AiSD L3

Maurycy Borkowski

31.03.2021

zadanie 1

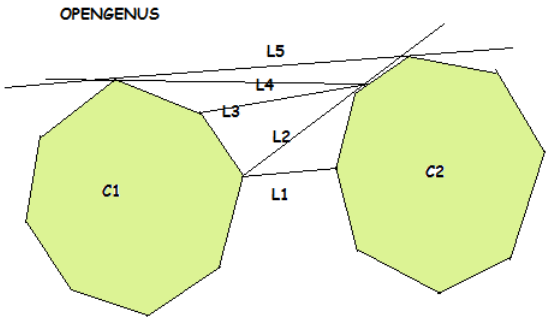
```
def gcd(a,b):
    if b > a:
        swap(a,b)
    if a == b:
        return a
    if min(a,b) == 1:
        return 1

    if a%2 == 0 and b%2 == 0:
        return 2*gcd(a/2,b/2)
    if a%2 == 1 and b%2 == 0:
        return gcd(a,b/2)
    if a%2 == 0 and b%2 == 1:
        return gcd(a/2,b)
    if a%b == 1 and b%2 == 1:
        return gcd((a-b)/2,b)
```

Poprawność algorytmu wynika z poprawności danej własności.

Zauważmy, że z każdym wywołaniem rekurencyjnym conajmniej jedna z liczb maleje dwukrotnie, zatem w pesymistycznym przypadku będziemy mieli $\mathcal{O}(\log a + \log b) = \mathcal{O}(\log ab)$ Tak samo jak alg. Euklidesa

zadanie 3

Będziemy chcieli połączyć dwie pary wierzchołków otoczek C_1, C_2 na *górze* i *dole* tak by powstała otoczka.
Możemy to zrobić łącząc najbliższe (skrajne wewnętrzne) punkty otoczek C_1, C_2 i później *wspinając* się z tym odcinkiem, dopóki nie będziemy mieli otoczki, tj. będą kąty wklęsłe.



DEMONSTRATION OF FINDING UPPER TANGENT OF TWO POLYGONS C1 AND C2

Otrzymujemy algorytm:

```
L = wewnetrzne(C1,C2)
while flag:
    flag = False
    while przecina(L, C1):
        przesun_wyzej(L, C1)
    while przecina(L, C2):
        flag = True
        przesun_wyzej(L, C2)
```

Funkcję $przecina(L, C)$ możemy zaimplementować korzystając z iloczynu wektorowego, dokładnie jego znaku, który określa nam skrętność, używając punktów określających L w obu otoczkach i ich następników.

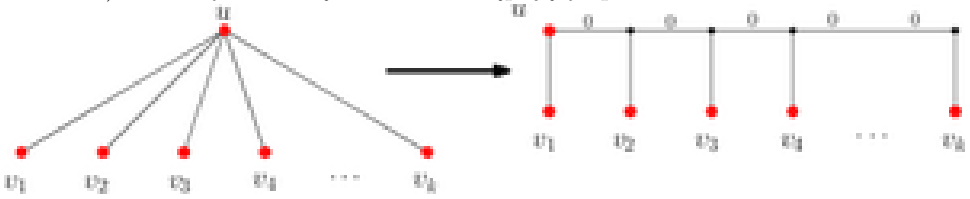
przesun_wyzej przesuwa punkt wyżej w danej otoczce: $v = C[(idx + 1)\%n]$
Analogicznie znajdujemy odcinek na dole otoczek.

Poprawność

Otrzymujemy wielokąt wypukły, odcinki łączące C_1, C_2 nie przecinają ich (dopiero wtedy kończymy pętlę).
C_1, C_2 zawierają wszystkie punkty, więc zbiór w którym obie się one zawierają też zawiera wszystkie punkty.

zadanie 4a

Na początku zmodyfikujmy (liniowo) nasze drzewo tak aby każdy wierzchołek v miał $deg(v) \leqslant 3$. Zrobimy to dodając czarne wierzchołki (czerwone = oryginalne z drzewa) z zerowymi krawędziami w następujący sposób:



każdego nadmiarowego syna podpinamy do nowego czarnego którego łączymy z poprzednim czarnym wierzchołkiem.
Zerowe krawędzie gwarantują nam, że w tak zmodyfikowanym drzewie rozwiązanie się nie zmieni.

Korzystamy z metody *Dziel i Zwyciężaj* dla danego wierzchołka u i jego conajwyżej trzech poddrzew T_1, T_2, T_3 o korzeniach (sąsiadach u) r_1, r_2, r_3, odpowiedź to ścieżki wewnątrz każdego z poddrzew T_i, ścieżki pomiędzy poddrzewami T_i, T_j (i ścieżki pomiędzy T_i a u).
Dla danych i, j $(i < j)$ oznaczmy przez $w = d(u, r_i) + d(u, r_j)$, $A = v, v \in T_i$, $B = w, w \in T_j$. W A, B trzymamy tylko czerwone wierzchołki.
A, B sortujemy po odległościach wierzchołków od korzeni np. A po $d(v, r_i)$.

Teraz szukamy takich par $x, y, x \in A, y \in B$, że $d(x, r_i) + d(y, r_j) = D - w$
Możemy to zrobić w czasie liniowym (po wcześniejszym sortowaniu A, B) idąc od lewej w A i od prawej w B.
Oszacujmy, złożoność takiego algorytmu:

$$T(n) = \mathcal{O}(n \log n) + T(n_1) + T(n_3) + T(n_3)$$

Aby złożoność nam nie wybuchła musimy jeszcze zadbać o to by rozmiary drzew wywołania rekurencyjnych były ograniczone lepiej niż n.
Bo wiemy, że $n_1 + n_2 + n_3 = n - 1$ w przeciwnym przypadku być może wykonamy n wywołań otrzymując złożoność $\mathcal{O}(n^2 \log n)$.
Możemy to osiągnąć biorąc jako wierzchołek do usunięcia u centroid obecnego drzewa, wtedy $n_i \leqslant \frac{n}{2}$, centroid można wyszukać w czasie $\mathcal{O}(n)$.
Każde wywołanie zmniejsza nam drzewo conajmniej dwukrotnie, więc pełne drzewo będzie

```
def getAns(l1, l2):
    sort(l1)

def solve(G):
    u = centroid(G)
    ans = 0
    red = []
    for r in G[u]:
        G2 = subtree(r)
        red.append(getRed(G2))
        ans += solve(G2)
    for (i,j) in list(combinations(range(3), 2)):
        ans += getAns(red[i],red[j])

dodaj_czarne(G)
```

Zad. 5

Algorytm

$M = \max(x_1, \dots, x_n)$

SWAP(M, x_1)
TMP = M
FOR $i = 2$ TO N:
    TMP = GCD(M, x_i)
RETURN TMP

Złożoność:

Lemat 1: $a \bmod b < \frac{a}{2}$ gdy $a > b$

$a = b \cdot k + r$

$r < b \cdot k \iff 2r < bk + r \iff$

$\iff r < \frac{a}{2}$

$\to a \bmod b$ ∎

*GCD(a, b) → GCD(b, a mod b) → GCD(a mod b, b mod (a mod b))

$\frac{a}{2}$ ↑ $\frac{b}{2}$

- Więc w algorytmie Euklidesa w 2 iteracjach dwa operandy zmniejszają się conajmniej dwukrotnie.

- Idąc w pętli FOR "ciągniemy" ze sobą TMP więc wykonamy conajwyżej log M niepotrzebnych GCD.

- Oznaczmy $k_1, \dots, k_z$ ilości operacji w kolejnych GCD. ($z < \log M$)

$$\left\lfloor \frac{k_1}{2} \right\rfloor + \left\lfloor \frac{k_2}{2} \right\rfloor + \dots + \left\lfloor \frac{k_z}{2} \right\rfloor \leq \log M$$

Kolejne* dwie iteracje zmniejszają dwa operandy conajmniej dwukrotnie

$$\Downarrow$$

$$k_1 + \dots + k_z \leq 2 \log M + z^{**} \leq 3 \log M$$

Stąd wynika $O(n + \log M)$ operacji

Zadanie 7.

a)

Zauważmy, że tak zadana macierz ma tylko $2n-1$ różnych elementów i wszystkie zanjdują się w pierwszej kolumnie i pierwszym wierszu (tam zaczyna się każda przekątna). Możemy więc macierz reprezentować jako wektor o rozmiarze $2n-1$ dodawanie dwóch macierzy możemy realizować przez dodawanie wektorów (przekątnę sie dodają). Stąd możemy je dodawać w $\mathcal{O}(n)$.

b)

Kluczowa obserwacja! Jeżeli $2 \mid n$ to macierz Teoplitz'a A możemy zapiać:

$$A = \begin{pmatrix} \mathbf{M} & N \\ P & \mathbf{M} \end{pmatrix}$$

Gdzie każda z macierzy M, N, P jest rozmiaru $\frac{n}{2} \times \frac{n}{2}$.
Teraz gdy chcemy ją pomnożyć przez wektor $v = \begin{bmatrix} u \\ w \end{bmatrix}$ gdzie u, w to odpowowiednio pierwsze i drugie $\frac{n}{2}$ elementów wektora v. Dalej:

$$Av = \begin{pmatrix} \mathbf{M} & N \\ P & \mathbf{M} \end{pmatrix} \cdot \begin{bmatrix} u \\ w \end{bmatrix} = \begin{pmatrix} Mu + Nw \\ Pu + Mw \end{pmatrix} = \begin{pmatrix} Mu + Nw + Mw - Mw \\ Pu + Mw + Mu - Mu \end{pmatrix}$$
$$= \begin{pmatrix} \mathbf{M(u+w)} + (N - M)w \\ \mathbf{M(w+u)} + (P - M)u \end{pmatrix}$$

Widzimy więc, że możemy uzyskać wynik wykonując 3 mnożenia (macierz·wektor) i 4 dodawania wektorów (N, P, M możemy reprezentować jako wektor z **a**)

W przypadku gdy $2 \nmid n$ musimy trochę pokombinować, żeby znaleźć *te same* fragmenty macierzy A.
Okazuje się, że wystarczy podzielić macierz na środkową kolumną i środkowym wierszem:

$$A = \begin{pmatrix} \mathbf{M} & d_1 & N \\ & d_2^T & \\ P & d_3 & \mathbf{M} \end{pmatrix}$$

Wtedy mnożenie ma postać:

$$Av = \begin{pmatrix} \mathbf{M} & d_1 & N \\ & d_2^T & \\ P & d_3 & \mathbf{M} \end{pmatrix} \cdot \begin{bmatrix} u \\ x \\ w \end{bmatrix} = \begin{pmatrix} Mu + d_1 x + Nw \\ d_2^T(u + x + w) \\ Pu + d_3 x + Mw \end{pmatrix} =$$
$$\begin{pmatrix} Mu + d_1 x + Nw + Mw - Mu \\ d_2^T v \\ Pu + d_3 x + Mw + Mu - Mu \end{pmatrix} = \begin{pmatrix} \mathbf{M(u+w)} + d_1 x + (N - M)w \\ d_2^T v \\ u(P - M) + d_3 x + \mathbf{M(w+u)} \end{pmatrix}$$

W tym przypadku wykonujemy 3 mnożenia (macierz·wektor), 3 dodawania wektorów i jeden iloczyn skalarny.

Stąd:

$$T(n) = \begin{cases} 1 & \text{gdy } n = 1 \\ \frac{n}{2} + 2 \cdot (2 \cdot \frac{n}{2} - 1) + 2 \cdot \frac{n}{2} + 3T(\frac{n}{2}) = O(n) + 3T(\frac{n}{2}) & \text{gdy } 2 \mid n \\ \frac{n-1}{2} + 2 \cdot (2 \cdot \frac{n-1}{2} - 1) + 4 \cdot (\frac{n-1}{2}) + 3T(\frac{n-1}{2}) = O(n) + 3T(\frac{n-1}{2}) & \text{gdy } 2 \nmid n \end{cases}$$

Stąd:

$$T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$$

Zadanie. 8

Możemy założyć, że w A wszytskie liczby są różne. Możemy patrzeć tak przy porównywaniu: $T_{i,j}, T_{m,n}$:

$$T_{i,j} < T_{m,n} \to i < m \to j < n$$

***** Zauważmy, że warunkiem wystarczającym na bycie medianą, jest bycię liczbą, która jest większa od $k \cdot \frac{n}{2}$ pozostałych elementów. (w nowym porządku)

Zakładamy, że rozwiązanie znajduje się na i-tej liście, możemy szukać konkretnego rozwiązania na tej liście używając binsearch'a.
Liczba elementów mniejszych od $T_{i,k}$ jest większa od liczby elementów mniejszych od $T_{i,j}$ dla $k > j$. (Liczba elementów mniejszych zwiększa się z indeksem w każdej z tablic)

Teraz mając konkretnego kandydata na medianę, chcemy sprawdzić czy jest on nią, tj. czy spełnia *****, policzyć od elementów . Możemy to też zrobić binsearch'em, w każdej z k list szukając od ilu elementów nasz kandydat jest większy. Jeżeli za mało przesuwamy się w *prawo* a jak za dużo to w *lewo*.

Sprawdzamy czy jest kandydat w każdej z k list, szukanie w niej kandydata trwa $\log n$ sprawdzenie czy dana liczba jest medianą trwa $k \cdot \log n$ (w każdej tablicy binsearch). Stąd $\mathcal{O}(k^2 \log^2 n)$

```
# zakladamy juz nowy porzadek
Ts = [] # tablica tablic
n = len(Ts[0])

def getSmaller(elm, T):
    l = 0
    r = n-1
    while l < r:
        mid = (l+r)//2
        if T[mid] < elm:
            l = mid
        else:
            r = mid - 1
    return l

def checkCandidate(elm):
    smaller = 0
    for T in Ts:
        smaller+=getSmaller(elm, T)
    if smaller < n//2:
        return -1
    if smaller > n//2:
        return 1
    return 0


for i in range(k):
    l = 0
    r = n-1
    while l < r:
        mid = (l+r)//2
        if checkCandidate(Ts[mid]) == 1:
            r = mid - 1
        if checkCandidate(Ts[mid]) == -1:
            l = mid + 1
        if checkCandidate(Ts[mid]) == 0:
            return mid
```