym kroku algorytmu obecny stan grafu zawiera się w jakimś MST M_k.

Baza (k=0) W pierwszym kroku algorytmu mamy tylko wierzchołki i pusty zbiór krawędzi, taki graf zawiera się w każdym MST. ***Krok (W(k) => W(k+1))*** Załóżmy nie wprost, że nieprawdą jest W(k+1). Wtedy istnieje krawędź $e \in T_{k+1}$ i $e \notin M$, którą algorytm dodał w tym kroku. Dodając tą krawędź do M w M powstanie cykl (bo M to drzewo).



Skoro e < f to podmieńmy je w M, nawijmy ten graf M'. Widzimy, że M' nadal jest drzewem rozpinającym, ale ma mniejszą wagę niż M. Sprzeczność z założeniem, że M jest MST.

Zadanie 6

Algorytm

Niech e = (v, u)

- Przeszukujemy graf przy użyciu zmodyfikowanego algorytmu BFS, który porusza się jedynie po krawędziach lżejszych od e.
- Jeśli BFS dotrze do wierzchołka u algorytm zwraca prawdę, jeśli nie to zwracany jest fałsz.

Złożoność czasowa

Przyjmując najgorszy przypadek BFS odwiedzi każdy wierzchołek i (prawie) każdą krawędź, więc będzie miał złożoność klasycznego algorytmu BFS tj. $O(|V| + |E|)$.

Użyteczne lematy

Cycle property Dla każdego cyklu w grafie jeśli waga pewnej krawędzi e na tym cyklu jest większa od wszystkich pozostałych to e nie należy do MST.

Cut property Jeśli dla jakiegoś cięcia grafu G, tj. podziału na zbiory wierzchołków A i B, jakaś krawędź e o końcach w A i B ma wagę ściśle mniejszą od wszystkich innych takich krawędzi, to należy ona do MST.

Dowód: Załóżmy nie wprost, że istnieje takie MST, które nie zawiera e. Dodajmy je do tego drzewa, a otrzymamy cykl, czyli mamy na nim inną krawędź łączącą zbiory A i B, oznaczmy ją e'. Wiemy z założenia że $w(e) < w(e')$. Usuńmy zatem e' z tego drzewa, a otrzymamy drzewo rozpinające o mniejszej wadze, co oznacza sprzeczność z założeniem, że jest to MST.

Dowód poprawności

Rozważmy przypadki

- BFS osiągnał wierzchołek u To znaczy, że przeszliśmy z wierzchołka v do wierzchołka u chodząc jedynie po lżejszych od e krawędziach, czyli e leży na cyklu i jest w nim krawędzią o największej wadze. Z *cycle property* wiemy, że e nie należy do żadnego MST.

- BFS nie znalazł wierzchołka u Czyli istnieje zbiór wierzchołków, do którego zmodyfikowany BFS nie mógł się dostać. Ta sytuacja mogła wystapić z dwóch powodów: e była jedyną krawędzią lub istniały inne krawędzie poza nią, ale jako, że BFS po nich nie przeszedł to musiały być cięższe od e (wg. założenia o różnych wagach). Co za tym idzie graf można rozspójnić ucinając wszystkie takie krawędzie. Taki zbiór krawędzi jest cięciem. Korzystając teraz z *cut property* oraz informacji, że e jest najlżejszą krawędzią wiemy, że e należy do MST.

Zadanie 9

Idea

Dążymy do ułożenia najcięższych liści możliwie nabliżej korzenia jednocześnie uważając na to jak zmieniają się wagi w zależności od odległości tych liści. W algorytmie wykorzystamy kolejkę priorytetową do budowania drzewa od dołu w górę łącząc lżejsze poddrzewa w cięższe.

Algorytm

1. Wypełniamy kolejkę priorytetową wierzchołkami v_i takimi, że $c(v_i) = w_i$
2. Dopóki K nie zawiera dokładnie jednego elementu: a. Ściągamy dwa najlżejsze elementy v, u z kolejki b. Tworzymy rodzica p, którego dziećmi są v i u c. Ustawiamy wagę $c(p)$ <- $c(v)$ + $c(u)$ d. Wrzucamy p do kolejki
3. Zwracamy jedyny element kolejki jako wynik

Dowód poprawności

Niech Opt będzie jakimś rozwiązaniem optymalnym.

Lemat 1

Pokażmy najpierw, że dla $i > 1$ Opt zawsze ma dwa wierzchołki w_a, w_b na najniższym poziomie.

Załóżmy nie wprost, że tak nie jest. W takim razie na najniższym poziomie Opt ma tylko liście, które nie współdzielą rodzica z żadnym innym liściem. Weźmy dowolny taki liść v. Jego odległość od korzenia w Opt to $d(v)$. Jeśli usuniemy jego rodzica i w jego miejsce wstawimy v to otrzymamy nowe drzewo Opt', w którym odległość $d'(v)$ jest mniejsza niż $d(v)$. Jako, że jest to jedyna zmiana wpływająca na wagę tego drzewa to $EL(Opt)$ < $EL(Opt')$ co jest sprzeczne z założeniem o optymalności Opt.

Lemat 2

Pokażmy teraz, że jeśli $w_i \le w_j$ są najmniejszymi wagami to istnieje takie rozwiązanie optymalne, że w_i, w_j są braćmi.

Wybierzmy braci na najniższym poziomie drzewa Opt (na mocy lematu 1 wiemy, że tacy istnieją). Nazwijmy ich w_a oraz w_b. Bez straty ogólności załóżmy, że $w_a \le w_b$.

Zauważmy, że po zamianie miejscami wierzchołków w_a <-> w_i oraz w_b <-> w_j otrzymaliśmy nowe drzewo Opt' takie, że:

- $d(w_i) \le d(w_a)$
- $d(w_j) \le d(w_b)$
- $w_i \le w_a$
- $w_j \le w_b$

Stąd wnioskujemy, że

- $d(w_i)*w_a + d(w_a)*w_i \le d(w_i)*w_i + d(w_a)*w_a$ bo $(d(w_i) - d(w_a))(w_a - w_i) \le 0$

- $d(w_j)*w_b + d(w_b)*w_j \le d(w_j)*w_j + d(w_b)*w_b$ bo $(d(w_j) - d(w_b))(w_b - w_j) \le 0$

- $[d(w_i)*w_a + d(w_j)*w_b + d(w_a)*w_i + d(w_b)*w_j] \le [d(w_i)*w_i + d(w_j)*w_j + d(w_a)*w_a + d(w_b)*w_b]$

Wiemy, że $EL(Opt') = EL(Opt) - [d(w_i)*w_i + d(w_j)*w_j + d(w_a)*w_a + d(w_b)*w_b] + [d(w_i)*w_a + d(w_j)*w_b + d(w_a)*w_i + d(w_b)*w_j]$

Czyli $EL(Opt') \le EL(Opt)$

Dowód

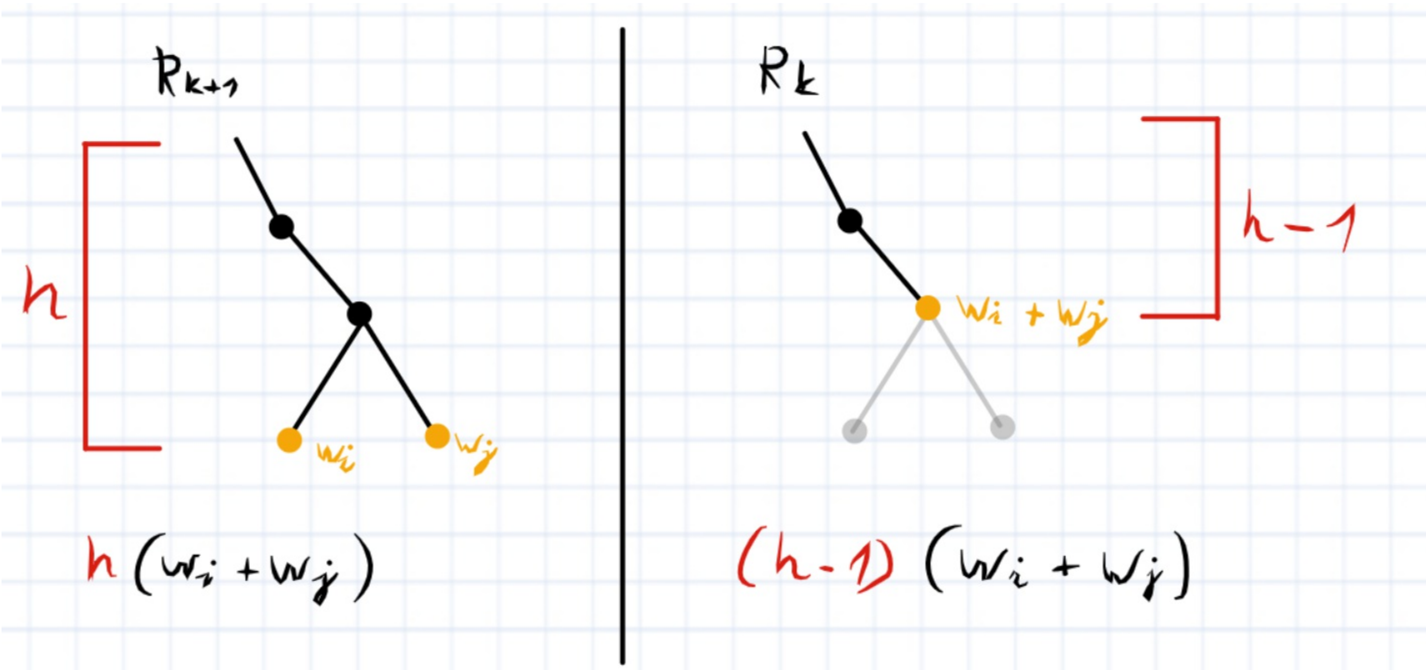
Pokażmy, że drzewo zwrócone przez algorytm jest optymalne.

Dowód ma charakter indukcyjny.

- ***Baza (n = 1 lub n = 2):*** Jest tylko jedno takie drzewo, więc algorytm zwraca drzewo optymalne.

- ***Krok (Zał: dla każdego $k \le n$ algorytm zwraca optymalne rozwiązanie):*** Niech w_i, w_j oznaczają najmniejsze wagi w kolejce K_{n+1}. Niech $K_n = K_{n+1} \setminus {w_i,w_j} \cup {w_i + w_j}$ Niech R_k, R_{k+1} będą rozwiązaniami algorytmu odpowiednio dla K_n, K_{n+1}.

  Zauważmy również, że $EL(R_{k+1}) = EL(R_k) + w_i + w_j$




```
 Korzystając z *lematu 2* weźmy rozwiązanie optymalne $Opt$ dla $K_n$
o wierzchołkach $w_i, w_j$ na najniższym poziomie mających wspólnego rodzica. Usuńmy teraz liście $w_i, w_j$ z $Opt$ i dodajmy do niego liść $w_i + w_j$. Nazwijmy to drzewo $T$.
```

Zauważmy, że T ma teraz n wierzchołków oraz $EL(Opt) = EL(T) + w_i + w_j$. Z założenia indukcyjnego wiemy, że R_k jest optymalne, więc $EL(T) \ge EL(R_k)$.

```
 $$EL(Opt) = EL(T) + w_i + w_j \ge  EL(R_k) + w_i + w_j = EL(R_{k+1})$$

Otrzymujemy $EL(Opt) \ge EL(R_{k+1})$,
ale skoro $Opt$ jest optymalny to $EL(Opt) = EL(R_{k+1})$
```