# DOKUMENTACJA PROJEKTU
# Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Algorytmy geometryczne

Informatyka 3. sem., 1. st., AGH

Kacper Kafara
Łukasz Czarniecki

# Spis treści

# Spis rysunków

## Spis tablic

# 1 Informacje techniczne

## 1.1 Budowa programu

Program złożony jest z następujących części:

- *lib* – moduł biblioteczny – zawiera zbiór pomocniczych funkcji i struktur danych wykorzystywanych przez algorytmy
- *pure* – moduł z algorytmami w *czystej postaci* tj. nie posiadające części wizualizacyjnej
- *vis* – moduł z algorytmami wraz z kodem odpowiadającym za wizualizację
- *kafara_czarniecki_program.ipynb* – główny plik programu, notebook z prezentacją algorytmów

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* – funkcje generujące zbiory punktów różnych typów
3. *sorting.py* – implementację iteracyjnej wersji algorytmu *QuickSort* wykorzystywaną w algorytmie Grahama
4. *stack.py* – klasę implementującą *stos*
5. *util.py* – szereg funkcji pomocniczych wykorzystywanych przez zaimplementowane algorytmy
6. *mytypes.py* – definicje typów stworzone w celu zwiększenia czytelności kodu.
7. *timemeasure.py* – funkcje do pomiaru czasu wykonania algorytmów
8. *tangent.py* – metoda wyznaczająca styczną (punkt styczności) do wielokąta, przechodzącą przez zadany punkt

### Kod modułu *mytypes*

```python
from typing import Tuple, List

Point = Tuple[float, float]

Segment = Tuple[Point, Point]

ListOfPoints = List[Point]

ListOfSegments = List[Segment]
```

### Kod modułu *getrand*

```python
import numpy as np
from typing import Tuple
from numpy import random

def dist(point_a, point_b):
    return np.sqrt((point_a[0] - point_b[0])** 2 + (point_a[1] - point_b
    [1])**2)

def rand_seg_point(point_a, point_b):
    rng = np.random.default_rng()

    point = np.empty(shape=2)

    if point_a[0] == point_b[0]:
        min_y = min(point_a[1], point_b[1])
        max_y = max(point_a[1], point_b[1])
        point[0] = point_a[0]
        point[1] = rng.random() * (max_y - min_y) + min_y

    else:
        min_x = min(point_a[0], point_b[0])
        max_x = max(point_a[0], point_b[0])

        a = (point_b[1] - point_a[1]) / (point_b[0] - point_a[0])
        b = (point_a[1] - a * point_a[0])

        param_t = rng.random() * (max_x - min_x) + min_x
        point[0] = param_t
        point[1] = a * param_t + b

    return point


def rand_rect_points(n: int, vertices):
    rng = np.random.default_rng()
    A, B, C, D = vertices[0], vertices[1], vertices[2], vertices[3]
    len_AB = dist(A, B)
    len_BC = dist(B, C)

    points = np.empty(shape=(n,2))
```

```

    for i in range(n):
        side = rng.random(1) * 2 * (len_AB + len_BC)

        if side < len_BC:
            points[i] = rand_seg_point(A, D)
        elif side < 2 * len_BC:
            points[i] = rand_seg_point(B, C)
        elif side < 2 * len_BC + len_AB:
            points[i] = rand_seg_point(A, B)
        else:
            points[i] = rand_seg_point(C, D)

    return points

def rand_in_range(  low: float = 0,
                    high: float = 0,
                    data_type: str = 'float64') -> np.array:
    rng = random.default_rng()

    return np.array((rng.random(1) * (high - low) + low), dtype=data_type)


def rand_arr(n: int,
            low: float,
            high: float,
            data_type: str = 'float64') -> np.array:

    rg = np.random.default_rng()

    return np.array( (rg.random(n) * (high - low) + low), dtype=data_type)

def rand_point2_set(n: int,
                    low: float = 0,
                    high: float = 1,
                    data_type: str = 'float64') -> np.array:
    return np.array(np.random.random((n, 2)) * (high - low) + low, dtype=data_type)

def rand_circle_points( n: int = 10,
                        x: float = 0,
                        y: float = 0,
                        r: float = 1,
                        data_type: str = 'float64') -> np.array:
    rng = np.random.default_rng()

    circle = np.empty(n * 2, dtype=data_type).reshape(n, 2)

    for i in range(n):
        rn = rng.random() * 2 * np.pi
        circle[i][0] = r * np.cos(rn) + x
        circle[i][1] = r * np.sin(rn) + y
```

```python

    return circle
```

### Kod modułu *sorting*

```python
import os
from sys import path
path.append(os.path.dirname(os.path.realpath(__file__)) + "/../")

from random import randint
from typing import Callable, List
from lib.stack import Stack


def partition_cmp(arr: List, p: int, q: int, cmp: Callable = lambda x, y:
     x <= y) -> int:
    i = p - 1

    pivot_idx = randint(p, q)
    pivot = arr[pivot_idx]

    # zamieniamy miejscami pivot z ostatnim elementem
    arr[pivot_idx], arr[q] = arr[q], arr[pivot_idx]

    for j in range(p, q):
        if cmp(arr[j], pivot):
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[q] = arr[q], arr[i + 1]

    return i + 1


def qsort_iterative(arr: List, cmp: Callable = lambda x, y: x <= y):
    stack = Stack(len(arr))

    stack.push(0)
    stack.push(len(arr) - 1)

    while not stack.is_empty():
        q = stack.top()
        stack.pop()

        p = stack.top()
        stack.pop()

        pivot_idx = partition_cmp(arr, p, q, cmp=cmp)

        if pivot_idx - 1 > p:
            stack.push(p)
            stack.push(pivot_idx - 1)

        if pivot_idx + 1 < q:
```

```
            stack.push(pivot_idx + 1)
            stack.push(q))
```

### Kod modułu *stack*

```
class Stack:
    def __init__(self, size = 0):
        self.s = [] if size == 0 else [None] * size
        self.size = size
        self.itop = -1

    def push(self, item):
        if self.itop < self.size - 1:
            self.itop += 1
            self.s[self.itop] = item
        else:
            self.itop += 1
            self.size += 1
            self.s.append(item)

    def pop(self):
        if self.itop >= 0:
            self.itop -= 1

    def top(self):
        if self.itop >= 0:
            return self.s[self.itop]
        else:
            return None

    def sec(self):
        if self.itop > 0:
            return self.s[self.itop - 1]
        else:
            return None

    def is_empty(self):
        return not (self.itop >= 0)

    def getsize(self):
        return self.itop+1

    def __str__(self):
        return str(self.s[:self.itop+1])
```

### Kod modułu *util*

```
import json
from typing import Literal, Union
from lib.mytypes import *
from random import randint
import matplotlib.colors as mcolors
import math
import numpy as np
```

```
import csv

def det3x3( ux, uy, uz,
            vx, vy, vz,
            wx, wy, wz) -> float:
    return (ux * vy * wz) + (uy * vz * wx) + (uz * vx * wy) - (wx * vy * uz) - (wy * vz * ux) - (wz * vx * uy)


def orientation(p1: Point, p2: Point, p3: Point, eps: float = 1e-5) -> Literal[-1, 0, 1]:
    det = det3x3(p1[0], p1[1], 1,
                 p2[0], p2[1], 1,
                 p3[0], p3[1], 1)
    if det < -eps:
        return -1
    elif det < eps:
        return 0
    else:
        return 1


def save_points_to_json(path: str, points: ListOfPoints, indent: int = None) -> None:
    with open(path, 'w') as file:
        file.write(json.dumps(points, indent = indent))


def load_points_from_json(path: str) -> None:
    with open(path, 'r') as file:
        return json.load(file)


def save_data_csv(path: str, data):
    with open(path, 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        for row in data:
            writer.writerow(row)


def index_of_min(points: ListOfPoints, cmp_idx = 0) -> Union[int, None]:
    if len(points) < 1 or cmp_idx < 0 or cmp_idx > 1: return None

    min_el = points[0]
    min_idx = 0

    for i in range(1, len(points)):
        if  points[i][cmp_idx] < min_el[cmp_idx] \
            or \
            points[i][cmp_idx] == min_el[cmp_idx] and points[i][1 - cmp_idx] < min_el[1 - cmp_idx]:

            min_el = points[i]
            min_idx = i
```

```python

    return min_idx


def index_of_max(points: ListOfPoints, cmp_idx = 0) -> Union[int, None]:
    if len(points) < 1 or cmp_idx < 0 or cmp_idx > 1: return None

    max_el = points[0]
    max_idx = 0

    for i in range(1, len(points)):
        if  points[i][cmp_idx] > max_el[cmp_idx] \
            or \
            points[i][cmp_idx] == max_el[cmp_idx] and points[i][1 - cmp_idx] > max_el[1 - cmp_idx]:

            max_el = points[i]
            max_idx = i

    return max_idx


def dist_sq(p1, p2):
    return (p2[0] - p1[0]) ** 2 + (p2[1] - p1[1]) ** 2

def swap_points(p, q):
    p[0], p[1], q[0], q[1] = q[0], q[1], p[0], p[1]



def divide(points,m): #dzieli zbior punktow na w miarae rowne podzbiory o rozmiarze m lub m-1
    n=len(points)
    for i in range(1,n):#gwarantuje, ze w pierwszym zbiorze Qi pierwszy element jest najnizszy, czyli nalezy do otoczki ostatecznej
        if points[i][1] < points[0][1]:
            buf=points[i]
            points[i] = points[0]
            points[0]=buf

    k=math.ceil(n/m)
    Q=[[] for i in range(k)]
    i=0
    while i<n:
        for j in range(k):
            if i==n:
                break
            Q[j].append(points[i])
            i+=1
    if len(Q[0]) > m:
        return None

    return Q

```

```
def makeSheaf(Points): #laczy punkty w kolejnosci jakiej sa podane
    Sheaf=[]
    for i in range(len(Points)-1):
        Sheaf.append([Points[i],Points[i+1]])
    return Sheaf

def makeFullSheaf(Points): #laczy punkty w kolejnosci jakiej sa podane, dodatkowo domyka cylk
    Sheaf=[]
    for i in range(len(Points)-1):
        Sheaf.append([Points[i],Points[i+1]])
    Sheaf.append([Points[len(Points)-1],Points[0]])

    return Sheaf

def length(v):
     return np.sqrt((v[1][0]-v[0][0])**2+(v[1][1]-v[0][1])**2)

def det(a,b,c):
    return a[0]*b[1]-a[0]*c[1]-b[0]*a[1]+b[0]*c[1]+c[0]*a[1]-c[0]*b[1]


def tangent_r(p, Q, accur=10 ** (-7)):  # Q-zbior punktow w formie otoczki
    ln = len(Q)

    def tangetUtil(p, Q, l, r):
        if r < l:  # zdarza sie tylko, gdy punkt jest wewnatrz otoczki
            return None

        mid = (l + r) // 2
        if det(Q[0], Q[1], p) >= accur and det(Q[ln - 1], Q[0], p) >= accur:
            if (det(Q[0], p, Q[mid]) <= -accur) or (det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and \
                                                    det(p, Q[mid], Q[(mid - 1) % ln]) <= -accur) or \
                    (det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det(p, Q[mid], Q[(mid - 1) % ln]) > -accur):
                return tangetUtil(p, Q, mid + 1, r)

        else:
            if det(Q[0], p, Q[mid]) > -accur and \
                    ((det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det(p, Q[mid], Q[(mid - 1) % ln]) > -accur) or \
                     (det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det(p, Q[mid], Q[
                         (mid - 1) % ln]) <= -accur)):  # chyba nie potrzebne sprawdz na koncu
                return tangetUtil(p, Q, mid + 1, r)

        if (det(p, Q[mid], Q[(mid + 1) % ln]) > -accur and det(p, Q[mid], Q[(mid - 1) % ln]) > -accur) \
                or (
```

```python
                -accur < det(p, Q[mid], Q[(mid + 1) % ln]) < accur and (Q[mid][0] <= p[0] <= Q[(mid + 1) % ln][0]) and \
                (Q[mid][1] <= p[1] <= Q[(mid + 1) % ln][1])):

            while -accur < det(p, Q[mid], Q[(mid + 1) % ln]) < accur and length([Q[(mid + 1) % ln], p]) > length(
                    [Q[mid], p]):
                mid = (mid + 1) % ln  # jesli jest styczna wspolliniowa, to bierzmy pod uwage punkt blizszy
            return mid

        else:
            return tangetUtil(p, Q, l, mid - 1)

    return tangetUtil(p, Q, 0, ln - 1)

def tangent_l(p, Q, accur=10 ** (-7)):  # Q-zbior punktow w formie otoczki
    ln = len(Q)

    def tangetUtil(p, Q, l, r):
        if r < l:  # zdarza sie tylko, gdy punkt jest wewnatrz otoczki
            return None

        mid = (l + r) // 2
        if det(Q[0], Q[1], p) >= accur and det(Q[ln - 1], Q[0], p) >= accur:
            if (det(Q[0], p, Q[mid]) >= accur) or (det(p, Q[mid], Q[(mid + 1) % ln]) >= accur and \
                                                    det(p, Q[mid], Q[(mid - 1) % ln]) >= accur) or \
                    (det(p, Q[mid], Q[(mid + 1) % ln]) <= -accur and det(p, Q[mid], Q[(mid - 1) % ln]) > -accur):
                return tangetUtil(p, Q, l, mid-1)

        else:
            if  det(Q[ln - 1], Q[0], p) >= accur and det(Q[0], Q[1], p) <= - accur:
                return 0
            if det(Q[0], p, Q[mid]) < accur and \
                    ((det(p, Q[mid], Q[(mid + 1) % ln]) >= accur and det(p, Q[mid], Q[(mid - 1) % ln]) < accur) or \
                     (det(p, Q[mid], Q[(mid + 1) % ln]) >= accur and det(p, Q[mid], Q[
                         (mid - 1) % ln]) >= accur)):  # chyba nie potrzebne sprawdz na koncu
                if ln > 2:
                    return tangetUtil(p, Q, l, mid-1)

        if (det(p, Q[mid], Q[(mid + 1) % ln]) < accur and det(p, Q[mid], Q[(mid - 1) % ln]) < accur) \
                or (
                -accur < det(p, Q[mid], Q[(mid + 1) % ln]) < accur and (Q[mid][0] <= p[0] <= Q[(mid + 1) % ln][0]) and \
```

```
                (Q[mid][1] <= p[1] <= Q[(mid + 1) % ln][1])):

            while -accur < det(p, Q[mid], Q[(mid - 1) % ln]) < accur and length([Q[(mid - 1) % ln], p]) > length(
                    [Q[mid], p]):
                mid = (mid - 1) % ln  # jesli jest styczna wspolliniowa, to bierzmy pod uwage punkt blizszy
            return mid

        else:
            return tangetUtil(p, Q, mid + 1, r)

    return tangetUtil(p, Q, 0, ln - 1)


def compr(p,q,current,accur=10**(-6)):
        if det(current,p,q)>accur:
            return -1
        elif det(current,p,q)<accur:
            return 1
        else:
            return 0
```

### Kod modułu *timemeasure*

```
import os
from sys import path
path.append(os.path.dirname(os.path.realpath(__file__)) + "/../")

from timeit import default_timer as timer
from typing import Any, Callable, List
from pprint import pprint
import numpy as np

def get_exec_time(func: Callable, *args, points = []) -> float:
    tstart = timer()
    func(points, *args)
    tstop = timer()
    return (tstop - tstart)


def avg_exec_time(func: Callable, *args, points = [], times = 1) -> float:
    total_exec_time = 0

    for _ in range(times):
        points_copy = points.copy()
        total_exec_time += get_exec_time(func, *args, points = points_copy)

    return total_exec_time / times
```

### Kod modułu *tangent*

```
def tangent(p, Q, accur=0):  # Q-zbior punktow w formie otoczki
```

```python
    ln = len(Q)

    def tangetUtil(p, Q, l, r):
        if r < l:  # zdarza sie tylko, gdy punkt jest wewnatrz otoczki
            return None

        mid = (l + r) // 2
        if det(Q[0], Q[1], p) > 0 and det(Q[ln - 1], Q[0], p) > 0:
            if (det(Q[0], p, Q[mid]) < 0) or (det(p, Q[mid], Q[(mid + 1) % ln]) < 0 and \
                                                  det(p, Q[mid], Q[(mid - 1) % ln]) < 0) or \
                    (det(p, Q[mid], Q[(mid + 1) % ln]) < 0 and det(p, Q[mid], Q[(mid - 1) % ln]) >= 0):
                return tangetUtil(p, Q, mid + 1, r)

        else:
            if det(Q[0], p, Q[mid]) >= 0 and \
                    ((det(p, Q[mid], Q[(mid + 1) % ln]) < 0 and det(p, Q[mid], Q[(mid - 1) % ln]) >= 0) or \
                     (det(p, Q[mid], Q[(mid + 1) % ln]) < 0 and det(p, Q[mid], Q[
                         (mid - 1) % ln]) < 0)):  # chyba nie potrzebne sprawdz na koncu
                return tangetUtil(p, Q, mid + 1, r)

        if det(p, Q[mid], Q[(mid + 1) % ln]) >= 0 and det(p, Q[mid], Q[(mid - 1) % ln]) >= 0 \
                or (det(p, Q[mid], Q[(mid + 1) % ln]) == 0 and (Q[mid][0] <= p[0] <= Q[(mid + 1) % ln][0]) and \
                    (Q[mid][1] <= p[1] <= Q[(mid + 1) % ln][1])):

            while (det(p, Q[mid], Q[(mid + 1) % ln]) == 0):
                mid = (mid + 1) % ln  # jesli jest styczna wspolliniowa, to bierzmy pod uwage punkt blizszy
            return mid

        else:
            return tangetUtil(p, Q, l, mid - 1)

    return tangetUtil(p, Q, 0, ln - 1)
```

### 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"

Kody algorytmów znajdują się w sekcji *Algorytmy*.

#### 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 3.8.3 64-bit lub nowszy wraz z modułami:
   - *matplotlib*
   - *numpy*
2. Jupyter Notebook

### 1.3 Korzystanie z programu

#### 1.3.1 Uruchomienie programu

W celu uruchomienia wizualizacji algorytmów należy uruchomić notebook (poprzez Jupyter Notebook) *kafara_czarniecki_program.ipynb*, oraz wykonywać kolejne komórki notatnika.

W celu uruchomienia pomiarow wydajności algorytmów należy przejść do sekcji *Pomiary czasu* wykonać pierwszą komórkę (z importami algorytmów) a następnie przeprowadzać testy i wyświetlać rezultaty w interesujących nas przypadkach.

## 2 Oznaczenia i definicje

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

*def.* 1. **Zbiorem wypukłym** nazwiemy dowolny podzbiór płaszczyzny taki, że dla każdych dwóch punktów do niego należących, odcinek je łączący również należy do tego zbioru.

*def.* 2. **Otoczką wypukłą** dowolnego zbioru punktów $S$ płaszczyzny nazwiemy najmniejszy zbiór wypukły $CH(S)$ zawierający $S$.

Algorytmicznie otoczkę wypułką dowolnego zbioru $S$ punktów płaszczyzny reprezentujemy jako ciąg punktów (wierzchołołków) $< v_1, v_2, \ldots, v_n >$ wielokąta wypukłego, gdzie $\forall i \in \{1, \ldots, n\}$ $v_i$ jest poprzednikiem $v_{i+1}$ (w kolejności wierzchołków przeciwnej do ruchu wskazówek zegara).

# 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, zaczynając 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

```
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] ]

```

```python
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**

Niech $Q$ będzie punktem przez który ma przechodzić styczna.

1. W czasie liniowym znajdujemy punkt otoczki $P_i$ o największej odciętej (jeżeli jest wiele, to wybieramy ten o najmniejszej rzędnej)
2. Dopóki $P_{i+1}$ nie znajduje się po lewej stronie odcinka $QP_i$:
   (a) Jeżeli $Q$, $P_i$, $P_{i+1}$ są współliniowe oraz $P_i$ leży dalej (lub w takiej samej odległości) $Q$ niż $P_{i+1}$ to przerwij działanie pętli.
   (b) $P_i := P_{i+1}$
3. W całkowicie analogiczny sposób wyznaczamy dolną styczną.

### 4.4.3 Złożoność

Posortowanie punktów zajmuje $O(n\, lgn)$. Wydaje główna pętla programu wykonuje się w czasie $O(n)$, ponieważ możemy usunąć maksymalnie $k-3$ punkty (gdy $k$ jest liczebnością zbioru punktów otoczki znaną w danej iteracji), ale zauważmy, że każdy z punktów usuwany jest co najwyżej raz. Wyszukanie stycznych w głównej pętli także zajmuje czas liniowy, więc główna pętla programu wykonuje się w czasie liniowym. Zatem złożoność algorytmu jest rzędu $O(n\, lgn)$

### 4.4.4 Kod

```python
def rltangent(polygon: ListOfPoints, point: Point):
    n = len(polygon)

    right = index_of_max(polygon, cmp_idx=0)

    left = right

    left_orient = orientation(point, polygon[left % n], polygon[(left-1)%n])
    while left_orient != -1:
        if left_orient == 0 and dist_sq(point, polygon[left]) >= dist_sq(point, polygon[(left-1)%n]):
            break
        left = (left-1) % n
        left_orient = orientation(point, polygon[left % n], polygon[(left-1)%n])
```

```

    right_orient = orientation(point, polygon[right%n], polygon[(right+1)%n])
    while right_orient != 1:
        if right_orient == 0 and dist_sq(point, polygon[right]) >= dist_sq(point, polygon[(right+1)%n]):
            break
        right = (right+1)%n
        right_orient = orientation(point, polygon[right%n], polygon[(right+1)%n])


    return left, right


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 )):
        rltang = rltangent(convex_hull, point2_set[i])
        left_tangent_idx = rltang[0]
        right_tangent_idx = rltang[1]

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

        deletion_side: Literal[-1, 0, 1] = orientation(left_tangent_point, right_tangent_point, point2_set[i])

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

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

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

            convex_hull.insert(left % len(convex_hull), point2_set[i])
```

```python
        else:
            convex_hull = [point2_set[left_tangent_idx], 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 $L$ i $P$ nie tworzą górnej stycznej, ”wychodzimy w góręńaprzemiennie punktami $L$ i $P$.
3. Analogicznie wyznaczamy dolną styczną.
4. Usuwamy punkty zawierające się we wnętrzu nowo utworzonej otoczki.

### 4.5.3 Złożoność

- Początkowe sortowanie: $O(nlgn)$
- Rekurencja: $T(n) = 2T(\frac{n}{2}) + O(n) \implies T(n) = O(nlgn)$
- Każde łączenie otoczek: $O(n)$

Algorytm dziel i zwyciężaj ma zatem złożoność $O(nlgn)$

### 4.5.4 Kod

```
def merge_convex_hulls(left_convex_hull: ListOfPoints, right_convex_hull: ListOfPoints) -> List[Point]:
    left_ch_size = len(left_convex_hull)
    right_ch_size = len(right_convex_hull)

    # znajdujemy prawy skrajny punkt lewej otoczki
    left_ch_rightmost_idx = index_of_max(left_convex_hull, cmp_idx=0)
    right_ch_leftmost_idx = index_of_min(right_convex_hull, cmp_idx=0)

    left = left_convex_hull[left_ch_rightmost_idx]
    right = right_convex_hull[right_ch_leftmost_idx]
    left_idx = left_ch_rightmost_idx
    right_idx = right_ch_leftmost_idx

    left_flag, right_flag = True, True
    while   orientation(left, right, right_convex_hull[(right_idx - 1) % right_ch_size]) != -1 and right_flag\
            or \
            orientation(right, left, left_convex_hull[(left_idx + 1) % left_ch_size]) != 1 and left_flag:

        left_flag, right_flag = False, False

        # podnosimy punkt na prawej otoczce
        left_right_orient = orientation(left, right, right_convex_hull[(right_idx - 1) % right_ch_size])
        while left_right_orient != -1:
            if left_right_orient == 0 and dist_sq(left, right) >= dist_sq(left, right_convex_hull[(right_idx - 1) % right_ch_size]):
                right_flag = False
                break

            right_idx = (right_idx - 1) % right_ch_size
            right = right_convex_hull[right_idx]
```

```python
            left_right_orient = orientation(left, right, right_convex_hull[(right_idx - 1) % right_ch_size])
        else:
            right_flag = True

        # podnosimy punkt na lewej otoczce
        right_left_orient = orientation(right, left, left_convex_hull[(left_idx + 1) % left_ch_size])
        while right_left_orient != 1:
            if right_left_orient == 0 and dist_sq(right, left) >= dist_sq(right, left_convex_hull[(left_idx + 1) % left_ch_size]):
                left_flag = False
                break

            left_idx = (left_idx + 1) % left_ch_size
            left = left_convex_hull[left_idx]
            right_left_orient = orientation(right, left, left_convex_hull[(left_idx + 1) % left_ch_size])
        else:
            left_flag = True


    upper_tangent_left_idx = left_idx
    upper_tangent_right_idx = right_idx

    # dolna styczna
    left = left_convex_hull[left_ch_rightmost_idx]
    right = right_convex_hull[right_ch_leftmost_idx]
    left_idx = left_ch_rightmost_idx
    right_idx = right_ch_leftmost_idx

    left_flag, right_flag = True, True
    while   orientation(left, right, right_convex_hull[(right_idx + 1) % right_ch_size]) != 1 and right_flag \
            or \
            orientation(right, left, left_convex_hull[(left_idx - 1) % left_ch_size]) != -1 and left_flag:

        left_flag, right_flag = False, False

        # opuszczamy punkt na prawej otoczce
        left_right_orient = orientation(left, right, right_convex_hull[(right_idx + 1) % right_ch_size])
        while left_right_orient != 1:
            if left_right_orient == 0 and dist_sq(left, right) >= dist_sq(left, right_convex_hull[(right_idx + 1) % right_ch_size]):
                right_flag = False
                break

            right_idx = (right_idx + 1) % right_ch_size
            right = right_convex_hull[right_idx]
            left_right_orient = orientation(left, right, right_convex_hull[(right_idx + 1) % right_ch_size])
        else:
```

```
            right_flag = True

        # opuszczamy punkt na lewej otoczce
        right_left_orient = orientation(right, left, left_convex_hull[(left_idx - 1) % left_ch_size])
        while right_left_orient != -1:
            if right_left_orient == 0 and dist_sq(right, left) >= dist_sq(right, left_convex_hull[(left_idx - 1) % left_ch_size]):
                left_flag = False
                break

            left_idx = (left_idx - 1) % left_ch_size
            left = left_convex_hull[left_idx]
            right_left_orient = orientation(right, left, left_convex_hull[(left_idx - 1) % left_ch_size])
        else:
            left_flag = True


    lower_tangent_left_idx = left_idx
    lower_tangent_right_idx = right_idx

    merged_convex_hull = [ ]

    while upper_tangent_left_idx != lower_tangent_left_idx:
        merged_convex_hull.append(left_convex_hull[upper_tangent_left_idx])
        upper_tangent_left_idx = (upper_tangent_left_idx + 1) % left_ch_size
    else:
        merged_convex_hull.append(left_convex_hull[lower_tangent_left_idx])


    while lower_tangent_right_idx != upper_tangent_right_idx:
        merged_convex_hull.append(right_convex_hull[lower_tangent_right_idx])
        lower_tangent_right_idx = (lower_tangent_right_idx + 1) % right_ch_size
    else:
        merged_convex_hull.append(right_convex_hull[lower_tangent_right_idx])

    return merged_convex_hull


def divide_conq(point2_set: List[Point], k: int = 2) -> Union[List[Point], None]:
    if len(point2_set) < 3 or k <= 0: return None


    def divide_conq_rec(point2_set: List[Point]) -> List[Point]:
        if len(point2_set) <= 2: return point2_set
        elif len(point2_set) <= k: return jarvis(np.array(point2_set))
```

```

        left_convex_hull = divide_conq_rec(point2_set[ : len(point2_set)
    // 2])
        right_convex_hull = divide_conq_rec(point2_set[len(point2_set) //
     2 : ])

        return merge_convex_hulls(left_convex_hull, right_convex_hull)

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

    return divide_conq_rec(point2_set)
```

## 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 nadrzędną 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 popsuć złożoności dużą ilością powtórzeń głównej części algorytmu najlepiej za każdym razem parametr $m$ podnosić do kwadratu. W przypadku gdy $m \geqslant n$ za $m$ przyjmujemy $n$. Wtedy algorytm Chana sprowadza się do algorytmu Grahama.

### 4.6.2 Szczegóły

- Najniższy punkt zbioru wyznaczamy w czasie liniowym. W trakcie podziału zpewniamy, że znajdzie się on w otoczce $C_0$ w indeksie 0.
- Podział zapewia, że w każdym podzbiorze będzie m, lub m-1 punktów.
- Znajdowanie stycznej wykonujemy w czasie O(log n). Wykorzystujemy do tego binary search, można tak zrobić, ponieważ otoczka zadana jest przeciwnie do ruchu wskazówek zegara. Najpierw obieramy wskaźniki $l$ i $r$. Następnie obliczamy $mid$ $=(l+r)/2$. Gdy obaj sąsiedzi mida są po lewej stronie - mid jest szukanym indeksem. Gdy mid znajduje się po po prawej stronie odcinka $pq$, gdzie $p$, to pierwszy wierzchołek otoczki(pierwszy$l$), a $q$, to punkt do którego stycznej szukamy. Wtedy prawa styczna znajduje się w przedziale mid+1,r i urachamiamy tą samą funkcję dla tych indeksów. Gdy $mid$ jest współliniowy, bądź po lewej stronie z $pq$, to jeśli jego następnik(mid+1) jest po prawej stronie odcinka $|qmid|$, to również styczna się znajduje w przedziale mid+1 ,r. W pozostałych przypadkach znajduje się w przedziale l, mid-1.

### 4.6.3 Złożoność

Złożoność głównej części algorytmu.

- Na złożoność pierwszej części algorytmu składa się:
  - Podział zbioru punktów na podzbiory $O(n)$;
  - Wyznaczenie otoczek dla podzbiorów. Mamy $\lceil \frac{n}{m} \rceil$ podzbiorów rozmiaru m, dla każdego z nich wyznaczamy otoczkę algorytmem Grahama. Algorytm Grahama działa $O(nlog(n))$. Więc łącznie mamy $O(\lceil \frac{n}{m} \rceil \cdot mlog(m)) = O(nlog(m))$.

  Łącznie dla pierewszej części mamy $O(nlog(m))$, gdzie $m$ jest wybranym maksymalnym rozmiarem podzbiorów.
- Na złożoność drugiej części algorytmu składa się:
  - 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 : $O(k \lceil (n/m) \rceil \, log(m)) = O(nlog(m))$ w idealnym przypadku $O(nlog(k))$

Cała złożoność głównej części algorytmu, to $O(nlog(m))$, gdzie $m$ jest wybraną liczebnością podzbioru. Złożonośc algorytmu dla próbowania algorytmu z kolejnymi $m$ postaci $2^{2^m}$ dla $m \geqslant 1$, to: w takim razie złożoność można opisać wzorem $\sum_{t=1}^{\lceil loglogk \rceil} O(nlog(2^{2^t})) = O(n * 2^{1+\lceil loglogk \rceil}) = O(nlogk)$

### 4.6.4 Kod

```
def divide(points, m):
    n = len(points)
    for i in range(1,
                   n):  # gwarantuje, ze w pierwszym zbiorze Qi pierwszy element jest najnizszy, czyli nalezy do otoczki ostatecznej
        if points[i][1] < points[0][1]:
            buf = points[i]
            points[i] = points[0]
            points[0] = buf

    k = math.ceil(n / m)
    Q = [[] for i in range(k)]
    i = 0
    while i < n:
        for j in range(k):
            if i == n:
                break
            Q[j].append(points[i])
            i += 1
    if len(Q[0]) > m:
        return None

    return Q


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 < n:
        for j in range(k):
            if i == n:
                break
            Q[j].append(points[i])
            i += 1
    if len(Q[0]) > m:
        return None

    return Q

    def nextvert(C, curr,accur=10**(-7)):  # 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)):
            if k == i:
                continue

            t = tangent_r(C[i][j], C[k])
            if t == None:
                print("zle")
                continue

            if  det(C[i][j],C[nxt[0]][nxt[1]],C[k][t])<accur and (k,t)!=(curr):
                if det(C[i][j],C[nxt[0]][nxt[1]],C[k][t])>-accur:
                    if length([C[k][t],C[i][j]]) <= length([C[nxt[0]][nxt[1]],C[i][j]]):
                        continue
                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]])

            nxt = nextvert(C, curr)
            if nxt == (0, 0):
                return ans
            curr = nxt
            i += 1

        return None


    def chan(points, visual=False):
        plot = None
        n = len(points)
        m = 4
        hoax = None


        return hoax

    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 długość dla każdego r, to wyznacznik ten jest wprostproporcjonalny do wysokości tego trójkąta opuszczonej na bok pq - odległości punktów.

### 4.7.3 Złożoność

Pesymistyczna złożoność algorytmu to $O(n^2)$ - gdy wszystkie punkty zbioru znajdują się w otoczce. Jendakże w średnim przypadku złożoność wynosi $O(nlogn)$

### 4.7.4 Kod

```
from copy import deepcopy
from lib.det import *

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


def insideTriangle(a, b, c, i):
    accur=10**(-7)
```

```python
        if det(a, b, i) > -accur and det(b, c, i) > -accur and det(c, a, i) > -accur:
            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)
        b = max(points, key=lambda x: x)

        considering = deepcopy(points)

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

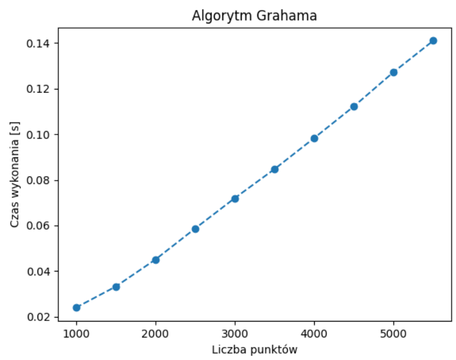
# 5 Wydajność algorytmów

Testy prowadzone były na następujących zbiorach:

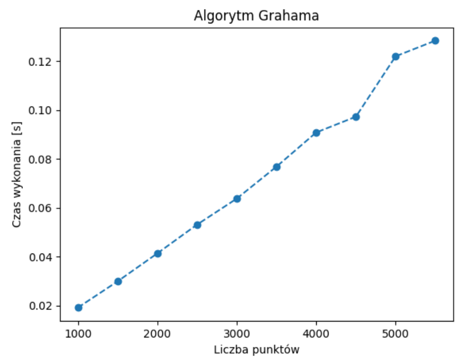
- typ A - losowo rozłożone punkty płaszczyzny o określonych zakresach współrzędnych
- typ B - losowo rozłożone punkty leżące na okręgu o zadanych parametrach
- typ C - losowo rozłożone punkty leżące na bokach prostokąta o zadanych parametrach
- typ D - losowo rozłożone punkty leżące na 2. przekątnych oraz 2. bokach kwadratu, umiejscowionego tak, że dwa boki pokrywają sie z osiami układu

Tablica 1: Czas wykonania algorytmu Grahama w zależności od typu zbioru testowego oraz mocy zbioru punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Typ zbioru | Czas wykonania [s] | | | | | | | | | |
| A | 0.0239 | 0.0331 | 0.045 | 0.0586 | 0.072 | 0.0847 | 0.0983 | 0.1122 | 0.1273 | 0.141 |
| B | 0.0191 | 0.0299 | 0.0413 | 0.0531 | 0.0637 | 0.0768 | 0.0908 | 0.0973 | 0.122 | 0.1284 |
| C | 0.069 | 0.1115 | 0.1392 | 0.1878 | 0.2349 | 0.2736 | 0.3153 | 0.351 | 0.3988 | 0.4601 |
| D | 0.4256 | 0.6523 | 0.924 | 1.2255 | 1.4233 | 1.8752 | 1.9285 | 2.3207 | 2.5692 | 2.88 |




Rysunek 1: Zbiór typu A, algorytm Grahama




Rysunek 2: Zbiór typu B, algorytm Grahama

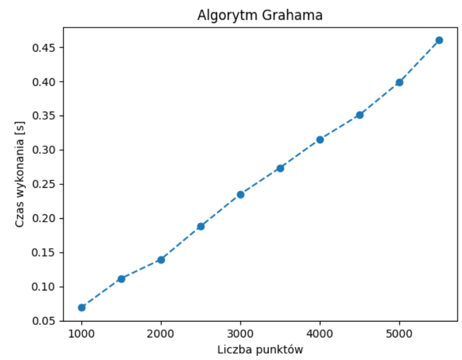
Charakterystyki zbiorów zostały dobrane w taki sposób aby odzwierciedlać możliwie różne zachowanie się poszczególnych algorytmów w zależności od ilości przypadków zdegenerowanych (liczebność współliniowych otoczek), ilości potrzebnych do wykonania porównań i operacji. Bardziej szczegółowe omówienie charakterystyki algorytmów dla poszczególnych zbiorów znajduje się w kolejnych sekcjach.

## 5.1 Algorytm Grahama

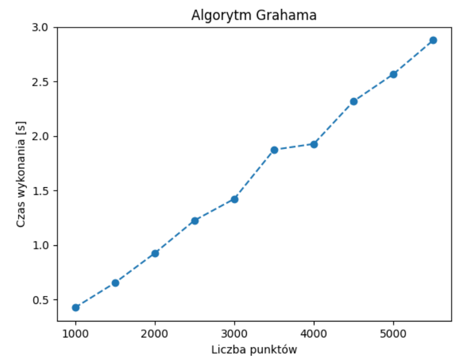
W tabeli 1 przedstawiamy czasy uzyskane przez algorytm Grahama dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 1, 2, 3, 4 widzimy ilustrację danych z tabeli 1.

## 5.2 Porównanie z przewidywaniami teoretycznymi

Kształt dla każdego z typu zbiorów dla algorytmu Grahama przypomina wykres liniowy. Zgdza się to z przewidywaniami teoretycznymi dla tegoż algorytmu - ma on złożoność O(n logn). Wykres o takiej złożoności taka bardzo przypomina wykres liniowy. Spośród algorytmów o tej złożoności Graham zdaje się być najwolniejszy. Ilość wierzchołków w otoczce nie ma znaczenia dla każdego rozkładu graham wykazuje się podobnym czasem.




Rysunek 3: Zbiór typu C, algorytm Grahama




Rysunek 4: Zbiór typu D, algorytm Grahama

Tablica 2: Czas wykonania algorytmu górna-dolna w zależności od typu zbioru testowego oraz mocy zbioru punktów.

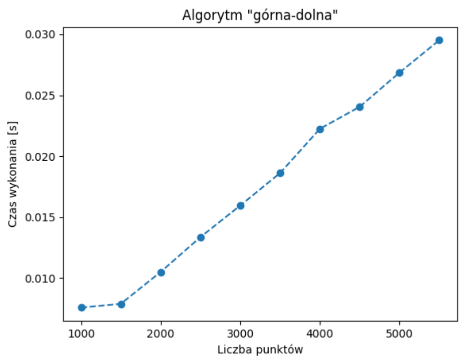
| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Typ zbioru | Czas wykonania [s] | | | | | | | | | |
| A | 0.0076 | 0.0079 | 0.0105 | 0.0134 | 0.016 | 0.0186 | 0.0222 | 0.0241 | 0.0269 | 0.0295 |
| B | 0.004 | 0.0064 | 0.0086 | 0.0111 | 0.0132 | 0.0164 | 0.018 | 0.0203 | 0.0229 | 0.0254 |
| C | 0.0155 | 0.0241 | 0.0321 | 0.0388 | 0.0486 | 0.0549 | 0.0651 | 0.0711 | 0.0784 | 0.084 |
| D | 0.0635 | 0.0927 | 0.1261 | 0.1566 | 0.1875 | 0.2217 | 0.259 | 0.2897 | 0.316 | 0.3495 |

### 5.3 Algorytm górna-dolna

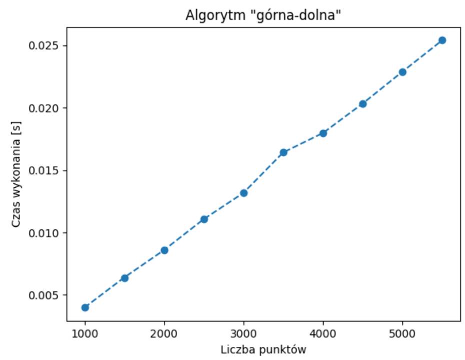
W tabeli 2 przedstawiamy czasy uzyskane przez algorytm górna-dolna dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 5, 6, 7, 8 widzimy ilustrację danych z tabeli 2.

### 5.4 Porównanie z przewidywaniami teoretycznymi

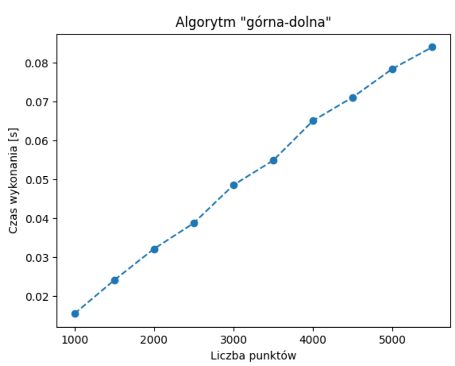
Kształt dla każdego z typu zbiorów dla algorytmu górna-dolna przypomina wykres liniowy.Jest bardzo szybki(szybsze są tylko quickhull i chan) na tle Grahama wykres złożoności tego algorytm przypomina raczej stałą. Jednak, gdy wyizolować sam wykres algorytmu góra dół, to widzimy, że także i on przypomina wykres liniowy Zgdza się to z przewidywaniami teoretycznymi dla tegoż algorytmu - ma on złożoność O(n logn). Wykres o takiej złożoności taka bardzo przypomina wykres liniowy. Ilość wierzchołków w otoczce nie ma znaczenia czas wykonania algorytmu góra-dół zależy tylko od mocy zbioru punktów, dla którego otoczki szukamy.



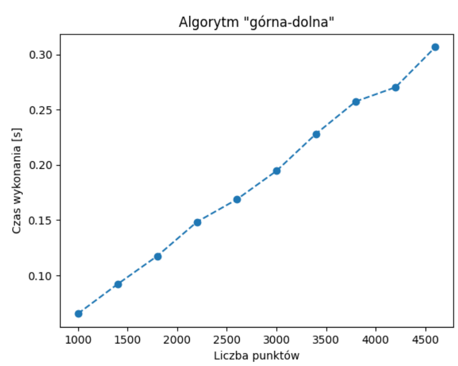

Rysunek 5: Zbiór typu A, algorytm górna-dolna




Rysunek 6: Zbiór typu B, algorytm górna-dolna




Rysunek 7: Zbiór typu C, algorytm górna-dolna




Rysunek 8: Zbiór typu D, algorytm górna-dolna

Tablica 3: Czas wykonania algorytmu Chana w zależności od typu zbioru testowego oraz mocy zbioru punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Typ zbioru | Czas wykonania [s] | | | | | | | | | |
| A | 0.1055 | 0.1544 | 0.3575 | 0.2408 | 0.5449 | 0.6155 | 0.7104 | 0.7904 | 0.8621 | 0.9826 |
| B | 0.3714 | 0.5483 | 0.6962 | 0.8893 | 1.0799 | 1.2537 | 1.415 | 1.827 | 1.7746 | 1.9095 |
| C | 0.2457 | 0.5615 | 0.727 | 0.9188 | 1.1007 | 1.3044 | 1.5073 | 1.6875 | 1.88 | 2.1234 |
| D | 0.6322 | 0.9372 | 1.2306 | 2.1543 | 2.8556 | 6.0844 | 2.4157 | 3.943 | 3.094 | 4.7075 |

## 5.5 Algorytm Chana

W tabeli 3 przedstawiamy czasy uzyskane przez algorytm Chana dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 9, 10, 11, 12 widzimy ilustrację danych z tabeli 3.

## 5.6 Porównanie z przewidywaniami teoretycznymi

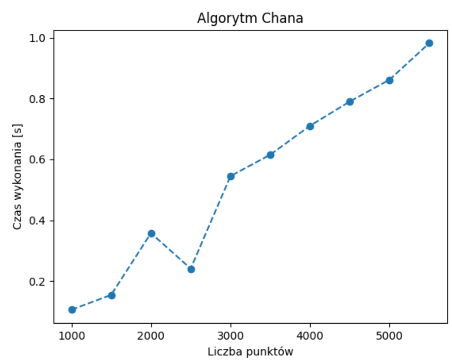
Chan spośród wszystkich rozważanych algorytmów wypadł najgorzej względem reszty. Teorytyczna złożoność chana wynosi O(nlogk), gdzie k jest ilością wierzchołków w otoczce. Patrząc na wykresy czasu dla Chana ilość wierzchołków w otoczce faktycznie wpływa na szybkość jego działania dla chmury punktów działał bardzo nieregularnie, natomiast dla pozostałych rozkładów, o w stałej ilości wierzchołków w otoczce względem wierzchołków zbioru w algorytmie chana dominała natura liniowa. Wygląda na to, że przewidywania teoretyczne po części sprawdzająsię dla chana, jego wykres ma kształt w przybliżeniu liniowy, a dla okręgu, gdzie jest najwięcej wierzchołków w otoczce jest najbardziej stromy. Jednakże poza tym, że ma najlepszą złożoność asymptotyczną, to niestety ta implementacja ma największy współczynnik stały w porównaniu z innymi algorytmami i dlatego wypada najgorzej. Jedyny wynik porównywalny z innymi uzyskał w zbiorze typu D, gdzie ma najlepsze warunki - są tylko 4 wierzchołki w otoczce.

## 5.7 Algorytm QuickHull

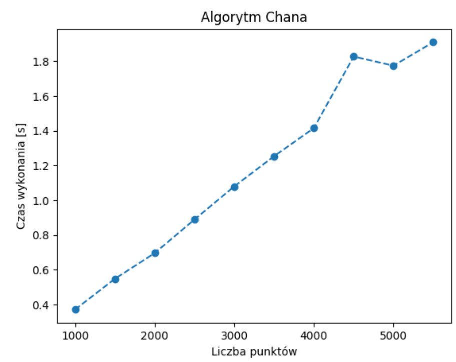
W tabeli 4 przedstawiamy czasy uzyskane przez algorytm QuickHull dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 13, 14, 15, 16 widzimy ilustrację danych z tabeli 4.

## 5.8 Porównanie z przewidywaniami teoretycznymi

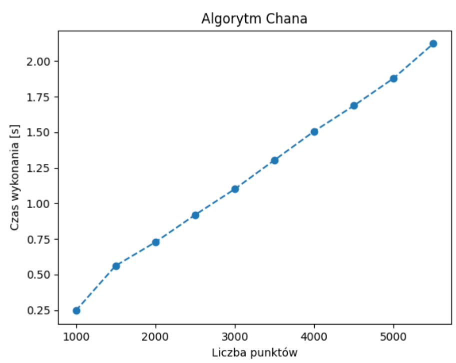
Algorytm QuickHull z wszystkich wypadła najlepiej. Na tle innych wykresów wygląda jakby miał asymptotyczność stałą. Po zobaczeniu jednak jego wykresu osobno widać, że jest to wykres w przybliżeniu liniowy(z wyglądu), co się zgadza z przewidywaniami teoretycznymi. Ma on jednak najmniejszy stały współczynnik względem pozostałych algorytmów, co odpowiada jego zdecydowanie większej prostoty względem innych. ///



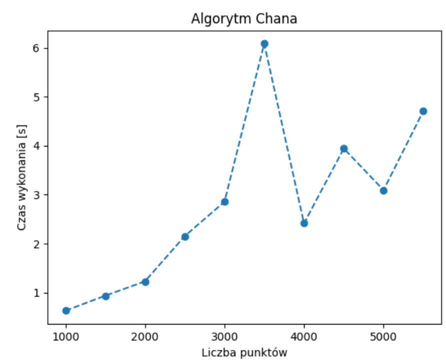

Rysunek 9: Zbiór typu A, algorytm Chana




Rysunek 10: Zbiór typu B, algorytm Chana



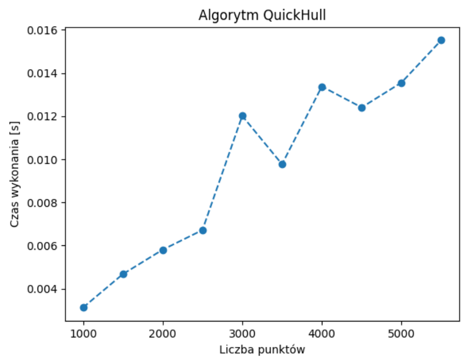

Rysunek 11: Zbiór typu C, algorytm Chana



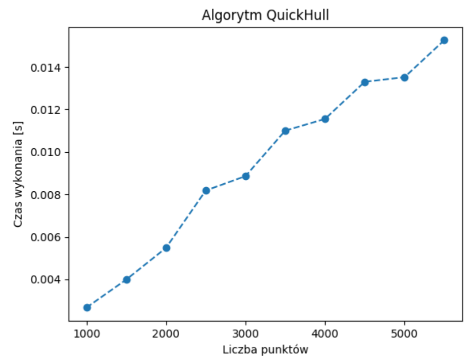

Rysunek 12: Zbiór typu D, algorytm Chana

Tablica 4: Czas wykonania algorytmu QuickHull w zależności od typu zbioru testowego oraz mocy zbioru punktów.

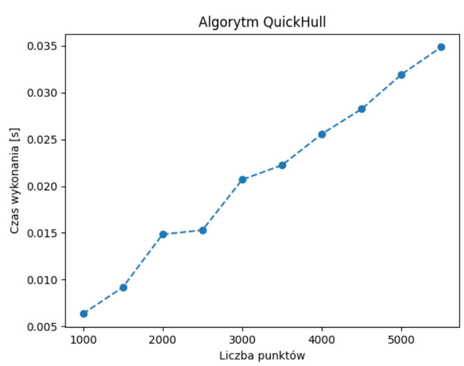
| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Typ zbioru | Czas wykonania [s] | | | | | | | | | |
| A | 0.0031 | 0.0047 | 0.0058 | 0.0067 | 0.012 | 0.0098 | 0.0134 | 0.0124 | 0.0136 | 0.0155 |
| B | 0.0027 | 0.004 | 0.0055 | 0.0082 | 0.0089 | 0.011 | 0.0116 | 0.0133 | 0.0135 | 0.0153 |
| C | 0.0064 | 0.0092 | 0.0148 | 0.0153 | 0.0207 | 0.0222 | 0.0256 | 0.0282 | 0.0319 | 0.0348 |
| D | 0.0287 | 0.0426 | 0.0551 | 0.0702 | 0.0847 | 0.0986 | 0.1143 | 0.1294 | 0.1411 | 0.1562 |



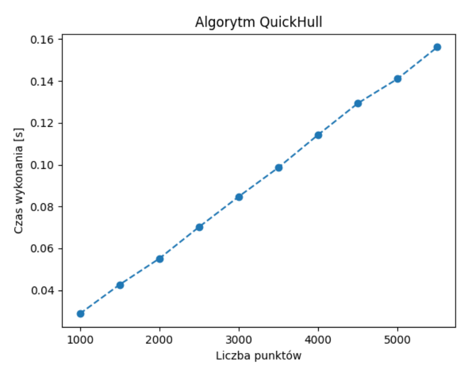

Rysunek 13: Zbiór typu A, algorytm QuickHull




Rysunek 14: Zbiór typu B, algorytm QuickHull



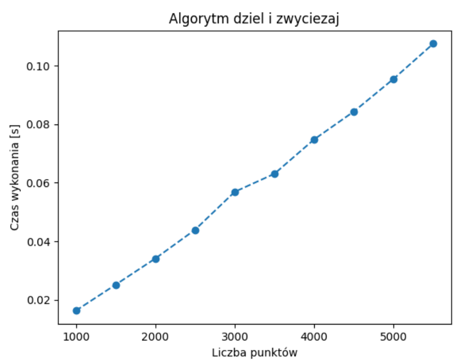

Rysunek 15: Zbiór typu C, algorytm QuickHull



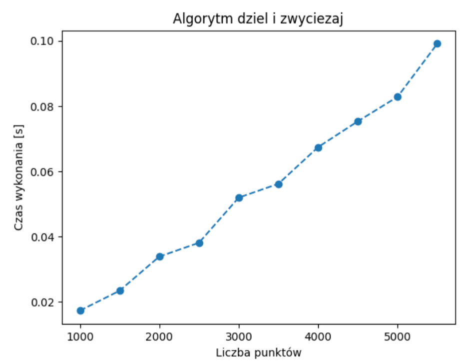

Rysunek 16: Zbiór typu D, algorytm QuickHull

Tablica 5: Czas wykonania algorytmu dziel i zwyciężaj w zależności od typu zbioru testowego oraz mocy zbioru punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Typ zbioru | Czas wykonania [s] | | | | | | | | | |
| A | 0.0163 | 0.0251 | 0.0341 | 0.0439 | 0.0569 | 0.0631 | 0.0748 | 0.0844 | 0.0955 | 0.1076 |
| B | 0.0173 | 0.0234 | 0.0339 | 0.0381 | 0.052 | 0.0562 | 0.0675 | 0.0754 | 0.0829 | 0.0992 |
| C | 0.0441 | 0.0683 | 0.0877 | 0.1189 | 0.1516 | 0.1863 | 0.2067 | 0.2437 | 0.27 | 0.3132 |
| B | 0.1634 | 0.2474 | 0.3518 | 0.4703 | 0.5694 | 0.6705 | 0.7849 | 0.8928 | 1.0458 | 1.1565 |




Rysunek 17: Zbiór typu A, algorytm dziel i zwyciężaj




Rysunek 18: Zbiór typu B, algorytm dziel i zwyciężaj

## 5.9 Algorytm dziel i zwyciężaj

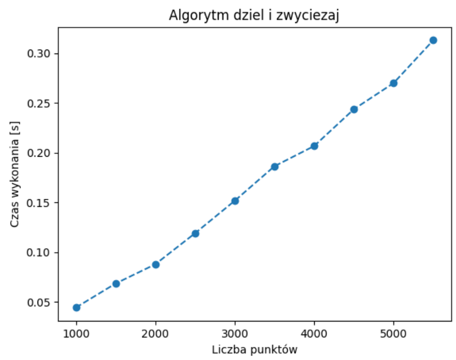
W tabeli 5 przedstawiamy czasy uzyskane przez algorytm Dziel i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 17, 18, 19, 20 widzimy ilustrację danych z tabeli 5.

## 5.10 Porównanie z przewidywaniami teoretycznymi

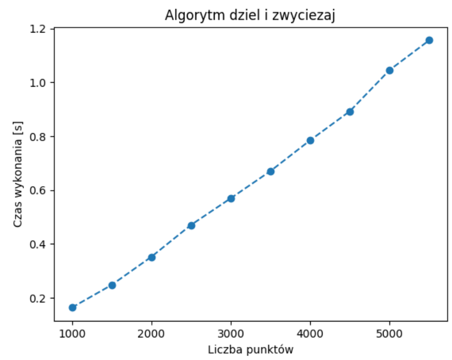
Algorytm dziel i zwyciężaj wypadł na drugim miejscu zaraz po grahamie. Jego wykres odpowiada przewidywaniom teoretycznym, jest to wykres w przybliżeniu liniowy bardzo powoli skłaniający się ku górze. Niezależnie od rozkładu punktów algorytm ten działa w zależności od ich rozmiarów.

## 5.11 Algorytm przyrostowy

W tabeli 6 przedstawiamy czasy uzyskane przez algorytm przyrostowy i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 21, 22, 23, 24 widzimy ilustrację danych z tabeli 6.




Rysunek 19: Zbiór typu C, algorytm dziel i zwyciężaj




Rysunek 20: Zbiór typu D, algorytm dziel i zwyciężaj

Tablica 6: Czas wykonania algorytmu przyrostowego w zależności od typu zbioru testowego oraz mocy zbioru punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Typ zbioru | Czas wykonania [s] | | | | | | | | | |
| A | 0.0125 | 0.0194 | 0.0251 | 0.0305 | 0.0377 | 0.0431 | 0.0497 | 0.0584 | 0.0621 | 0.0705 |
| B | 0.0864 | 0.1788 | 0.2978 | 0.4428 | 0.6359 | 0.7745 | 0.9738 | 1.1858 | 1.3526 | 1.5613 |
| C | 0.0315 | 0.0475 | 0.063 | 0.0793 | 0.0997 | 0.1136 | 0.1292 | 0.1449 | 0.1555 | 0.1742 |
| D | 0.1373 | 0.1992 | 0.2715 | 0.326 | 0.4165 | 0.4639 | 0.5195 | 0.5937 | 0.6773 | 0.7446 |

### 5.12 Porównanie z przewidywaniami teoretycznymi

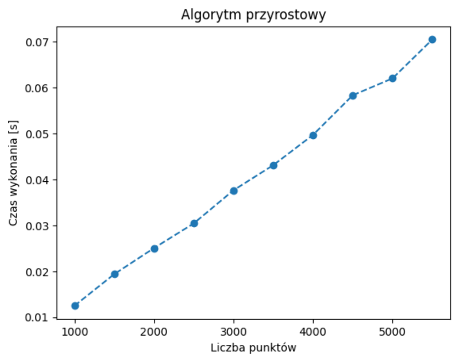
Algorytm przyrostowy wypadł odrobinę lepiej od dziel i zwyciężaj. Jego wykres odpowiada przewidywaniom teoretycznym, jest to wykres w przybliżeniu liniowy bardzo powoli skłaniający się ku górze. Niezależnie od rozkładu punktów algorytm ten działa w zależności od ich rozmiarów.

### 5.13 Algorytm Jarvisa

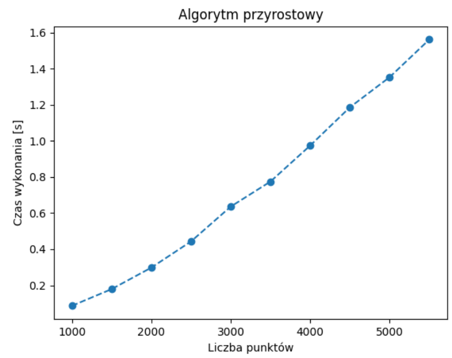
W tabeli 7 przedstawiamy czasy uzyskane przez algorytm Jarvisa i zwyciężaj dla kolejnych zbiorów punktów, przy różnej liczebności punktów w zbiorze. Na wykresach 25, 26, 27, 28 widzimy ilustrację danych z tabeli 7.

### 5.14 Porównanie z przewidywaniami teoretycznymi

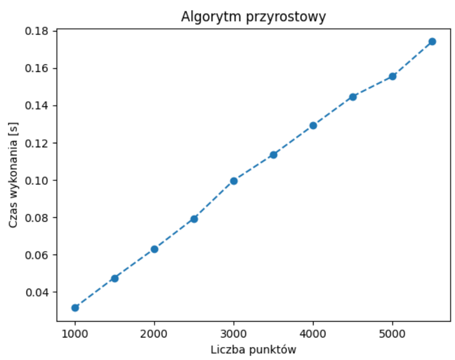
Algorytm Jarvisa zachowywał się zgodnie z przewidywaniami teoretycznymi. Jego złożoność wynosi O(kn), gdzie k to liczba wierzchołków w otoczce. Dla chmury punktów, gdszie liczba wierzchołków w otoczce jest losowa zachowywał się nieregularnie. Dla pozostałych zbiorów: dla rozkładu kołowego ukwadratowił się i miał zdecydowanie najgorsze



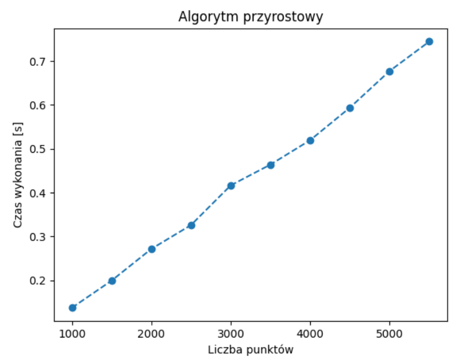

Rysunek 21: Zbiór typu A, algorytm przyrostowy




Rysunek 22: Zbiór typu B, algorytm przyrostowy



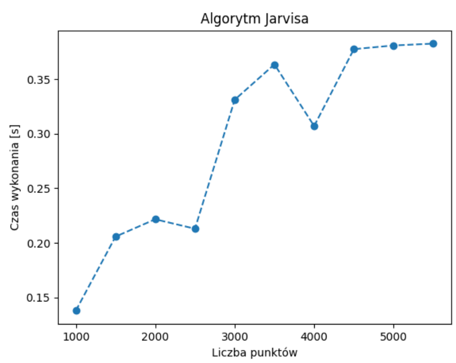

Rysunek 23: Zbiór typu C, algorytm przyrostowy



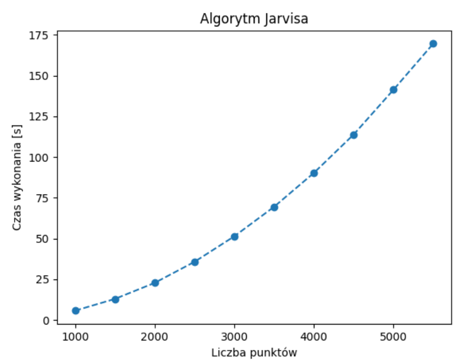

Rysunek 24: Zbiór typu D, algorytm przyrostowy

Tablica 7: Czas wykonania algorytmu Jarvisa w zależności od typu zbioru testowego oraz mocy zbioru punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Typ danych | Czas wykonania [s] | | | | | | | | | |
| A | 0.1379 | 0.2058 | 0.2217 | 0.213 | 0.3311 | 0.3635 | 0.307 | 0.3775 | 0.3808 | 0.3825 |
| B | 5.684 | 12.882 | 22.737 | 35.648 | 51.305 | 69.425 | 90.227 | 113.80 | 141.27 | 169.58 |
| C | 0.0529 | 0.077 | 0.1046 | 0.1306 | 0.1543 | 0.184 | 0.2086 | 0.2304 | 0.2571 | 0.2858 |
| D | 0.1094 | 0.1572 | 0.2167 | 0.2611 | 0.3129 | 0.3711 | 0.4218 | 0.4774 | 0.5255 | 0.5886 |




Rysunek 25: Zbiór typu A, algorytm Jarvisa




Rysunek 26: Zbiór typu B, algorytm Jarvisa

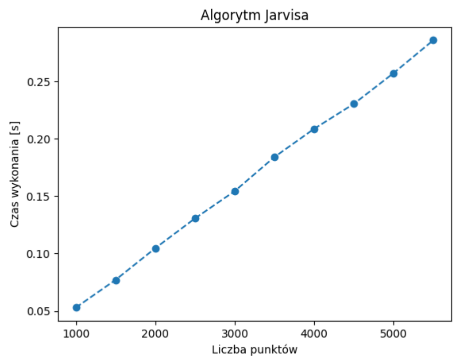
wyniki. Dla zbioru o rozmiarze 5500 wykonywał się niemal 3 min. Jednakże dla pozostałych zbiorów, jego złożoność stanowiła przewagę mógł śmiało konkurować z pozostałymi algorytmami O(nlogn) i odznaczał się podobnymi czasami.

# 6 Porównanie algorytmów

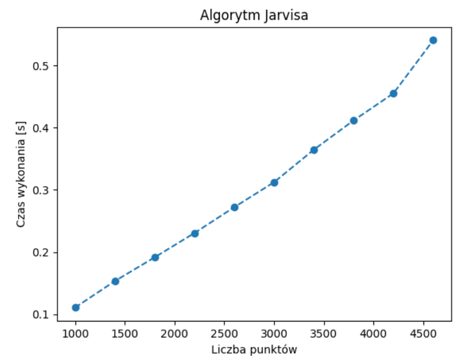
Ze względu na różną charakterystykę zbiorów punktów i zachowania algorytmów, porównania dzielimy na 4 sekcje, każda odpowiadająca wybranemu typowi zbioru punktów testowych.

Dla uproszczenia zapisu w tabelach wprowadzamy oznaczenia dla poszczególnych algorytmów:

- $GR$ – algorytm Grahama
- $LU$ – algorytm górna dolna
- $CH$ – algorytm Chana
- $QH$ – algorytm QuickHull




Rysunek 27: Zbiór typu C, algorytm Jarvisa




Rysunek 28: Zbiór typu D, algorytm Jarvisa

Tablica 8: Czasy wykonania poszczególnych algorytmów dla zbioru typu A przy różnych mocach zbiorów punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Algorytm | Czas wykonania [s] | | | | | | | | | |
| $GR$ | 0.0239 | 0.0331 | 0.045 | 0.0586 | 0.072 | 0.0847 | 0.0983 | 0.1122 | 0.1273 | 0.141 |
| $LU$ | 0.0076 | 0.0079 | 0.0105 | 0.0134 | 0.016 | 0.0186 | 0.0222 | 0.0241 | 0.0269 | 0.0295 |
| $CH$ | 0.1055 | 0.1544 | 0.3575 | 0.2408 | 0.5449 | 0.6155 | 0.7104 | 0.7904 | 0.8621 | 0.9826 |
| $QH$ | 0.0031 | 0.0047 | 0.0058 | 0.0067 | 0.012 | 0.0098 | 0.0134 | 0.0124 | 0.0136 | 0.0155 |
| $DC$ | 0.0163 | 0.0251 | 0.0341 | 0.0439 | 0.0569 | 0.0631 | 0.0748 | 0.0844 | 0.0955 | 0.1076 |
| $IN$ | 0.0125 | 0.0194 | 0.0251 | 0.0305 | 0.0377 | 0.0431 | 0.0497 | 0.0584 | 0.0621 | 0.0705 |
| $JR$ | 0.1379 | 0.2058 | 0.2217 | 0.213 | 0.3311 | 0.3635 | 0.307 | 0.3775 | 0.3808 | 0.3825 |

- $DC$ – algorytm dziel i zwyciężaj
- $IN$ – algorytm przyrostowy
- $JR$ – algorytm Jarvisa

### 6.1 Zbiór typu A

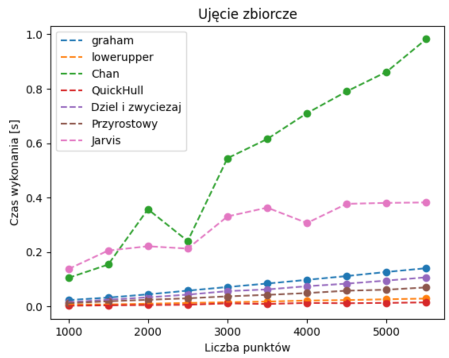
### 6.2 Zbiór typu B

### 6.3 Zbiór typu C

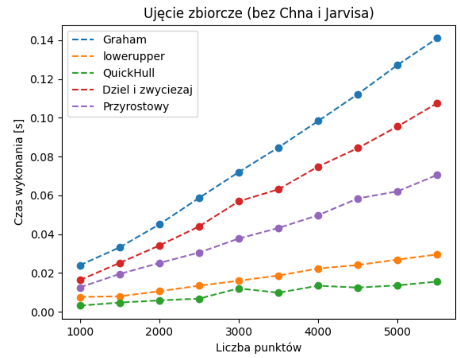
### 6.4 Zbiór typu D

## 7 Bibliografia


1. *Wykład z przedmiotu Algorytmy Geometryczne, Informatyka 3. sem., 1. st. AGH, Barbara Głut*



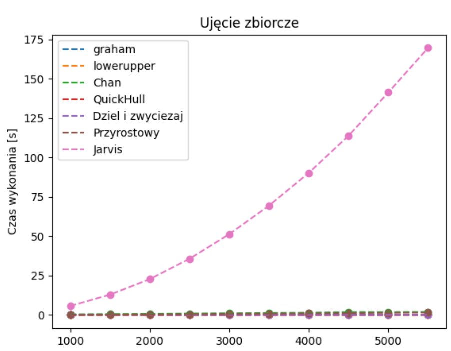

Rysunek 29: Zbiór typu A, zestawienie



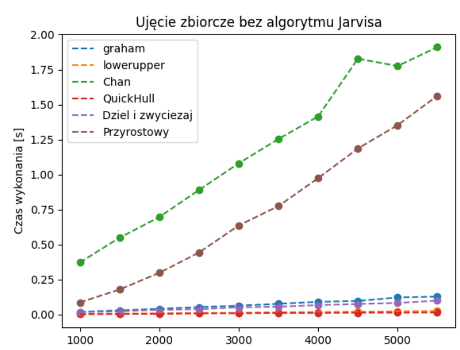

Rysunek 30: Zbiór typu A, zestawienie bez Jarvisa i Chana

Tablica 9: Czasy wykonania poszczególnych algorytmów dla zbioru typu B przy różnych mocach zbiorów punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Algorytm | Czas wykonania [s] | | | | | | | | | |
| $GR$ | 0.0191 | 0.0299 | 0.0413 | 0.0531 | 0.0637 | 0.0768 | 0.0908 | 0.0973 | 0.122 | 0.1284 |
| $LU$ | 0.004 | 0.0064 | 0.0086 | 0.0111 | 0.0132 | 0.0164 | 0.018 | 0.0203 | 0.0229 | 0.0254 |
| $CH$ | 0.3714 | 0.5483 | 0.6962 | 0.8893 | 1.0799 | 1.2537 | 1.415 | 1.827 | 1.7746 | 1.9095 |
| $QH$ | 0.0027 | 0.004 | 0.0055 | 0.0082 | 0.0089 | 0.011 | 0.0116 | 0.0133 | 0.0135 | 0.0153 |
| $DC$ | 0.0173 | 0.0234 | 0.0339 | 0.0381 | 0.052 | 0.0562 | 0.0675 | 0.0754 | 0.0829 | 0.0992 |
| $IN$ | 0.0864 | 0.1788 | 0.2978 | 0.4428 | 0.6359 | 0.7745 | 0.9738 | 1.1858 | 1.3526 | 1.5613 |
| $JR$ | 5.68 | 12.88 | 22.73 | 35.64 | 51.30 | 69.42 | 90.22 | 113.80 | 141.27 | 169.58 |



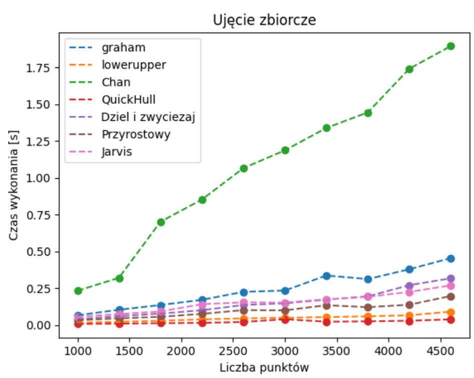

Rysunek 31: Zbiór typu B, zestawienie



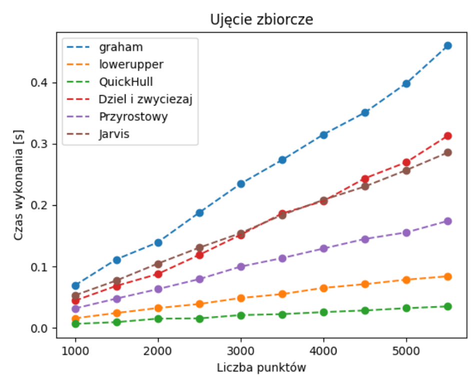

Rysunek 32: Zbiór typu B, zestawienie bez Jarvisa

Tablica 10: Czasy wykonania poszczególnych algorytmów dla zbioru typu C przy różnych mocach zbiorów punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Algorytm | Czas wykonania [s] | | | | | | | | | |
| $GR$ | 0.069 | 0.1115 | 0.1392 | 0.1878 | 0.2349 | 0.2736 | 0.3153 | 0.351 | 0.3988 | 0.4601 |
| $LU$ | 0.0155 | 0.0241 | 0.0321 | 0.0388 | 0.0486 | 0.0549 | 0.0651 | 0.0711 | 0.0784 | 0.084 |
| $CH$ | 0.2457 | 0.5615 | 0.727 | 0.9188 | 1.1007 | 1.3044 | 1.5073 | 1.6875 | 1.88 | 2.1234 |
| $QH$ | 0.0064 | 0.0092 | 0.0148 | 0.0153 | 0.0207 | 0.0222 | 0.0256 | 0.0282 | 0.0319 | 0.0348 |
| $DC$ | 0.0441 | 0.0683 | 0.0877 | 0.1189 | 0.1516 | 0.1863 | 0.2067 | 0.2437 | 0.27 | 0.3132 |
| $IN$ | 0.0315 | 0.0475 | 0.063 | 0.0793 | 0.0997 | 0.1136 | 0.1292 | 0.1449 | 0.1555 | 0.1742 |
| $JR$ | 0.0529 | 0.077 | 0.1046 | 0.1306 | 0.1543 | 0.184 | 0.2086 | 0.2304 | 0.2571 | 0.2858 |



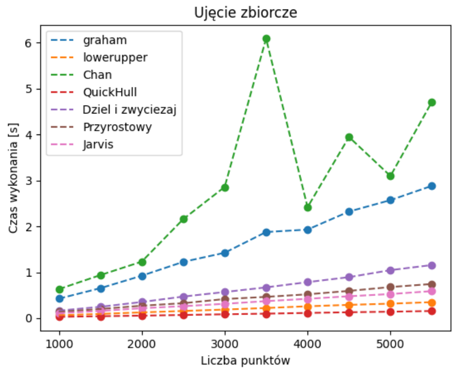

Rysunek 33: Zbiór typu C, zestawienie



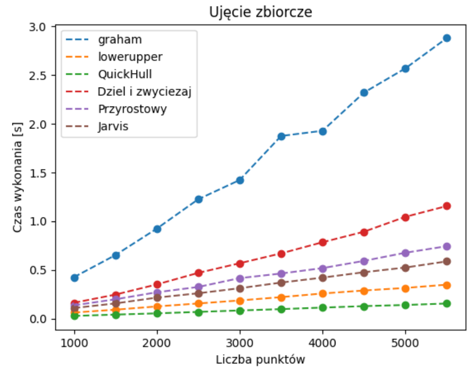

Rysunek 34: Zbiór typu C, zestawienie bez Chana

Tablica 11: Czasy wykonania poszczególnych algorytmów dla zbioru typu D przy różnych mocach zbiorów punktów.

| | Liczba punktów | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1000 | 1500 | 2000 | 2500 | 3000 | 3500 | 4000 | 4500 | 5000 | 5500 |
| Algorytm | Czas wykonania [s] | | | | | | | | | |
| $GR$ | 0.4256 | 0.6523 | 0.924 | 1.2255 | 1.4233 | 1.8752 | 1.9285 | 2.3207 | 2.5692 | 2.88 |
| $LU$ | 0.0635 | 0.0927 | 0.1261 | 0.1566 | 0.1875 | 0.2217 | 0.259 | 0.2897 | 0.316 | 0.3495 |
| $CH$ | 0.6322 | 0.9372 | 1.2306 | 2.1543 | 2.8556 | 6.0844 | 2.4157 | 3.943 | 3.094 | 4.7075 |
| $QH$ | 0.0287 | 0.0426 | 0.0551 | 0.0702 | 0.0847 | 0.0986 | 0.1143 | 0.1294 | 0.1411 | 0.1562 |
| $DC$ | 0.1634 | 0.2474 | 0.3518 | 0.4703 | 0.5694 | 0.6705 | 0.7849 | 0.8928 | 1.0458 | 1.1565 |
| $IN$ | 0.1373 | 0.1992 | 0.2715 | 0.326 | 0.4165 | 0.4639 | 0.5195 | 0.5937 | 0.6773 | 0.7446 |
| $JR$ | 0.1094 | 0.1572 | 0.2167 | 0.2611 | 0.3129 | 0.3711 | 0.4218 | 0.4774 | 0.5255 | 0.5886 |




Rysunek 35: Zbiór typu D, zestawienie




Rysunek 36: Zbiór typu D, zestawienie bez Chana


2. *Computational Geometry – Algorithms and Applications, Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars*

3. `https://jeffe.cs.illinois.edu/teaching/compgeom/notes/01-convexhull.pdf`