Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Dokumentacja projektu

Algorytmy geometryczne

K. Kafara
Ł. Czarniecki

Spis treści

Spis rysunków

Spis tablic

1 Informacje techniczne

1.1 Budowa programu

Program złożony jest z następujących modułów:

- *lib* – biblioteczny – zawiera zbiór pomocniczych funkcji i struktur danych wykorzystywanych przez algorytmy.
- *pure* – algorytmy w *czystej postaci* tj. nie posiadające części wizualizacyjnej.
- *vis* – algorytmy wraz z kodem odpowiadającym za wizualizację

Poniżej przedstawiamy dokładny opis zawartości poszczególnych modułów.

1.1.1 Moduł *lib*

Moduł zawiera w sobie następujące podmoduły:

1. *geometric_tool_lab.py* – narzędzie graficzne dostarczone w ramach przedmiotu *Algorytmy geometryczne*
2. *getrand.py* – zawiera funkcje generujące zbiory punktów różnych typów
3. *sorting.py* – zawiera implementację iteracyjnej wersji algorytmu *QuickSort* wykorzystywaną m.in w algorytmie Grahama
4. *stack.py* – zawiera klasę implementującą *stos*
5. *util.py* – zawiera szereg funkcji pomocniczych wykorzystywanych przez zaimplementowane algorytmy
6. *mytypes.py* – zawiera definicje typów stworzone w celu zwiększenia czytelności kodu

1.1.2 Moduł *pure*

Moduł zawiera w sobie następujące podmoduły:

1. *divide_conq.py* – implementacja algorytmu dziel i zwyciężaj
2. *graham.py* – implementacja algorytmu Grahama
3. *increase.py* – implementacja algorytmu przyrostowego
4. *jarvis.py* – implementacja algorytmu Jarvisa
5. *lowerupper.py* – implementacja algorytmu "górna-dolna"

1.1.3 Moduł *vis*

Moduł zawiera w sobie następujące podmoduły:

1. *divide_conq_vis.py* – implementacja algorytmu dziel i zwyciężaj wraz z kodem tworzącym wizualizację
2. *graham_vis.py* – implementacja algorytmu Grahama wraz z kodem tworzącym wizualizację
3. *increase_vis.py* – implementacja algorytmu przyrostowego wraz z kodem tworzącym wizualizację
4. *jarvis_vis.py* – implementacja algorytmu Jarvisa wraz z kodem tworzącym wizualizację
5. *lowerupper_vis.py* – implementacja algorytmu "górna-dolna" wraz z kodem tworzącym wizualizację

1.2 Wymagania techniczne

1. Python 3.9.0 64-bit lub nowszy
2. Jupyter Notebook

1.3 Korzystanie z programu

1.3.1 Uruchomienie wizualizacji

W celu uruchomienia wizualizacji algorytmów należy uruchomić notebook (poprzez Jupyter Notebook) *program.ipynb*, a następnie zapoznać się z zamieszczoną tam instrukcją.

2 Oznaczenia i definicje

Na potrzeby dalszych wywodów przyjmujemy w tym miejscu szereg oznaczeń i definicji:

3 Problem

Wyznaczyć otoczkę wypukłą podanego zbioru punktów płaszczyzny dwuwymiarowej.

4 Algorytmy

4.1 Algorytm Grahama

W celu opisania sposobu działania algorytmu Grahama, definiujemy następującą relację $\preceq_Q$ określoną dla dowolnych dwóch punktów płaszczyzny P_1, P_2 względem wybranego i ustalonego punktu odniesienia Q.

$P_1 \preceq_Q P_2 \Leftrightarrow (\angle(P_1, Q, OX) < \angle(P_2, Q, OX)) \vee (\angle(P_1, Q, OX) = \angle(P_2, Q, OX) \wedge d(P_1, Q) \leqslant d(P_2, Q))$

gdzie $d(P, Q)$ oznacza odległość od siebie dwóch dowolnych punktów płaszczyzny.

Tak zdefiniowana relacja jest liniowym porządkiem (zwrotna, antysymetryczna, przechodnia i spójna).

4.1.1 Opis działania

1. Wyznaczamy najniższy punkt Q wyjściowego zbioru (jeżeli jest wiele o tej samej rzędnej – bierzemy ten o najmniejszej odciętej).
2. Ustawiamy go jako pierwszy element zbioru.
3. Sortujemy pozostałe punkty względem relacji $\preceq_Q$.
4. Usuwamy wszystkie, poza najbardziej oddalonym od Q, punkty leżące na półprostej QP, dla każdego P
5. Kładziemy pierwsze 3 punkty zbioru na stos S.
6. Iterujemy kolejno po punktach z posortowanego zbioru nie będących na stosie: Niech bieżącym punktem będzie P:
   (a) Dopóki P nie jest po lewej stronie $S_{n-1}S_n$ wykonujemy (b)
   (b) Uswamy punkt ze stosu.
   (c) Dodajemy P na stos.
7. Zwracamy zawartość stosu.

4.1.2 Szczegóły

- Najniższy punkt wyjściowego zbioru (punkt 1) wyznaczamy w czasie liniowym, iterując po kolejnych punktach zbioru.
- Wszystkie punkty leżacej na jednej prostej, poza najbardziej oddalonym od Q usuwamy w czasie liniowym w następujący sposób: Iterując przez posortowaną tablicę, zaczynająć od indeksu $i := 1$, zapamiętujemy ostatni indeks na który wstawialiśmy j (na początku $j := 1$). Jeżeli Q, P_i, P_{i+1} są współliniowe to $i := i+1$. Jeżeli nie są współliniowe to P_i wpisujemy na pozycję j, a następnie $j := j + 1$. Następnie, w dalszej części algorytmu posługujemy się częścią tablicy $[0, \ldots, j-1]$.

4.1.3 Złożoność

Operacją dominującą w algorytmie jest sortowanie – realizowane w czasie $O(n\,lgn)$. Wybór punktu najniższego, redukcja punktów współlinowych oraz iterowanie (punkt 6, zauważmy, że każdy punkt zbioru wyjściowego jest obsługiwany co najwyżej 2 razy – gdy jest dodawany do otoczki i gdy jest ewentualnie usuwany) są realizowane w czasie $O(n)$. Algorytm Grahama ma zatem złożoność $O(n\,lgn)$.

4.1.4 Kod

```
def get_point_cmp(ref_point: Point, eps: float = 1e-7) -> Callable:
    def point_cmp(point1, point2):
        orient = orientation(ref_point, point1, point2, eps)

        if orient == -1:
            return False
        elif orient == 1:
            return True
        elif dist_sq(ref_point, point1) <= dist_sq(ref_point, point2):
            return True
        else:
            return False

    return point_cmp


def graham(points: ListOfPoints) -> ListOfPoints:
    istart = index_of_min(points, 1)

    points[istart], points[0] = points[0], points[istart]

    qsort_iterative(points, get_point_cmp(points[0]))

    i, new_size = 1, 1
    while i < len(points):
        while (i < len(points) - 1) \
        and \
        (orientation(points[0], points[i], points[i + 1], 1e-7) == 0):
            i += 1

        points[new_size] = points[i]
        new_size += 1
        i += 1

    s = Stack()
    s.push(points[0])
    s.push(points[1])
    s.push(points[2])

    for i in range(3, new_size, 1):
        while orientation(s.sec(), s.top(), points[i], 1e-7) != 1:
            s.pop()

        s.push(points[i])

    return s.s[:s.itop+1]

```

4.2 Algorytm Jarvisa

4.2.1 Opis działania

1. Wyznaczamy najniższy punkt Q wyjściowego zbioru (jeżeli jest wiele o tej samej rzędnej – bierzemy ten o najmniejszej odciętej).
2. Dodajemy Q do zbioru punktów otoczki.
3. Przeglądamy punkty zbioru w poszukiwaniu takiego, który wraz z ostatnim punktem otoczki tworzy najmniejszy kąt skierowany względem ostatniej znanej krawędzi otoczki. Dla pierwszego szukanego punktu, kąt namierzamy względem poziomu.
4. Znaleziony punkt dodajemy do zbioru punktów otoczki, jeżeli jest różny od Q.
5. Powtarzamy punkty 3 i 4 tak długo aż znalezionym punktem nie będzie Q.
6. Zwracamy listę punktów otoczki.

4.2.2 Szczegóły

- Najniższy punkt wyjściowego zbioru (punkt 1) wyznaczamy w czasie liniowym, iterując po kolejnych punktach zbioru.
- W celu wyznaczenia punktu wyspecyfikowanego w punkcie 3. nie obliczamy wartości odpowiedniego kąta. Zamiast tego, równoważnie, wyznaczamy punkt P, który wraz z ostatnim znanym punktem otoczki P_0 tworzy wektor $\vec{P_0P}$ dla którego wszystkie pozostałe punkty zbioru są po lewej stronie. Robimy to w czasie liniowym korzystając z znanych własności wyznacznika.

4.2.3 Złożoność

Zauważmy, że jeżeli otoczka jest k - elementowa, to główna pętla algorytmu (punkty 3–4) wykonuje się k-razy. Każdy krok pętli (znalezienie odpowiedniego punktu P) zajmuje czas liniowy. Pozostałe operacj w algorytmie zajmują co najwyżej czas liniowy. Zatem algrytm Jarvisa ma złożoność $O(nk)$.

4.2.4 Kod

```
def jarvis(points: ListOfPoints) -> ListOfPoints:
EPS = 1e-8

convex_hull = []

start_idx = index_of_min(points, 1)

convex_hull.append(start_idx)

rand_idx = 0 if start_idx != 0 else 1

```

```
prev = start_idx

while True:
    imax = rand_idx

    for i in range(len(points)):
        if i != prev and i != imax:
            orient = orientation(
                        points[prev],
                        points[imax],
                        points[i],
                        EPS
                     )
            if orient == -1:
                imax = i

            elif orient == 0 and \
                 (dist_sq(points[prev], points[imax]) < dist_sq(points[prev], points[i])):
                imax = i

    if imax == start_idx:
        break;

    convex_hull.append(imax)

    prev = imax

return points[convex_hull]

```

W ostatniej linii algorytmu, korzystamy z możliwości bibliteki *numpy*.

4.3 Algorytm górna-dolna

4.3.1 Opis działania

1. Sortujemy punkty rosnąco po odciętych (w przypadku rówych, mniejszy jest punkt o mniejszej rzędnej).
2. Pierwsze dwa punkty z posortowanego zbioru wpisujemy do zbioru punktów otoczki górnej oraz dolnej.
3. Iterujemy po zbiorze punktów zaczynając od $i = 2$ (trzeciego punktu), niech P będzie bieżącym punktem:
   (a) Dopóki górna (dolna) otoczka ma co najmniej 2 punkty i P nie znajduje się po prawej (lewej) stronie odcinka skierowanego utworzonego przez ostatniej dwa punkty otoczki (ostatni jest końcem odcinka), wykonujemy (b):
   (b) Usuwamy ostatni punkt z otoczki górej (dolnej).
   (c) Dodajemy P do punktów otoczki górnej (dolnej).

4. Odwracamy kolejność wierzchołków w otoczce dolnej.
5. Łączymy zbioru punktów otoczki górnej oraz dolnej.
6. Zwracamy złączony zbiór punktów otoczki.

4.3.2 Złożoność

Dominującą operacją w algorytmie jest sortowanie realizowane w czasie $O(n\,lgn)$. Każdy krok pętli (dla wyznaczania otoczki górnej oraz dolnej) zajmuje czas stały. Zauważmy, że podobnie do algorytmu Grahama każdy z punktów jest rozważany co najwyżej dwukrotnie – w momencie dodania do otoczki i przy ewentualnym usunięciu ze zbioru punktów otoczki. Pozostałe operacje realizowane są w czasie liniowym. Zatem algorytm "górna-dolna"ma złożoność $O(n\,lgn)$.

4.3.3 Kod

```python
def lower_upper(point2_set: ListOfPoints) -> ListOfPoints:
if len(point2_set) < 3: return None

point2_set.sort(key = operator.itemgetter(0, 1))

upper_ch = [ point2_set[0], point2_set[1] ]
lower_ch = [ point2_set[0], point2_set[1] ]

for i in range( 2, len(point2_set) ):
    while len(upper_ch) > 1 and orientation(upper_ch[-2], upper_ch[-1], point2_set[i]) != -1:
        upper_ch.pop()

    upper_ch.append(point2_set[i])

for i in range(2, len(point2_set) ):
    while len(lower_ch) > 1 and orientation(lower_ch[-2], lower_ch[-1], point2_set[i]) != 1:
        lower_ch.pop()

    lower_ch.append(point2_set[i])

lower_ch.reverse()
upper_ch.extend(lower_ch)

return upper_ch
```

4.4 Algorytm przyrostowy

4.4.1 Opis działania

Ogólne sformułowanie algorytmu ma postać:

1. Dodajemy pierwsze 3 punkty do zbioru punktów otoczki.

2. Iterujemy po pozostałych punktach. Niech P będzie punktem bieżącym:
   (a) Jeżeli P nie należy do wnętrza obecnie znanej otoczki wykonumejmy (b) oraz (c).
   (b) Znajdujemy styczne do obecnie znanej otoczki poprowadzone przez punkt P.
   (c) Aktualizujemy otoczkę.
3. Zwracamy punkty otoczki.

Możemy go jednak sformułować inaczej, co pozwoli na uproszenie implementacji, przy zachowaniu takiego samego rzędu złożoności.

1. Sortujemy punkty rosnąco po odciętych (w przypadku rówych, mniejszy jest punkt o mniejszej rzędnej).
2. Dodajemy pierwsze 3 punkty do zbioru punktów otoczki, w takiej kolejności, aby były podane w kolejności odwrotnej do ruchu wskazówek zegara.
3. Iterujemy po pozostałych punktach. Niech P będzie punktem bieżącym:
   (a) Znajdujemy styczne do obecnie znanej otoczki poprowadzone przez punkt P.
   (b) Aktualizujemy otoczkę.
4. Zwracamy punkty otoczki.

Dzięki wstępnemu posortowaniu puntków, omijamy konieczność testowania należenia P do otoczki znanej w danym kroku algorytmu, ponieważ biorąc kolejny punkt mamy gwarancję, że nie należy on do wcześniej znanej otoczki.

4.4.2 Szczegóły

Wyznaczanie stycznych

Styczne wyznaczamy w czasie logartymicznym względem liczby punktów należących do otoczki do której szukamy stycznych, wykonując poszukiwanie binarne elementów skrajnych (najmniejszego i największego) względem następującego porządku, określonego dla dowolnych dwóch punktów płaszczyzny P_1, P_2, względem ustalonego punktu Q:
$P_1 >_Q P_2 \Leftrightarrow P_2$ znajduje się po lewej stronie odcinka skierowanego QP_1.

W tak określonym porządku elementem największym będzie prawy punkt styczności P_{max}, ponieważ wszystkie inne punkty otoczki znajdują się na lewo od prostej (stycznej) QP_{max}. Podobnie lewym punktem styczności będzie element najmniejszy w zadanym porządku P_{min}.

Wyznaczanie P_{max} (P_{min} wyznaczamy w sposób zupełnie analogiczny):
Wprowadźmy najpierw potrzebne oznaczenia:
Q punkt zewnętrzny, przez który mają przechodzić styczne do otoczki.
Niech otoczka będzie dana w postaci ciągu punktów $P_0, \ldots, P_{k-1}$ ($P_k := P_0$) będących

współrzędnymi kolejnych wierzchołków w kolejności przeciwnej do ruchu wskazówek zegara.
Krawędź skierowaną otoczki e_i definiujemy następująco: $e_i := P_i P_{i+1}$.
Krawędź $e_i := P_i P_{i+1}$ jest skierowana w górę $\Leftrightarrow P_{i+1} >_Q P_i$.
Krawędź e_i jest skierowana w dół, jeżeli nie jest skierowana w górę.

1. Jeżeli P_0 jest większy od swoich obudwu sąsiadów, to go zwracamy jako element największy.

2. Definujemy indeksy $l := 0$, $r := k$

3. Dopóki nie znajdziemy elementu największego:

   (a) Wyznaczamy indeks środkowego elementu $m := \left\lfloor \frac{l+k}{2} \right\rfloor$

   (b) Jezeli P_m jest elementem największym:
      - Zwracamy P_m.

   (c) Jeżeli e_l jest skierowana w górę:

      i. Jeżeli e_m jest skierowana w dół
         - $r := m$

      ii. W przeciwnym przypadku

         A. Jeżeli $P_l >_Q P_m$:
            - $r := m$

         B. W przeciwnym przypadku
            - $l := m$

   (d) W przeciwnym przypadku:

      i. Jeżeli e_m jest skierowana w górę:
         - $l := m$

      ii. W przeciwnym przypadku:

         A. Jeżeli $P_m >_Q P_l$:
            - $r := m$

         B. W przeciwnym przypadku:
            - $l := m$

4.4.3 Złożoność

Posortowanie punktów zajmuje $O(n\,lgn)$. Każde wykonanie pętli zajmuje czas logarytmiczny względem liczebności otoczki znanej w danej iteracji. Zatem złożoność algorytmu jest rzędu $O(n\,lgn)$

4.4.4 Kod

```python
def right_tangent(polygon: ListOfPoints, point: Point) -> Union[int, None]:
    n = len(polygon)

    if n < 3: return None

    if orientation(polygon[0], polygon[1], point) == 1 and orientation(polygon[-1], polygon[0], point) == -1:
        return 0

    left = 0
    right = n

    while True:
        mid = (left + right) // 2

        is_mid_down: bool = (orientation(polygon[mid], polygon[(mid + 1) % n], point) == 1)

        if is_mid_down and orientation(polygon[mid - 1], polygon[mid], point) == -1:
            return mid % n

        is_left_up: bool = (orientation(polygon[left % n], polygon[(left + 1) % n], point) == -1)

        if is_left_up:
            if is_mid_down:
                right = mid
            else:
                if orientation(point, polygon[left], polygon[mid]) == 1:
                    right = mid
                else:
                    left = mid
        else:
            if not is_mid_down:
                left = mid
            else:
                if orientation(point, polygon[mid % n], polygon[left % n]) == 1:
                    right = mid
                else:
                    left = mid


def left_tangent(polygon: ListOfPoints, point: Point) -> Union[Point, None]:
    n = len(polygon)

    if n < 3: return None

    left = 0
```

```
    right = n

    if orientation(point, polygon[1], polygon[0]) == 1 and orientation(point, polygon[-1], polygon[0]) == 1:
        return 0

    while True:
        mid = (left + right) // 2

        is_mid_down: bool = (orientation(point, polygon[mid % n], polygon[(mid + 1) % n]) == 1)

        if (not is_mid_down) and orientation(point, polygon[(mid - 1) % n], polygon[mid % n]) == 1:
            return mid % n

        is_left_down: bool = (orientation(point, polygon[left], polygon[(left + 1) % n]) == 1)

        if is_left_down:
            if not is_mid_down:
                right = mid
            else:
                if orientation(point, polygon[mid % n], polygon[left % n]) == 1:
                    right = mid
                else:
                    left = mid
        else:
            if is_mid_down:
                left = mid
            else:
                if orientation(point, polygon[left % n], polygon[mid % n]) == 1:
                    right = mid
                else:
                    left = mid


def increase_with_sorting(point2_set: ListOfPoints) -> Union[ListOfPoints, None]:
    if len( point2_set ) < 3: return None

    point2_set.sort(key = operator.itemgetter(0, 1))

    convex_hull = point2_set[:3]

    if orientation(convex_hull[0], convex_hull[1], convex_hull[2]) == -1:
        convex_hull[1], convex_hull[2] = convex_hull[2], convex_hull[1]

    for i in range(3, len( point2_set )):
        left_tangent_idx = left_tangent(convex_hull, point2_set[i])
        right_tangent_idx = right_tangent(convex_hull, point2_set[i])

```

```python
        left_tangent_point = convex_hull[left_tangent_idx]
        right_tangent_point = convex_hull[right_tangent_idx]

        deletion_side: Literal[-1, 1] = orientation(left_tangent_point, right_tangent_point, point2_set[i])

        if orientation(left_tangent_point, right_tangent_point, convex_hull[(left_tangent_idx + 1) % len(convex_hull)]) == deletion_side:
            step = 0
        else:
            step = -1

        left = (left_tangent_idx + 1) % len(convex_hull)

        while convex_hull[left] != right_tangent_point:
            convex_hull.pop(left)
            left = (left + step) % len(convex_hull)

        convex_hull.insert(left, point2_set[i])

    return convex_hull
```

4.5 Algorytm dziel i zwyciężaj

4.5.1 Opis działania

Prócz zbioru punktów, dodatkową daną wejściową dla algorytmu jest stała k oznaczająca liczebość zbioru punktów, przy której przechodzimy w algorytmie rekurencyjnym do przypadku bazowego – wyznaczamy otoczkę innym, wybranym algorytmem.

Opisany algorytm jest algorytmem rekurencyjnym. Przed pierwszym wywołaniem rekurencyjnym należy zbiór punktów posortować rosnąco po odciętych (w przypadku równych, mniejszy jest punkty o mniejszej rzędnej).

Jest to standardowe zastosowanie metody "*dziel i zwyciężaj*":

1. Dzielimy wyjściowy problem na mniejsze tak długo, aż znajdujemy się w przypadku któryi potrafimy rozwiązać elementarnie / w inny sposób.

2. Łączymy kolejne rozwiązania częściowe w całość.

Popatrzmy na schemat działania:

1. Jeżeli liczebność rozważanego zbioru jest mniejsza bądź równa danej stałej k, to:

    (a) Wyznaczamy otoczkę rozważanego zbioru punktów, za pomocą innej metody (np. innego algorytmu wyznaczania otoczki).

    (b) Zwracamy tak uzyskaną otoczkę.

2. W przeciwnym przypadku:

    (a) Wywołujemy się rekurencyjnie na zbiorze punktów o odciętych mniejszych od mediany.

(b) Wywołujemy się rekurencyjnie na zbiorze punktóœ o odciętych większych bądź równych medianie.

(c) Łączymy lewą i prawą otoczkę (pozyskane z wywołań rekurencyjnych) w jedną.

(d) Zwracamy tak uzyskaną otoczkę.

4.5.2 Szczegóły

- Do wyznaczania otoczki w przypadku podstawowym wykorzystany został algortym Jarvisa, ponieważ dla $k << n$ ma on złożoność właściwie liniową.
- Sposób łączenia otoczek jest następujący:
  1. Wyznaczamy skrajny prawy punkt L lewej otoczki oraz skrajny lewy P punkt prawej otoczki.
  2. Dopóki poprzednik P

4.5.3 Złożoność

4.5.4 Kod

4.6 Algorytm Chana

4.6.1 Opis działania

Główna część algorytmu Chana składa się się z dwóch części:

1. Pierwsza, która składa się na :
   - Podział zbioru punktów Q na podzbiory Q_i o w miarę równych ilościach punktów w nich zawartych, z czego żaden nie zawiera więcej niż dane m.
   - Wyznaczenie otoczek C_i
2. Druga polega na wykonaniu algorytmu na wzór Jarvisa, tylko na otoczkach. Dokładniej mówiąc:
   - Startujemy z najniższy wierzchołkeim z całego zbioru Q i dodajemy go do finalnej otoczki jako pierwszy wierzchołek.
   - Dla każdego punktu należącego do otoczki, możemy znaleźć jego następnego sąsiada w otoczce idąc w kolejności przeciwnej do ruchu wskazóek zegara. Aby to zrobić należy wybrać spośród zbioru punktów utworzonego z: punktów tworzących prawą styczną z otoczkami C_i dla rozważanego wierzchołka, kolejnego punktu podotoczki do której dany punkt należy taki wierzchołek, że wszystkie inne wierzchołki z tego zbioru są na lewo od niego.
   - W ten sposób wyznaczamy kolejne wierzchołki otoczki, dopóki następnym wierzchołkiem otoczki nie jest jej pierwszy punkt. Wtedy otoczka jest pełna i kończymy algorytm.

3. Zwracamy punkty otoczki.

Jednakże nadżędną istotą powyższego algorytmu jest to, że wykona się on w drugiej części w co najwyżej m krokach(dany rozmiar podzbioru). Inaczej mówiąc m musi być większe bądź równe(w idealnym przypadku) liczbie punktów należących do otoczki k. Jeśli wykonamy m kroków i wciąż nie mamy otoczki, to przerywamy algorytm. I próbujemy z większym m. Aby nie popsóć złożoności dużą ilością powtórzeń głównej części algorytmu najlepiej za każdym razem parametr m podnosić do kwadratu. W przypadku gdy m¿=n po prustu za m przyjmujemy n. Wtedy algorytm chana sprowadza się do algorytmu Grahama.

4.6.2 Szczegóły

4.6.3 Złożoność

Złożoność głównej części algorytmu.

- Złożoność pierwsszej części składa się na :
  - Podział zbioru punktów na podzbiory O(n);
  - Wyznaczenie otoczek dla podzbiorów. Mamy ceil(n/m) podzbiorów rozmiaru m, dla każdego z nich wyznaczamy otoczkę algorytmem Grahama. Algorytm Grahama działa O(nlog(n)). Więc łącznie mamy O(ceil(n/m)*mlog(m))= O(nlog(m)).

  Łącznie dla pierewszej części mamy O(nlog(m)), gdzie m jest wybranym maksymalnym rozmiarem podzbiorów.

- Złożoność drugiej części składa się na :
  - Wyznaczenie następnego punktu dla każdego punktu z otoczki głównej o rozmiarze k
  - Wyznacznie następnego punktu składa się na wyznaczenie dla każdej z m podotoczek stycznej do tej podotoczki. Styczną wyznaczamy binary searchem w czasie $O(log(m))$ (otoczka C_i ma co najwyżej m wierzchołków). Otoczek jest $\lceil(n/m)\rceil$, a zatem czas wyznaczenia kolejngo wierzchołka otoczki to $O(\lceil(n/m)\rceil log(m))$.

  Zakładając, że liczba wierzchołków otoczki $k \leqslant m$(gdy $k > m$ przerywamy algorytm, więc złożoność pozostaje ta sama), to ostatecznie mamy złożoność dla drugiej części rzędu : $O(k\left\lceil(n/m)\right\rceil log(m)) = O(nlog(m))$w idealnym przypadku $O(nlog(k))$

Cała złożoność głównej części algorytmu, to $O(nlog(m))$, gdzie m jest wybraną wiellkości podzbioru. Złożonośc algorytmu dla próbowania algorytmu z kolejnymi m postaci 2^{2^m} dla $m \geqslant 1$, to: Iteracja zako

4.6.4 Kod

```python
from divide import *
from det import *
from tangentBothsides import *

def compr(p, q, current,
            accur=10 ** (-6)):  # jezeli p jest po prawej  odcinka [current,q] - jest 'wiekszy', to zwracamy 1
    if det(current, p, q) > accur:
        return -1
    elif det(current, p, q) < accur:
        return 1
    else:
        return 0

def nextvert(C, curr):  # dla danego punktu wspolzednymi z Q[i][j] jesli jest to punkt nalezacy do finalnej otoczki, to
    # zwraca nastepny punkt nalezacy do finalnej otoczki zadanego w takich samych wspolzednych Q[nxt[0]][nxt[1]]
    i, j = curr
    nxt = (i, (j + 1) % len(C[i]))
    for k in range(len(C)):
        t = tangent(C[i][j], C[k])
        if t != None and k != i and compr(C[nxt[0]][nxt[1]], C[k][t], C[i][j]) > 0 and (k, t) != (curr):
            nxt = (k, t)

    return nxt

def chanUtil(points, m):
    Q = divide(points, m)
    C = []
    for i in range(len(Q)):
        C.append(Graham(Q[i]))

    curr = (0, 0)
    ans = []
    i = 0
    while i < m:
        ans.append(C[curr[0]][curr[1]])
        if nextvert(C, curr) == (0, 0):
            return ans
        curr = nextvert(C, curr)
        i += 1

    return None

def chan(points):
    n = len(points)
    m = 4
    hoax = None
    while hoax == None:
        hoax = chanUtil(points, m)
```

```
        m = min(n, m * m)

    return hoax
```

4.7 Algorytm QuickHull

4.7.1 Opis działania

Algorytm QuickHull polega na rekurencyjnym wyznaczaniu kolejnych punktów otoczki.

1. Algorytm rozpoczynamy od wyznaczenie dwóch punktów skrajnych a,b - tj. o najmniejszej i największej współżędnej x-owej.

2. Następnie uruchamiamy funkcję rekurencyjnego znajdowania łuku należącego do otoczki między danym punktami należącymi do tej otoczki p,q na prawo od odcinka —p,q—. Otoczką jest suma punktów a, wyniku działania funkcji rekurencyjnej dla odcinka —a,b—,b oraz wyniku działania funkcji rekurencyjnej dla —b,a—

3. Funkcja rekurencyjnego wyznaczenia łuku należącego do otoczki między punktami p i q polega na :

    - Wyznaczeniu nabardziej oddalonego punktu na prawo od —p,q— jeśli są punkty po prawej.
    - Jeśli nie ma takich punktów, to takiego łuku nie ma i zwracamy pustą tablicę.
    - W przeciwnym przypadku p,q należą do otoczki, to wyznaczony punkt skrajny r musi należeć do otoczki.
    - Skoro p,k,r należy do otoczki, to wszystkie wierzchołki wewnątrz trójkąta pkr napewno do najmniej nie należą - usuwamy je.
    - Szukany łuk, to suma działania tej samej funkcji dla punktów p,r, punktu r , oraz wyniku tej funkcji dla punktów r,q w zadanej kolejności.
    - Na koniec zwracamy wyznaczony w ten sposób łuk.

4.7.2 Szczegóły

- Rozpatrywane punkty p,q,r zawsze są podane w kolejności przeciwnej do ruchu wskazówek zegara. Aby usunąć punkty wewnątrz takich trójkątów należy dla każdego punktu z rozważanych sprawdzić, należy do danego trójkąta.

- Sprawdzenie, czy dany punkt należy do trójkąta pkr wykonujemy poprzez sprawdzenie, czy dla każdego z odcinków pr, rq, qp dany punkt znajduje się na lewo od tego odcinka, bądź jest z nim współliniowy.

- Porównywanie odległości punktów r znajdujących się na prawo od odcinka pq wykonujemy za pomocą wyznacznika. Jest on wprostproporcjonlny do pola trójkąta rozpiętego na wektorach pq,pr. Ponieważ odcinek pq ma stałą długośćdla każdego r, to wyznacznik ten jest wprostproporcjonalny do wysokości tego trójkąta opuszczonej na bok pq - odległości punktów.

4.7.3 Złożoność

Złożoność głównej części algorytmu.

- Złożoność pierwsszej części składa się na :
  - Podział zbioru punktów na podzbiory O(n);
  - Wyznaczenie otoczek dla podzbiorów. Mamy $\lceil (n/m) \rceil$ podzbiorów rozmiaru m, dla każdego z nich wyznaczamy otoczkę algorytmem Grahama. Algorytm Grahama działa $O(nlog(n))$. Więc łącznie mamy $O(\lceil (n/m) \rceil mlog(m)) = O(nlog(m))$.

  Łącznie dla pierewszej części mamy $O(nlog(m))$, gdzie m jest wybranym maksymalnym rozmiarem podzbiorów.

- Złożoność drugiej części składa się na :
  - Wyznaczenie następnego punktu dla każdego punktu z otoczki głównej o rozmiarze k;
  - Wyznacznie następnego punktu składa się na wyznaczenie dla każdej z m podotoczek stycznej do tej podotoczki. Styczną wyznaczamy z pomocą algorytmu wyszukiwania binarnego w czasie $O(log(m))$ (otoczka C_i ma conajwyżej m wierzchołków). Otoczek jest $\lceil (n/m) \rceil$, a zatem czas wyznaczenia kolejngo wierzchołka otoczki to $O(\lceil (n/m) \rceil log(m))$.

  Zakładając , że liczba wierzchołków otoczki k¡=m(gdy k¿m przerywamy algorytm, więc złożoność pozostaje ta sama), to ostatecznie mamy złożoność dla drugiej części rzędu : O(k*ceil(n/m)*log(m)) = O(nlog(m)) w idealnym przypadku O(nlog(k))

Cała złożoność głównej części algorytmu, to $O(nlog(m))$, gdzie m jest wybraną wiellkości podzbioru. Złożonośc algorytmu dla próbowania algorytmu z kolejnymi m postaci 2^{2^m} dla $m \geqslant 1$, to: Iteracja zako

4.7.4 Kod

```
def furthest(a, b, considering):
    n = len(considering)
    i = 0
    ans = None
    while i < n:
        if det(a, b, considering[i]) < 0:  # rozwazany wierzcholek jest po prawej stronie ab
            if ans == None or det(a, b, considering[i]) < det(a, b,
                                                              ans):  # | det(a,b,c)| = 1/2|ab|*h, gdzie h jest wysokoscia z c na ab
                ans = considering[i]
        i += 1
    return ans

```

```python
def insideTriangle(a, b, c, i):
    if det(a, b, i) > 0 and det(b, c, i) > 0 and det(c, a, i) > 0:
        return True
    return False

def removeInner(a, b, c, considering):
    new=[]
    for i in considering:
        if not insideTriangle(a, b, c, i):
            new.append(i)
    considering.clear()
    considering+=new

def quickHullUtil(a, b, considering):
    if len(considering) == 0:
        return []

    c = furthest(a, b, considering)
    if c == None:
        return []
    considering.remove(c)

    removeInner(a, c, b, considering)
    return quickHullUtil(a, c, considering) +[c]+ quickHullUtil(c, b, considering)

def quickHull(points):
    a = min(points, key=lambda x: x[0])
    b = max(points, key=lambda x: x[0])

    considering = deepcopy(points)

    considering.remove(a)
    considering.remove(b)
```