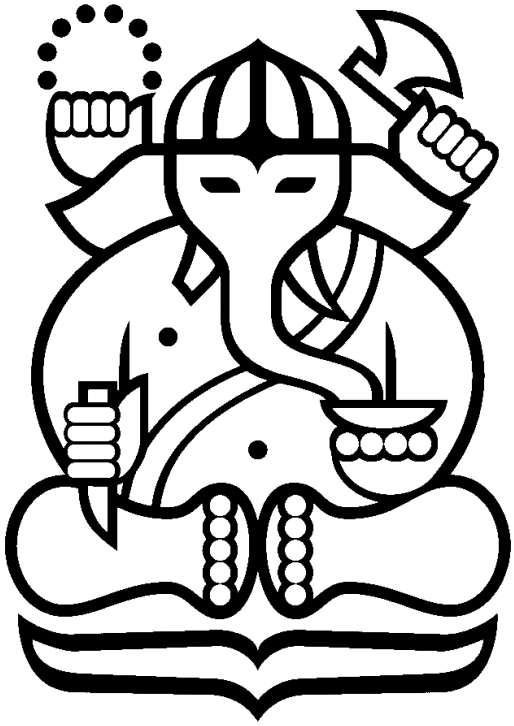
IF2211 Strategi Algoritma

IMPLEMENTASI ALGORITMA DIVIDE AND CONQUER DALAM MENCARI PASANGAN TITIK TERDEKAT DI RUANG EUCLIDEAN

Laporan Tugas Kecil I

Disusun untuk memenuhi tugas mata kuliah IF2211 Strategi Algoritma pada Semester 2 (dua) Tahun Akademik 2022/2023



Disusun oleh:
Michael Jonathan Halim
Enrique Alifio Ditya
13521124 / 13521142
K02

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

BANDUNG 2023

DAFTAR ISI

BAB I
PENDAHULUAN

Pada laporan ini, akan dijelaskan sebuah solusi permasalahan pencarian dua titik terdekat pada *Euclidean Space* dengan algoritma *Divide and Conquer*. Permasalahan ini merupakan salah satu isu fundamental di bidang geometri komputasi dengan aplikasi dunia nyata yang luas dan beragam, mulai dari grafika komputasi, pemrosesan citra, robotika, dan sistem informasi geografis. Inti dari permasalahan ini dapat dirumuskan sebagai berikut: "Diberikan n-buah titik dalam *Euclidean Space* dan d dimensi, carilah dua titik dengan jarak terdekat".

Algoritma *Divide and Conquer* merupakan teknik yang cukup sering digunakan dalam menyelesaikan permasalahan di bidang komputasi geometri. Secara dasarnya, algoritma ini bekerja dengan mendekomposisi permasalahan menjadi sub-permasalahan yang diselesaikan masing-masing. Proses ini diakhiri dengan penggabungan solusi dari setiap sub-permasalahan untuk mendapatkan solusi general. Algoritma ini memandang penyelesaian sub-permasalahan sebagai suatu set instruksi yang berulang sehingga solusi dapat dicapai dengan pendekatan rekursif.

Masalah pencarian pasangan titik terdekat telah dipelajari secara ekstensif dari masa ke masa, dan beberapa algoritma telah diusulkan untuk menyelesaikannya. Meskipun itu, algoritma *divide and conquer* yang disajikan dalam laporan ini memiliki kompleksitas waktu $O(n\,(log(n))^{d-1})$ dengan d sebagai dimensi, yang merupakan kompleksitas waktu lebih baik daripada brute force. Dalam laporan ini, kami memberikan analisis mendetail mengenai algoritma *divide and conquer* serta membandingkannya dengan algoritma lain yang menggunakan pendekatan *brute force*.

BAB II
LANDASAN TEORI

Pencarian dua buah titik terdekat dalam ruang Euclidean dengan algoritma *divide and conquer* didasarkan pada prinsip dasar bahwa jarak terdekat antara dua buah titik dapat dicari dengan membagi wilayah yang memuat kedua titik menjadi beberapa sub-wilayah yang lebih kecil. Setiap sub-wilayah ini kemudian diperiksa untuk menemukan jarak terdekat antara dua titik yang mungkin terletak di dalamnya. Jarak antara dua titik pada ruang Euclidean berdimensi 'n' didefinisikan sebagai berikut.

$$d = \sqrt{\sum_{i}^{n} (p_{1_i} - p_{2_i})^2}$$

Dengan p_1 sebagai titik pertama, p_2 sebagai titik kedua, dan i sebagai indeks sumbu koordinat dari titik.

Metode *divide and conquer* digunakan untuk membagi wilayah menjadi sub-wilayah yang lebih kecil. Pada setiap tahap, wilayah dibagi menjadi dua bagian sejajar dengan salah satu sumbu koordinat. Kemudian, pencarian jarak terdekat dilakukan secara rekursif pada kedua bagian ini. Setelah pencarian pada kedua sub-wilayah selesai, jarak terdekat dari dua titik yang berbeda di seluruh wilayah dipilih.

Metode ini menghasilkan kompleksitas waktu yang efisien untuk mencari dua titik terdekat di dalam wilayah dengan ukuran besar. Dengan melakukan pembagian wilayah secara rekursif, kompleksitas waktu dapat dijaga agar menjadi $O(n\,(log(n))^{d-1})$, dengan n sebagai jumlah titik yang harus diperiksa dan d sebagai dimensi dari ruang Euclidean yang ditelusuri.

BAB III
ALGORITMA

3.1 Penjelasan Algoritma *Brute Force*

Algoritma *brute force* adalah metode yang digunakan untuk menemukan solusi dengan mencoba semua kemungkinan yang ada. Dalam hal ini, algoritma *brute forc*e akan menghitung jarak dari seluruh pasangan titik yang ada. Secara garis besar, algoritma *brute force* berjalan seperti berikut:

1. Pilih suatu titik sembarang untuk memulai pencarian dan inisialisasi nilai jarak terdekat
2. Cari titik lain dan periksa apabila pasangan titik belum pernah dicek sebelumnya dan memiliki jarak lebih kecil dari nilai jarak terdekat yang diketahui. Jika iya, perbarui nilai jarak terdekat
3. Lakukan untuk setiap titik pada ruang pencarian

Prosedur pencarian ini cukup simpel namun memakan waktu yang lama untuk jumlah titik yang banyak. Kompleksitas waktu dari algoritma ini adalah $O(n^2)$.

3.2 Penjelasan Algoritma *Divide and Conquer*

Sebagaimana dijelaskan sebelumnya, algoritma *divide and conquer* membagi permasalahan menjadi subset-subset kecil dengan solusi masing-masing yang kemudian digabungkan menjadi solusi umum. Dalam implementasinya untuk mencari pasangan titik terdekat, algoritma *divide and conquer* mengikuti tahapan-tahapan berikut:

1. Urutkan titik-titik berdasarkan absis terurut menaik dengan *quick sort*.
2. Bagi himpunan titik menjadi subset kiri dan kanan berdasarkan koordinatnya terhadap suatu garis/bidang maya pembagi, umumnya terletak pada absis median
3. Terdapat tiga kasus terletaknya pasangan titik terdekat: Keduanya terletak pada subset kiri, keduanya terletak pada subset kanan, atau kedua titik terpisah oleh garis/bidang pembagi.
4. Untuk menghadapi kasus ketiga, manfaatkan fakta bahwa jarak titik ke garis pembagi tidaklah lebih besar dari *d*, dengan *d* merupakan jarak terkecil dari subset kanan dan kiri. Identifikasikan titik-titik tersebut yang terletak di dalam daerah garis pembagi dan rentangan *d*. Jika terdapat pasangan titik yang jaraknya lebih kecil dari *d*, maka untuk setiap selisih koordinat kedua titik, tidaklah lebih besar dari *d*.
5. Lakukan secara rekursif untuk subset kanan dan kiri.
6. Bandingkan antara ketiga kasus tersebut untuk mendapatkan pasangan titik terdekat.

Algoritma *divide and conquer* ini merupakan yang paling sangkil dalam menyelesaikan permasalahan pencarian pasangan titik terdekat, dengan kompleksitas waktu $O(n\,(log(n))^{d-1})$. Analisis kompleksitas lebih lanjut akan dibahas pada bagian V laporan ini.

BAB IV
PENJELASAN KODE PROGRAM

4.1 Deskripsi Umum *Source* Program

Pada laporan ini, program ditulis dalam bahasa Python. Bahasa ini dipilih dengan pertimbangan kemudahan visualisasi hasil akhir dengan menggunakan *library* yang tersedia. *Source code* dari program ini dibagi menjadi tiga bagian, yakni modul *brute force, divide and conquer,* serta program utama. Modul *brute force* sebagaimana namanya, mengandung implementasi algoritma *brute force* dalam pencarian pasangan titik terdekat. Modul *divide and conquer* pada sisi lain mengandung algoritma pengurutan serta fungsi-fungsi penyelesaian permasalahan *closest pair*. Terakhir, program utama melakukan kompilasi dari seluruh fungsi dan disajikan dalam bentuk *Graphical User Interface* (GUI) dengan bantuan library tkinter. Saat waktu pengoperasian, program ini akan menerima input berupa jumlah titik dan orde dimensi dari ruang pencarian.

4.2 Struktur Data

Untuk mensimulasikan sebuah titik dalam ruang Euclidean, kami membuat kelas bernama Point dengan atribut koordinat berupa *list* dari *integer* yang merepresentasikan posisi pada orde ke-n. Selain itu, didefinisikan pula beberapa *method* dan operator tambahan untuk mempermudah manipulasi data dari objek Point. Kelas Point ini didefinisikan dalam file ‘Point.py’ sebagai berikut.

```
import math

class Point:
    """
    Represents a point in R^n space.

    Attributes
    ----------
        coor (tuple): A tuple of n coordinates that define the
position of the point in R^n space.

    Methods
```

```
    ----------
        __init__(*coor): Initializes a new Point object with the
given coordinates.
        __repr__(): Returns a string representation of the Point
object.
        __len__(): Returns the number of coordinates in the Point
object.
        distance_to(other): Returns the Euclidean distance between
the Point object and the other Point object.

    Operators
    ----------
        __getitem__(index): `[]` Returns the coordinate at the given
index in the Point object.
        __eq__(other): `=` Returns True if the Point object is equal
to the other Point object.
        __add__(other): `+` Returns a new Point object that is the
sum of the Point object and the other Point object.
        __sub__(other): `-` Returns a new Point object that is the
difference between the Point object and the other Point object.
        __mul__(other): `*` Returns the dot product of the Point
object and the other Point object, or a new Point object that is the
scalar product of the Point object and the scalar other.
        __pow__(other): `**` Returns a new Point object that is the
element-wise power of the Point object to the power of the scalar
other.
    """

    def __init__(self, *coor):
        self.coor = coor

    def __repr__(self):
```

```
        return f"Point({', '.join(str(c) for c in self.coor)})"

    def __len__(self):
        return len(self.coor)

    def __getitem__(self, index):
        return self.coor[index]

    def __eq__(self, other):
        if not isinstance(other, Point):
            return False
        return self.coor == other.coor

    def __add__(self, other):
        if not isinstance(other, Point):
            raise TypeError("unsupported operand type(s) for +:
'Point' and '{type(other).__name__}'")
        if len(self) != len(other):
            raise ValueError("Points must have the same dimension to
add them")
        return Point(*[self[i] + other[i] for i in range(len(self))])

    def __sub__(self, other):
        if not isinstance(other, Point):
            raise TypeError("unsupported operand type(s) for -:
'Point' and '{type(other).__name__}'")
        if len(self) != len(other):
            raise ValueError("Points must have the same dimension to
subtract them")
        return Point(*[self[i] - other[i] for i in range(len(self))])

    def distance_to(self, other):
```

```
        if not isinstance(other, Point):
            raise TypeError("unsupported operand type(s) for
distance_to: 'Point' and '{type(other).__name__}'")
        if len(self) != len(other):
            raise ValueError("Points must have the same dimension to
calculate distance between them")
        return math.sqrt(sum((a - b) ** 2 for a, b in zip(self.coor,
other.coor)))

    def __mul__(self, other):
        if isinstance(other, (int, float)):
            return Point(*[self[i] * other for i in
range(len(self))])
        elif isinstance(other, Point):
            if len(self) != len(other):
                raise ValueError("Points must have the same dimension
to dot multiply them")
            return sum([self[i] * other[i] for i in
range(len(self))])
        else:
            raise TypeError(f"unsupported operand type(s) for *:
'Point' and '{type(other).__name__}'")

    def __pow__(self, other):
        if isinstance(other, (int, float)):
            return Point(*[self[i] ** other for i in
range(len(self))])
        else:
            raise TypeError(f"unsupported operand type(s) for **:
'Point' and '{type(other).__name__}'")
```

Selain kelas Point, sebuah struktur data *List of Points* juga dimanfaatkan dalam program ini untuk menyimpan titik-titik dalam sebuah ruang Euclidean. Implementasinya dapat dilihat pada modul-modul penyusun program.

4.3 Algoritma *Brute Force*

Penyelesaian dengan pendekatan algoritma *brute force* terletak pada modul 'bruteforce.py' sebagai berikut.

```
import time

def brute_force_closest_pair(space):
    """
    Finds the closest pair of points in R^n Euclidean space using
brute force algorithm.

    Parameters
    ----------
    space: List of Point objects.

    Returns
    ----------
    pair: a list containing a pair of Point objects with the closest
distance.
    """

    # If the list contains two points or less, by definition contains
the closest pair
    if (len(space) ⩽ 2):
        return space

    # Initialize the list of pair and minimum distance
    min_distance = space[0].distance_to(space[1])
    pair = (space[0], space[1])
```

```
    # Brute force by checking distance to each other for each points
in space
    for i in range (len(space)):
        for j in range (i + 1, len(space)):
            distance = space[i].distance_to(space[j])
            if distance < min_distance:
                min_distance = distance
                pair = (space[i], space[j])

    return pair


def run_brute_force_closest_pair(points, answer):
    """
    Run brute force algorithm to solve the closest pair of points
problem and display solution on GUI

    Args:
        points (list): list of point objects
        answer (string): solution
    """

    # Defining n
    n = len(points)

    # Brute Force Algorithm
    start_time = time.time()
    p1, p2 = brute_force_closest_pair(points)
    end_time = time.time()

    # Output
```

```
    # Closest Pair and Their Distance
    output = "First Point: " + str(p1) + "\n" + "Second Point: " +
str(p2) + "\n" + f"Minimum Distance: {p1.distance_to(p2)}\n" +
"Euclidean Calculations Done: " + str(int((n-1) * n / 2)) + "\n" +
f"Execution Time: {(end_time - start_time) * 1000} ms\n"
    answer.set(output)

    # Print solution in terminal
    print("Solution by brute force :")
    print(output)
```

Pada modul ini, terdapat dua fungsi yakni `brute_force_closest_pair(space)` dan `run_brute_force_closest_pair(points, answer)`. Fungsi pertama menerima *list of Points* kemudian menggunakan pendekatan *brute force* dengan mengiterasi setiap titik dan menghitung jarak ke titik lainnya untuk mencari pasangan titik terdekat. Fungsi ini kemudian dibungkus dalam fungsi `run` yang akan dipanggil pada driver di program utama.

4.4 Algoritma *Divide and Conquer*

Penyelesaian dengan pendekatan algoritma *divide and conquer* terletak pada modul 'divideconquer.py'. Modul ini terdiri dari beberapa fungsi yang mendukung tahapan-tahapan berbeda dari algoritma *divide and conquer*, yakni fungsi pengurutan (*preprocessing*), fungsi pembagi (*divide*), dan fungsi pencari titik terdekat (*conquer*).

4.4.1 *Preprocessing*

Sebelum melangkah ke tahap pencarian solusi, dilakukan tahap praproses untuk mempermudah *processing* saat *divide and conquer*. Berikut adalah kode dan penjelasan dari tahap *preprocessing*.

```
def partition(points, low, high):
    """
    Divides a list of points by their x-axis.

    Parameters
```

```
    ----------
    points: a list of Point objects.
    low: Lower index of the partitioned list.
    high: Upper index of the partitioned list.

    Returns
    ----------
    i: Corrected index of pivot.
    """

    # Element to be placed in the correct position
    pivot = points[high]

    # Initialize the lower iterator
    i = low - 1

    # Loop the higher iterator through the list
    for j in range(low, high):

        # If element on the higher iterator index is smaller than
pivot, put it on the 'left' of the list
        if (points[j][0] ⩽ pivot[0]):
            # Increment lower iterator
            i += 1
            # Swap the point of smaller value to pivot with the lower
iterator
            points[j], points[i] = points[i], points[j]

    # Swap the pivot on the right position in the list
    points[i+1], points[high] = points[high], points[i+1]

    # Returns the index of pivot
```

```
    return i+1

def quicksort(points, low, high):
    """
    Sorts every points in a list by their x-axis.

    Parameters
    ----------
    points: a list of Point objects.
    low: Lower index of the partitioned list.
    high: Upper index of the partitioned list.

    Returns
    ----------
    points: Points sorted by their x-axis in ascending order.
    """

    if low < high:
        # Get the index of pivot on partition
        idx = partition(points, low, high)

        # Sort the list on the left and right of partition point
        quicksort(points, low, idx-1)
        quicksort(points, idx+1, high)

    return points
```

Untuk mencari titik terdekat dengan algoritma *divide and conquer*, langkah pertama yang dilakukan adalah mengurutkan himpunan titik berdasarkan absis agar himpunan tersebut bisa dibagi menjadi dua. Pengurutan himpunan titik dilakukan dengan algoritma *quick sort*. Algoritma ini dipilih sebab memiliki kompleksitas waktu rata-rata yang paling cepat untuk algoritma pengurutan, yakni $O(n\,log(n))$. Dengan itu, perhitungan batas atas asimtotik kompleksitas waktu pada algoritma utama pencarian pasangan titik terdekat diharapkan tidak

terpengaruhi secara signifikan oleh praproses ini. Proses quicksort ini terbagi dalam dua fungsi, yakni `partition` dan `quicksort`.

Fungsi `partition` menerima tiga argumen: `points`, `low`, dan `high`. Argumen `points` adalah list dari objek-objek Point yang akan diurutkan. Argumen `low` dan `high` merupakan indeks dari bagian-bagian list yang akan diurutkan. Fungsi ini membagi list menjadi dua bagian, yaitu bagian yang elemennya lebih kecil dari pivot dan bagian yang elemennya lebih besar dari pivot. Pivot yang diambil dalam algoritma quicksort ini adalah elemen terakhir dari list. Fungsi ini mengembalikan indeks dari pivot yang telah diposisikan dengan benar setelah pemisahan.

Fungsi `quicksort` menerima tiga argumen: `points`, `low`, dan `high`, sama seperti fungsi `partition` sebelumnya. Fungsi ini melakukan pengurutan dengan memanggil fungsi `partition` untuk membagi list menjadi dua bagian dan memanggil dirinya sendiri secara rekursif untuk mengurutkan kedua bagian tersebut. Fungsi ini mengembalikan list titik-titik yang telah diurutkan.

4.4.2 *Divide*

Setelah mengurutkan himpunan titik tersebut berdasarkan absis, dilakukan tahap *dividing* untuk mengecilkan *scope* permasalahan menjadi subset yang lebih kecil. Berikut adalah kode dan penjelasan dari proses *divide*.

```
def createDivide(points):
    """
    Calculate x coordinate to split a list of points to two list of
points with the same amount

    Args:
        points : a list of Point objects

    Returns:
        x : Coordinate to split on
    """

    # Calculate the middle x coordinate to split the points
    x = (points[len(points) // 2 - 1][0] + points[len(points) //
```

```
2][0]) / 2
    return x

def splitPoints(points):
    """
    Splits points into two sections based off of an axis.

    Args:
        points: a list of Point objects

    Returns:
        left_points : a list of Point objects with coordinate on the
left of split
        right_points : a list of Point objects with coordinate on the
right of split
        x : Coordinate to split on
    """

    # Divide points into left and right zone with each contains half
size of the initial size
    x = createDivide(points)
    left_points = []
    right_points = []
    for point in points:
        if(point[0] <= x and len(left_points) < len(points) // 2):
            left_points.append(point)
        else:
            right_points.append(point)

    return left_points, right_points, x

def ClassifyPointsInStrip(left_p, right_p, x, temp_min):
```

```
    """
    Classify which points are inside of the strip zone

    Args:
        left_p : a list of Point objects with coordinate on the left
of split
        right_p : a list of Point objects with coordinate on the
right of split
        x : Coordinate to split on
        temp_min : The current minimum distance

    Returns:
        left_strip_points : a list of Point objects inside of the
left strip zone
        right_strip_points : a list of Point objects inside of the
right strip zone
    """

    # Classifying points into left and right strip zone
    left_strip_points = []
    i = len(left_p) - 1
    while(i ⩾ 0 and abs(left_p[i][0] - x) ⩽ temp_min):
        left_strip_points.append(left_p[i])
        i -= 1

    right_strip_points = []
    i = 0
    while(i < len(right_p) and abs(right_p[i][0] - x) ⩽ temp_min):
        right_strip_points.append(right_p[i])
        i += 1

    return left_strip_points, right_strip_points
```

Proses *divide* ini membagi *list of points* menjadi subset kiri dan kanan berdasarkan koordinatnya relatif terhadap garis/bidang maya. Tahap ini didukung oleh tiga fungsi, yakni `createDivide(points)`, `splitPoints(points)`, dan `ClassifyPointsInStrip(left_p, right_p, x, temp_min)`.

Fungsi `createDivide(points)` digunakan untuk menghitung koordinat x yang digunakan untuk membagi titik-titik menjadi dua bagian yang sama. Algoritma ini mencari titik tengah pada sumbu x dari *list of points* yang telah diurutkan.

Fungsi `splitPoints` membagi list `points` menjadi dua bagian berdasarkan sumbu x dengan titik tengah di koordinat x yang dihasilkan oleh fungsi `createDivide`. Fungsi ini mengembalikan tiga nilai yaitu left_points, right_points, dan x. left_points dan right_points adalah list dari objek-objek Point yang telah dibagi berdasarkan sumbu x, sedangkan x adalah koordinat yang digunakan untuk membagi *list*.

Fungsi `ClassifyPointsInStrip(left_p, right_p, x, temp_min)` digunakan untuk mengklasifikasikan titik-titik mana yang berada di dalam strip (selisih x-nya dengan koordinat x yang diperoleh dari `createDivide` kurang dari atau sama dengan jarak terdekat sementara yang dihitung pada iterasi sebelumnya). Fungsi ini mengembalikan *list* titik-titik di dalam strip kiri dan strip kanan.

4.4.3 *Conquer*

Setelah step *divide*, dilakukan proses *conquering* untuk menyelesaikan permasalahan tiap subset yang kemudian akan digabungkan. Berikut adalah kode dan penjelasan dari proses *conquer*.

```
def findSmallerThanMinimumDistance(point1, point2, distance, count):
    """
    Finding smaller distance than the minimum distance inside the
strip zone

    Args:
        point1 : a Point object
        point2 : a Point object
        distance : The current minimum distance
```

```
        count : euclidean calculation count

    Returns:
        boolean : True if smaller than the current minimum distance
        temp_distance : the new minimum distance
    """

    # Checking if point1 and point2 have smaller distance than the
current minimum distance
    check = 0
    threshold = distance ** 2
    for i in range (len(point1)):
        check += (abs(point1[i] - point2[i]) ** 2)
        if(check > threshold):
            return False, 0

    # Calculate euclidean distance
    count[0] += 1
    check = math.sqrt(check)
    if(check < distance):
        return True, check

    return False, 0
```

```
def ClassifyPointsInStrip(left_p, right_p, x, temp_min):
    """
    Classify which points are inside of the strip zone

    Args:
        left_p : a list of Point objects with coordinate on the left
of split
        right_p : a list of Point objects with coordinate on the
```

```
right of split
        x : Coordinate to split on
        temp_min : The current minimum distance

    Returns:
        left_strip_points : a list of Point objects inside of the
left strip zone
        right_strip_points : a list of Point objects inside of the
right strip zone
    """

    # Classifying points into left and right strip zone
    left_strip_points = []
    i = len(left_p) - 1
    while(i ⩾ 0 and abs(left_p[i][0] - x) ⩽ temp_min):
        left_strip_points.append(left_p[i])
        i -= 1

    right_strip_points = []
    i = 0
    while(i < len(right_p) and abs(right_p[i][0] - x) ⩽ temp_min):
        right_strip_points.append(right_p[i])
        i += 1

    return left_strip_points, right_strip_points
```

```
def findClosestPairInStrip(left_p, right_p, x, temp_min, temp_pair,
count):
    """
    Finding the smallest distance between points in the strip zone

    Args:
```

```
        left_p : a list of Point objects with coordinate on the left
of split
        right_p : a list of Point objects with coordinate on the
right of split
        x : Coordinate to split on
        temp_min : The current minimum distance
        temp_pair : The current closest pair of point

    Returns:
        new_min : New minimum distance
        new_pair : New closest pair of point
    """

    # Classify points into left and right strip zone
    left_strip_points, right_strip_points =
ClassifyPointsInStrip(left_p, right_p, x, temp_min)

    # Initialize the current minimum distance and the closest pair
    new_min = temp_min
    new_pair = temp_pair

    # Finding pair of points that are more closer than the initial
pair
    for point1 in left_strip_points:
        for point2 in right_strip_points:
            valid, newDistance =
findSmallerThanMinimumDistance(point1, point2, new_min, count)
            if(valid):
                new_min = newDistance
                new_pair = (point1, point2)

    return new_min, new_pair
```

```
def findClosestPair(points, count):
    """
    Finding closest pair of points with divide and conquer

    Args:
        points : a list of Point objects
        count : euclidean calculation count

    Returns:
        distance : the smallest distance
        pair : pair of points with the smallest distance
    """

    # Handle case for n ≠ 2^k
    if(len(points) ⩽ 1):
        return -1, ()
    # Base Recurrens
    elif(len(points) == 2):
        count[0] += 1
        return points[0].distance_to(points[1]), (points[0],
points[1])
    else:
        # Splitting Points
        left_points, right_points, x = splitPoints(points)

        # Finding closest pair of points on the left side
        min_left, pair_left = findClosestPair(left_points, count)

        # Finding closest pair of points on the right side
        min_right, pair_right = findClosestPair(right_points, count)

        # Find the minimum distance and closest pair of points
```

```
between left side and right side
        temp_min = min_left if min_left < min_right and min_left >= 0
else min_right
        new_pair = pair_left if min_left < min_right and min_left >=
0 else pair_right

        # Find the minimum distance and closest pair of points
between the strip zone
        min_strip, pair_strip = findClosestPairInStrip(left_points,
right_points, x, temp_min, new_pair, count)

        # Return the minimum distance and the closest pair of points
        if(min_strip < temp_min):
            return min_strip, pair_strip
        else:
            return temp_min, new_pair
```

Secara garis besar, algoritma di atas bekerja dengan membagi bidang menjadi dua bagian secara rekursif, kemudian menemukan pasangan titik terdekat di bagian masing-masing, dan terakhir menentukan pasangan titik terdekat di antara kedua bagian tersebut. Algoritma ini didukung oleh empat fungsi, yakni `findSmallerThanMinimumDistance(point1, point2, distance, count)`, `ClassifyPointsInStrip(left_p, right_p, x, temp_min)`, `findClosestPairInStrip(left_p, right_p, x, temp_min, temp_pair, count)`, dan `findClosestPair(points, count)`.

Fungsi `ClassifyPointsInStrip(left_p, right_p, x, temp_min)` akan membagi titik-titik yang berada di dalam strip menjadi dua bagian, yaitu bagian kiri dan bagian kanan. Fungsi ini akan mengembalikan list of Point objects untuk setiap bagian tersebut.

Hasil pembagian kemudian dimasukkan ke dalam fungsi `findClosestPairInStrip(left_p, right_p, x, temp_min, temp_pair, count)` yang akan mencari pasangan titik terdekat di dalam strip. Fungsi ini mencari pasangan titik terdekat dengan melakukan perulangan untuk setiap titik pada bagian kiri dan mencocokkannya dengan titik pada bagian kanan. Jika jarak antara kedua titik kurang dari jarak terdekat yang sudah ada, maka akan mengganti nilai jarak terdekat dan pasangan titik terdekat.

Fungsi `findSmallerThanMinimumDistance(point1, point2, distance, count)` akan mengecek apakah jarak antara dua titik kurang dari jarak terdekat yang sudah ada. Fungsi ini akan mengembalikan nilai True jika jarak antara dua titik kurang dari jarak terdekat yang sudah ada, kemudian akan mengembalikan nilai False jika sebaliknya.

Fungsi utama yang digunakan untuk algoritma *divide and conquer* ini adalah `findClosestPair(points, count)`. Fungsi ini merupakan fungsi rekursif yang mengembalikan jarak titik terdekat dan pasangan titik terdekat dengan basis rekurens yaitu saat himpunan titik terdiri dari kurang dari dua titik dan saat himpunan titik terdiri dari dua titik.

Pada bagian awal fungsi, akan dilakukan pengecekan jumlah titik yang ada. Pertama dilakukan pengecekan apabila jumlah titik kurang dari atau sama dengan 1. Kasus ini terjadi ketika N (jumlah keseluruhan titik) bukan merupakan nilai dari suatu 2^k sehingga pembagian himpunan titik itu tidak menjadi rata yang mengakibatkan suatu saat himpunan titik itu tersisa menjadi satu titik saja. Apabila terjadi, fungsi akan mengembalikan jarak yang tidak valid dan pasangan titik yang kosong untuk menghindari perhitungan jarak dengan dirinya sendiri.

Pada sisi lain, kasus himpunan titik terdiri dari dua titik akan mengembalikan *euclidean distance* dari kedua titik tersebut dan pasangan titik tersebut. Untuk kasus rekursif, fungsi akan memanggil dirinya sendiri untuk menemukan pasangan titik terdekat di setiap bagian bidang, yaitu di sebelah kiri dan kanan bidang. Kemudian, fungsi akan menentukan pasangan titik terdekat di antara kedua bagian tersebut.

4.4.5 Kompilasi *Divide and Conquer*

Seluruh fungsi komponen pembentuk algoritma *divide and conquer* dalam penyelesaian permasalahan pencarian pasangan titik terdekat disusun dan dikompilasi dalam fungsi *run* sebagai berikut.

```
def run_divide_and_conquer_closest_pair(points, answer, canvas,
is3D):
    """
    Run divide and conquer algorithm to the closest pair of points
problem and display the solution on GUI

    Args:
        points (list): list of point objects
```

```
        answer (string): solution
        canvas (widget): widget to display plot on GUI
        is3D (boolean): check if the points can be plotted or not
    """

    # Divide And Conquer Algorithm
    start_time = time.time()
    points = quicksort(points, 0 , len(points) - 1)
    ed_count = [0]
    min_distance, pair = findClosestPair(points, ed_count)
    end_time = time.time()

    # Output
    # Closest Pair and Their Distance
    output = "First Point: " + str(pair[0]) + "\n" + "Second Point: "
+ str(pair[1]) + "\n" + f"Minimum Distance: {min_distance}\n" +
"Euclidean Calculations Done: " + str(ed_count[0]) + "\n" +
f"Execution Time: {(end_time - start_time) * 1000} ms\n"
    answer.set(output)

    # Print solution in terminal
    print("Solution by divide and conquer :")
    print(output)

    # If the dimension below or equal to 3
    if(is3D):
        # Create a 3D scatter plot
        figure = plt.figure(figsize=(6, 6))
        ax = figure.add_subplot(111, projection='3d')

        # Filter points
        x = [p[0] for p in points if p != pair[0] and p != pair[1]]
```

```
        if(len(pair[0]) >= 2):
            y = [p[1] for p in points if p != pair[0] and p !=
pair[1]]
        if(len(pair[0]) >= 3):
            z = [p[2] for p in points if p != pair[0] and p !=
pair[1]]

        # Scatter plot
        if(len(pair[0]) == 1):
            ax.scatter(x, 0, 0, alpha=0.25)
        elif(len(pair[0]) == 2):
            ax.scatter(x, y, 0, alpha=0.25)
        else:
            ax.scatter(x, y, z, alpha=0.25)

        if(len(pair[0]) == 1):
            ax.scatter(pair[0][0], 0, 0)
            ax.scatter(pair[1][0], 0, 0)
        elif(len(pair[0]) == 2):
            ax.scatter(pair[0][0], pair[0][1], 0)
            ax.scatter(pair[1][0], pair[1][1], 0)
        else:
            ax.scatter(pair[0][0], pair[0][1], pair[0][2])
            ax.scatter(pair[1][0], pair[1][1], pair[1][2])

        # Set the axis labels
        ax.set_xlabel('X')
        if(len(pair[0]) >= 2):
            ax.set_ylabel('Y')
        if(len(pair[0]) >= 3):
            ax.set_zlabel('Z')
```

```
        # Plot a line between the two points
        if(len(pair[0]) == 1):
            ax.plot([pair[0][0], pair[1][0]], [0, 0], [0, 0], "Red")
        elif(len(pair[0]) == 2):
            ax.plot([pair[0][0], pair[1][0]], [pair[0][1],
pair[1][1]], [0, 0], "Red")
        else:
            ax.plot([pair[0][0], pair[1][0]], [pair[0][1],
pair[1][1]], [pair[0][2], pair[1][2]], "Red")

        # Set Plot Title
        ax.set_title("3D Scatter Plot")

        # Destroy last plot
        if output:
            for child in canvas.winfo_children():
                child.destroy()
        output = None

        # Show the plot
        output = FigureCanvasTkAgg(figure, master = canvas)
        output.draw()
        toolbar = NavigationToolbar2Tk(output,
                                       canvas)
        toolbar.update()
        output.get_tk_widget().pack()

    else: # Destroy last plot if exists
        if output:
            for child in canvas.winfo_children():
                child.destroy()
        output = None
```

4.5 Main Program

Program utama berisi kode untuk melakukan komputasi pencarian solusi *closest pair problem* yang disajikan dalam bentuk GUI serta plotting terhadap hasil untuk dimensi yang lebih kecil sama dengan tiga. Berikut adalah kode dari program utama.

```
# Main File

# Import Library and Modules
from module.bruteforce import run_brute_force_closest_pair
from module.divideconquer import run_divide_and_conquer_closest_pair
import tkinter as tk
from tkinter import *
from tkinter import messagebox as mb
import random
from module.Point import Point

def generate_points(n, dimension):
    """
    Generate n points that has d dimension

    Args:
        n (int): number of points
        dimension (int): the dimension of the points

    Returns:
        points (list) : list of points
    """

    # Generate random data
    coordinates = []
    for _ in range (dimension):
        temp_coordinate = [random.randint(1, n) for _ in range(n)]
```

```
        coordinates.append(temp_coordinate)

    # Pack the data into List of Point Objects
    points = []
    for i in range (n):
        temp = []
        for j in range (dimension):
            temp.append(coordinates[j][i])
        temp_point = Point(*temp)
        points.append(temp_point)

    return points

# Function to display solution on GUI
def display_solution(event):
    """
    Display solutions and plot on the GUI

    Args:
        event (event): Event handler
    """

    # Validate input n and input dimension
    if(input_n.get() ≠ "" and input_dimension.get() ≠ "" and
input_n.get().isnumeric() and input_dimension.get().isnumeric()):
        n = int(input_n.get())
        dimension = int(input_dimension.get())
        if(n ⩽ 1 or dimension ⩽ 0):
            # Display error
            mb.showerror(title="Error", message="Input not valid. N
must be greater than 1 and Dimension must be greater than 0.")
        else:
```

```
            # Set title of solution
            titlebf.set("Brute Force")
            titlednc.set("Divide and Conquer")

            # Generate list of point objects
            points = generate_points(n, dimension)

            # Display plot if dimension below than or equal to 3
            if(dimension <= 3):
                run_brute_force_closest_pair(points, answer1)
                run_divide_and_conquer_closest_pair(points, answer2,
canvas, True)
            else:
                run_brute_force_closest_pair(points, answer1)
                run_divide_and_conquer_closest_pair(points, answer2,
canvas, False)
    else:
        # Display error
        mb.showerror(title="Error", message="Don't forget to input N
and dimension.")

# Initializing GUI
window = tk.Tk()
answer1 = tk.StringVar()
answer2 = tk.StringVar()
titlebf = tk.StringVar()
titlednc = tk.StringVar()

# Configure background
window.configure(background='#1C1C1C')

# Configure resolution and title
```

```
window.geometry("1280x720")
window.title("Closest Pair of Points App")

# Create label for app's name
label1 = tk.Label(master=window, text="Closest Pair of Points",
font=("Helvetica", 24, 'bold'), background='#1C1C1C',
foreground="white")
label1.pack(padx=50)

# Create label for group identity
label2 = tk.Label(master=window, text="by: Michael Jonathan Halim |
13521124\n      Enrique Alifio Ditya         | 13521142",
font=("Helvetica", 14), background='#1C1C1C', foreground="white")
label2.pack(padx=50)

# Create input widgets
inputs = tk.Frame(window, background='#1C1C1C')
inputs.pack(padx=50)

firstInput = tk.Frame(inputs, background='#1C1C1C')
firstInput.pack(pady=10, anchor="w")

label_input_n = tk.Label(master=firstInput, text="N (number of
points)  = ", font=("Helvetica", 10), background='#1C1C1C',
foreground="white")
label_input_n.pack(side=LEFT)
input_n = Entry(firstInput, width = 20)
input_n.focus_set()
input_n.pack()

secondInput = tk.Frame(inputs, background='#1C1C1C')
secondInput.pack(anchor="w")
```

```
label_input_dimension = tk.Label(master=secondInput, text="Dimension
= ", font=("Helvetica", 10), background='#1C1C1C',
foreground="white")
label_input_dimension.pack(side=LEFT)
input_dimension = Entry(secondInput, width = 20)
input_dimension.focus_set()
input_dimension.pack()

# Create button to calculate
buttons = tk.Frame(window, background='#1C1C1C')
buttons.pack(padx=50, pady=10)

button1 = tk.Button(master=buttons, text="Calculate", bg='#1C1C1C',
fg="white")
button1.pack(side=LEFT, padx=10)
button1.bind("<Button-1>", display_solution)

# Create widget for solutions
answers = tk.Frame(window, background='#1C1C1C')
answers.pack(padx=50)

bflabel = tk.Frame(answers, background='#1C1C1C')
bflabel.pack(side=LEFT, padx=20)

dnclabel = tk.Frame(answers, background='#1C1C1C')
dnclabel.pack()

subtitle1 = tk.Label(master=bflabel, textvariable=titlebf,
font=("Helvetica", 15), background='#1C1C1C', foreground="white")
subtitle1.pack()
```

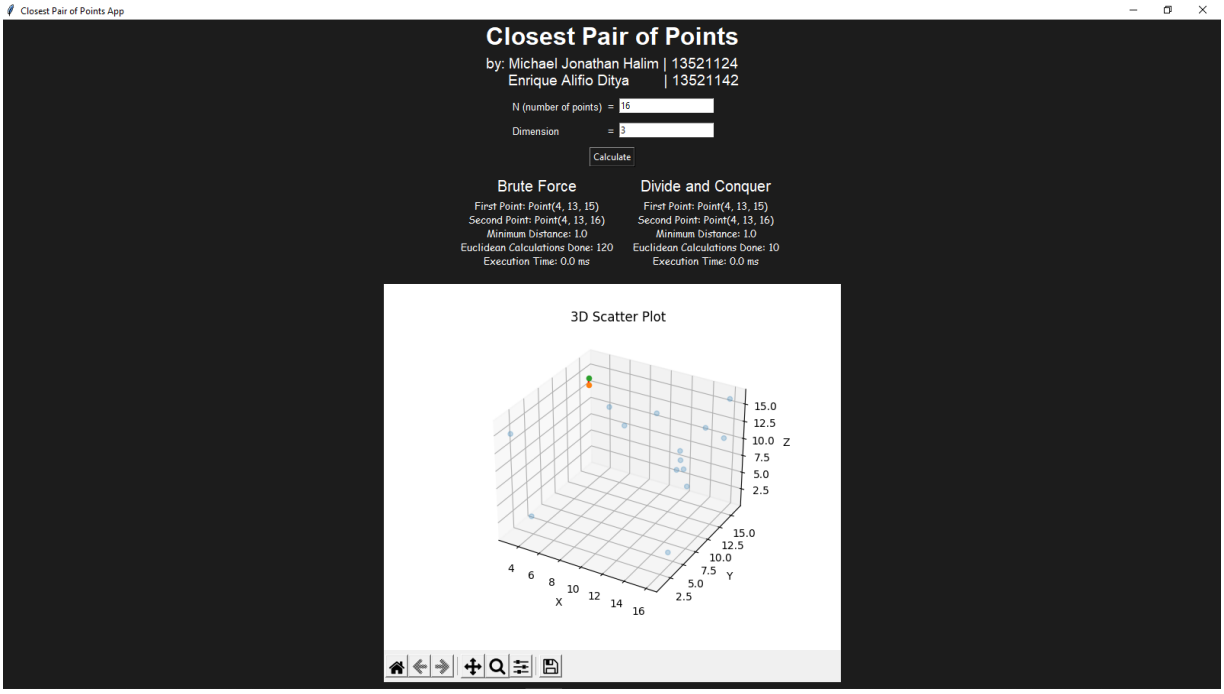
```
label3 = tk.Label(master=bflabel, textvariable=answer1, font=("Comic
Sans MS", 10), background='#1C1C1C', foreground="white")
label3.pack()

subtitle2 = tk.Label(master=dnclabel, textvariable=titlednc,
font=("Helvetica", 15), background='#1C1C1C', foreground="white")
subtitle2.pack()

label4 = tk.Label(master=dnclabel, textvariable=answer2, font=("Comic
Sans MS", 10), background='#1C1C1C', foreground="white")
label4.pack()

# Create widget to show plot
canvas = Canvas(window, background='#1C1C1C', bd=0,
highlightthickness=0, relief='ridge')
canvas.pack(padx=50, pady=(0,10))

window.mainloop()
```

BAB V
HASIL SCREENSHOT

Spesifikasi Laptop untuk Test Case :

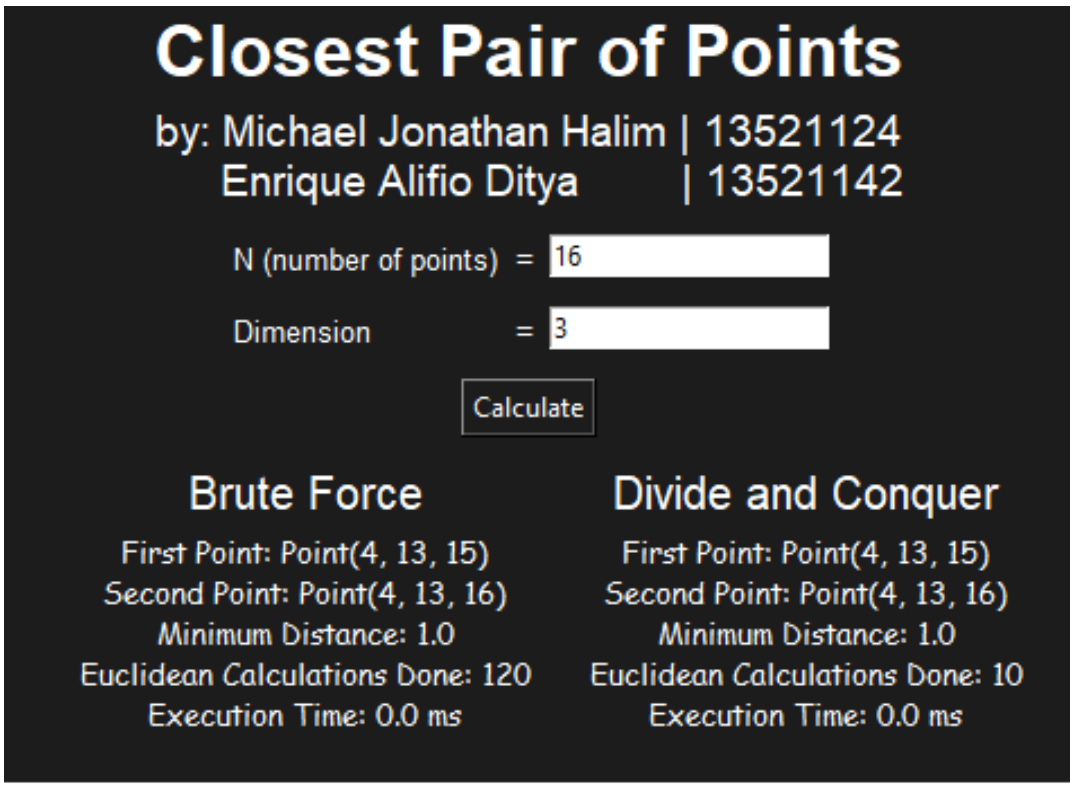
- Asus ROG
- Ryzen 7
- RTX 3060
- RAM 16 GB

5.1 Tampilan GUI Secara Utuh

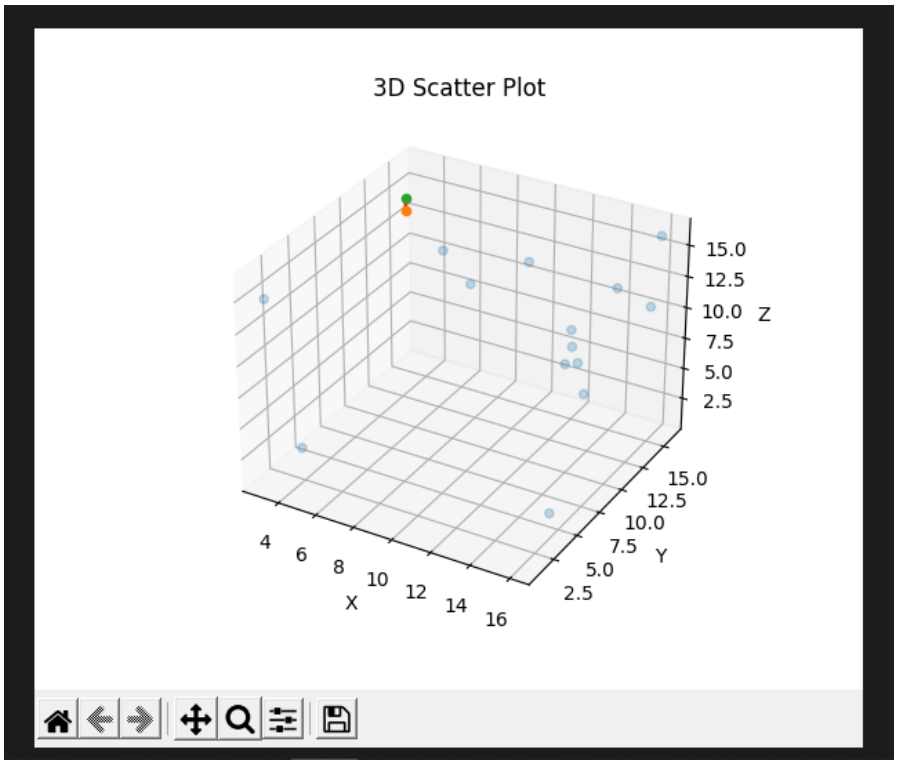


Gambar 5.1.1 Tampilan GUI

5.2 Kasus 16 Titik pada Dimensi 3

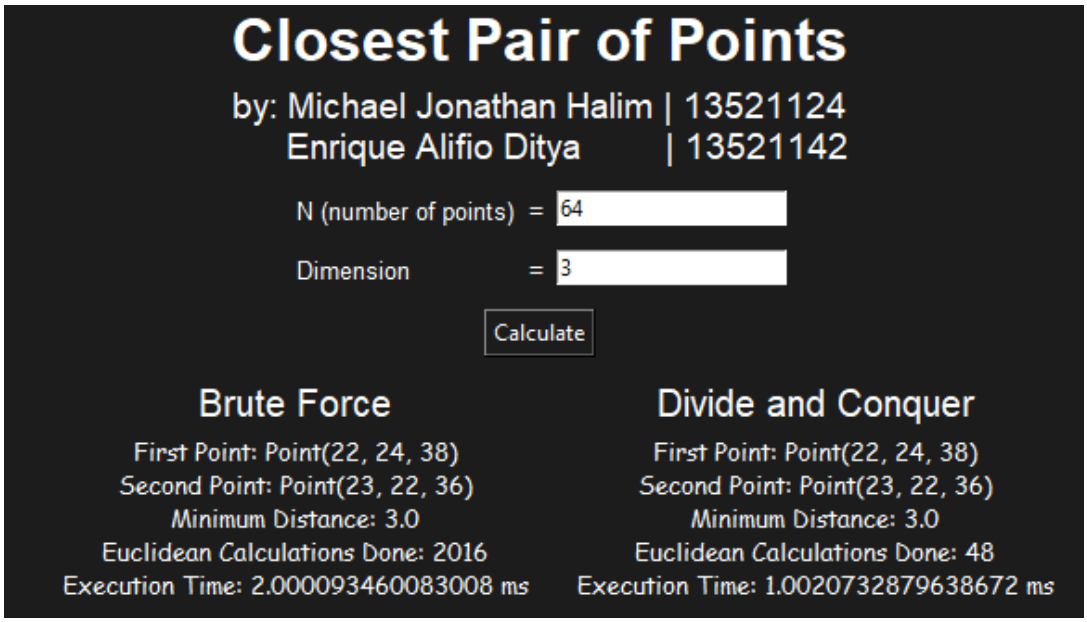


Gambar 5.2.1 Solusi untuk n = 16 di Dimensi 3

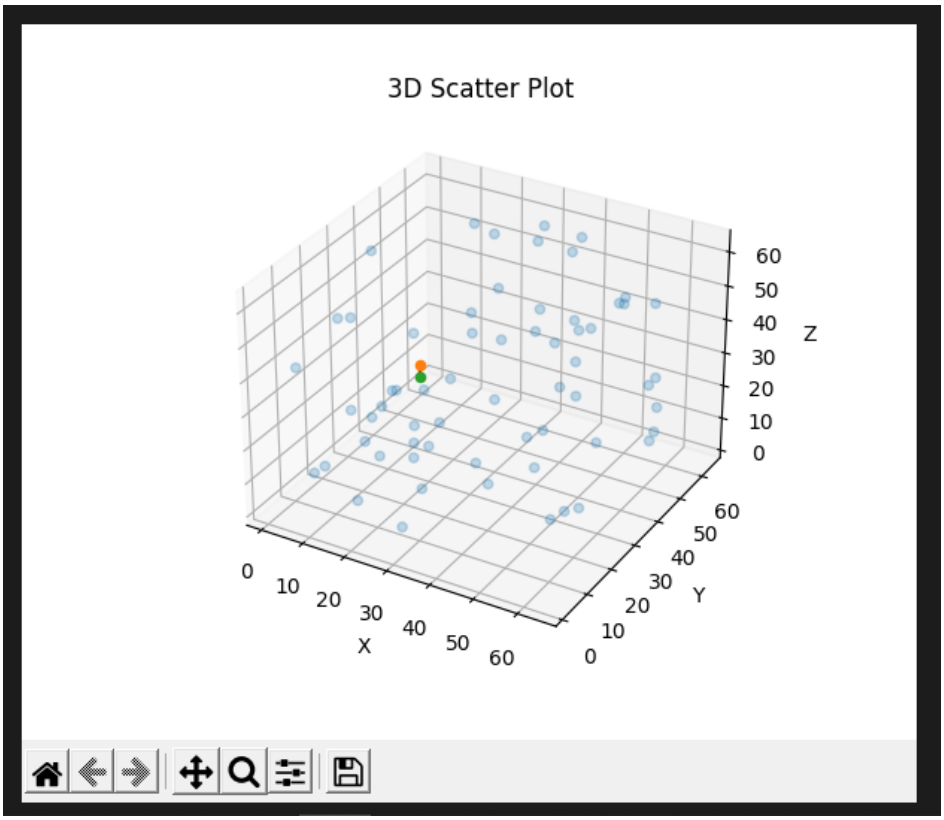


Gambar 5.2.2 Plot untuk n = 16 di Dimensi 3

5.3 Kasus 64 Titik pada Dimensi 3

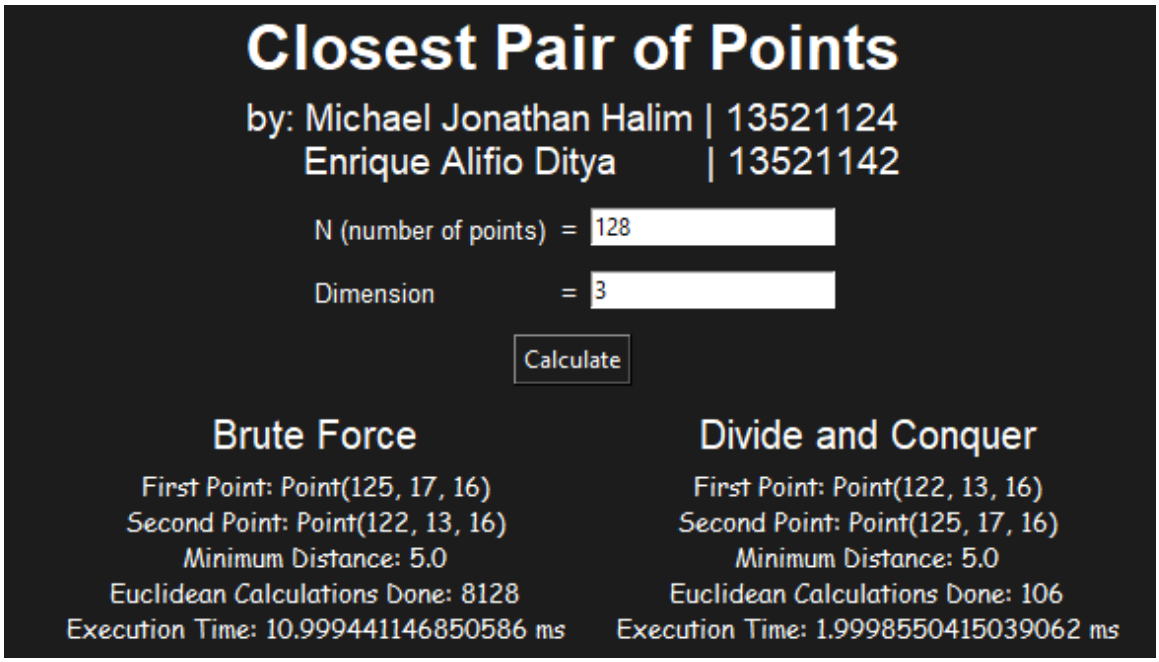


Gambar 5.3.1 Solusi untuk n = 64 di Dimensi 3

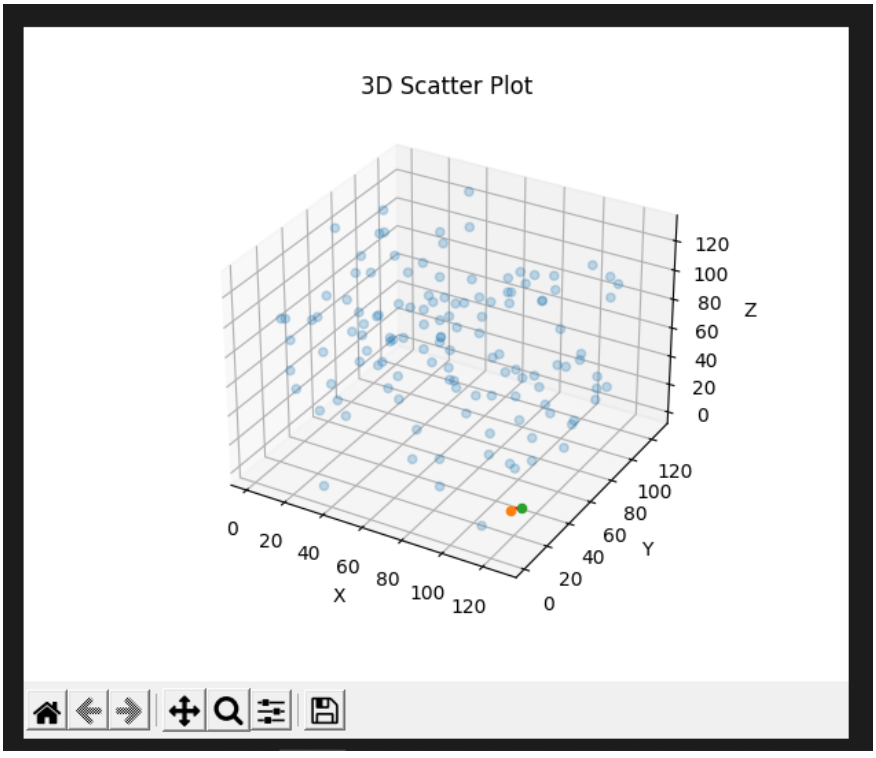


Gambar 5.3.2 Plot untuk n = 64 di Dimensi 3

5.4 Kasus 128 Titik pada Dimensi 3

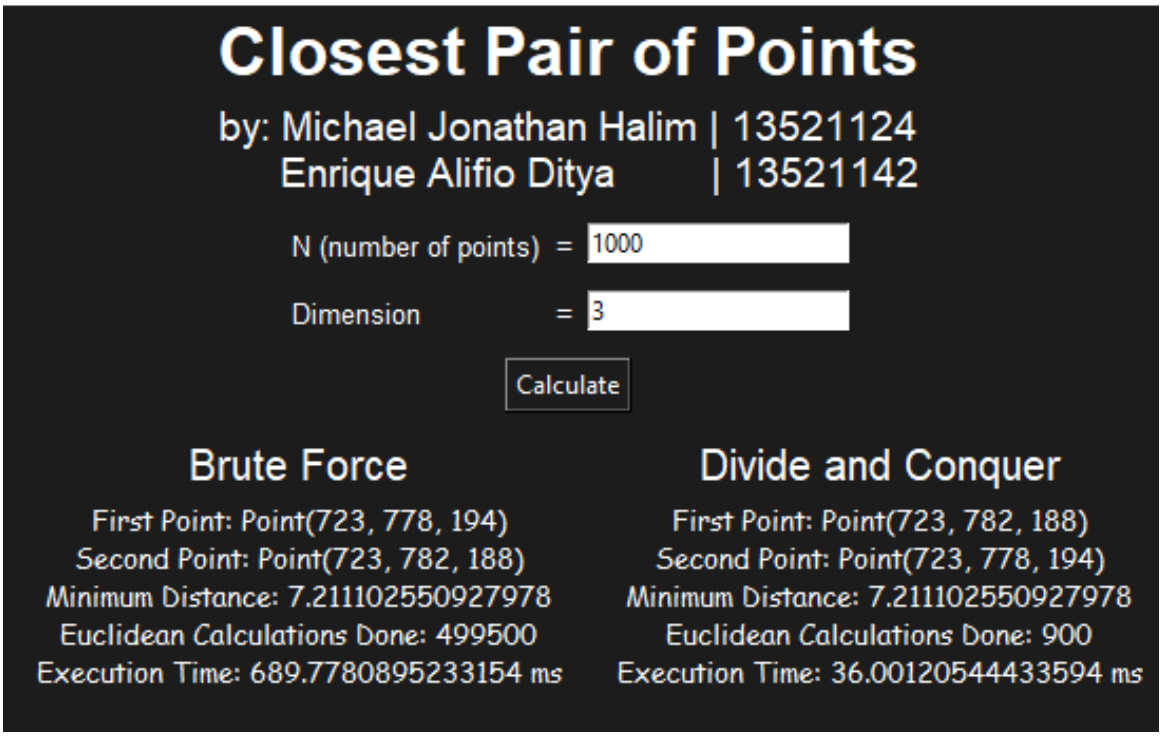


Gambar 5.4.1 Solusi untuk n = 128 di Dimensi 3

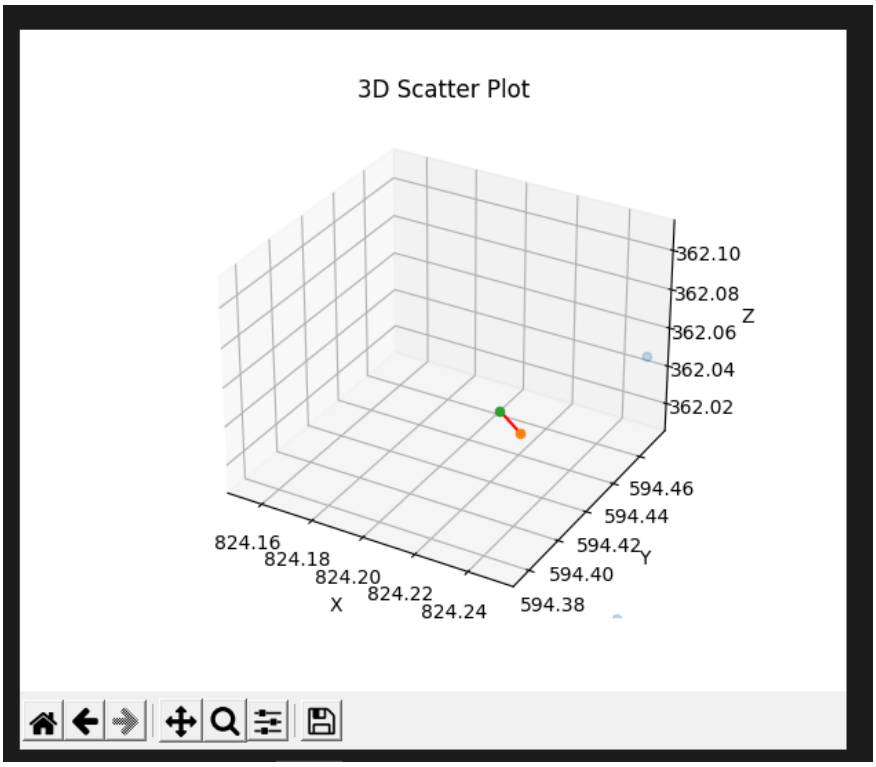


Gambar 5.4.2 Plot untuk n = 128 di Dimensi 3

5.5 Kasus 1000 Titik pada Dimensi 3

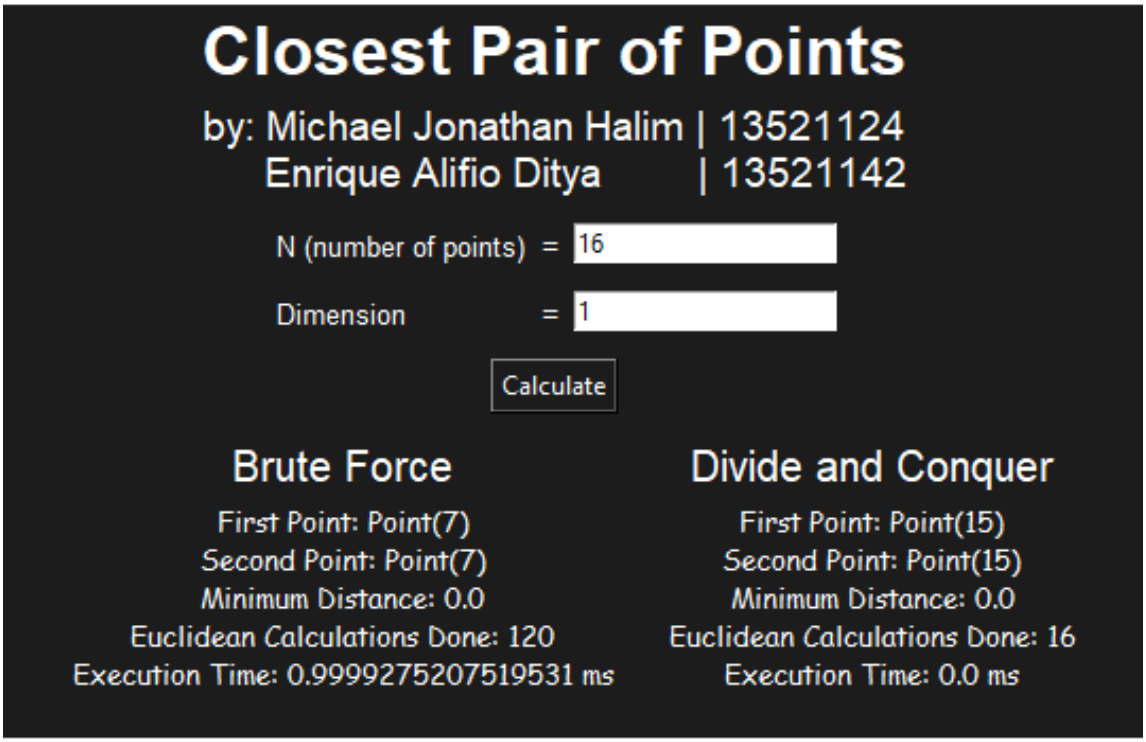


Gambar 5.5.1 Solusi untuk n = 1000 di Dimensi 3

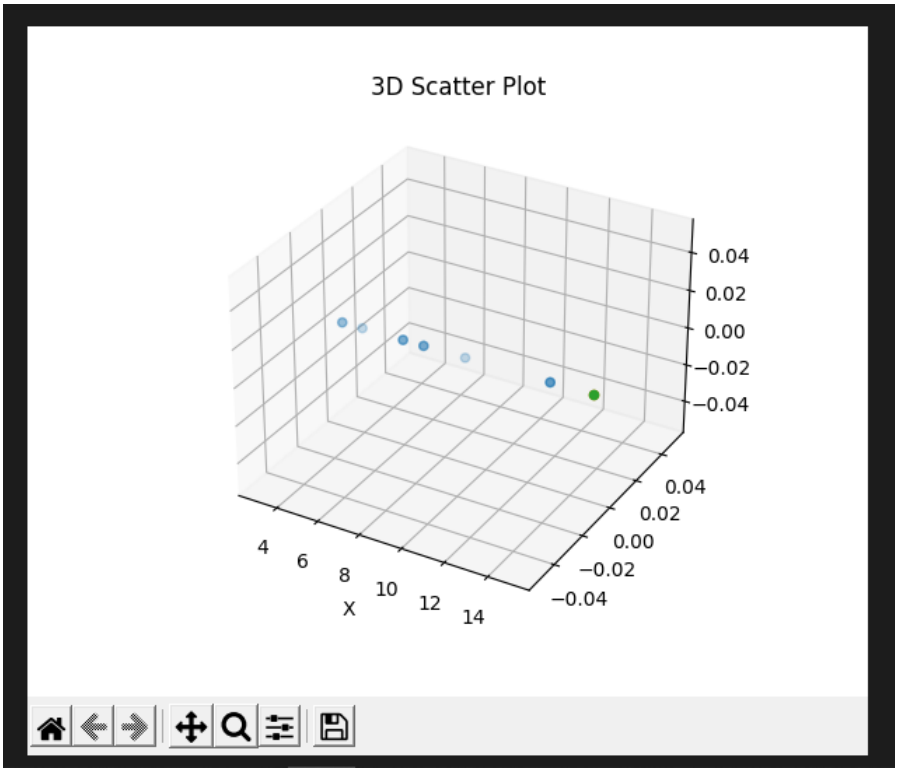


Gambar 5.5.2 Plot untuk n = 1000 di Dimensi 3

5.6 Kasus 16 Titik pada Dimensi 1

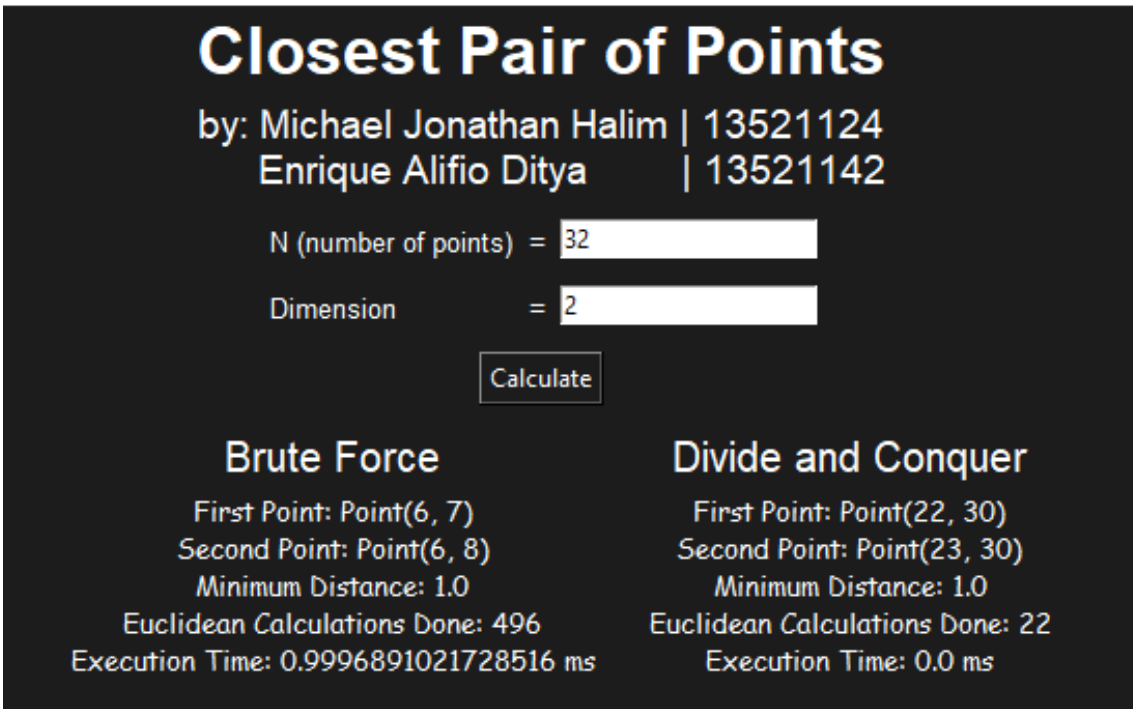


Gambar 5.6.1 Solusi untuk n = 16 di Dimensi 1

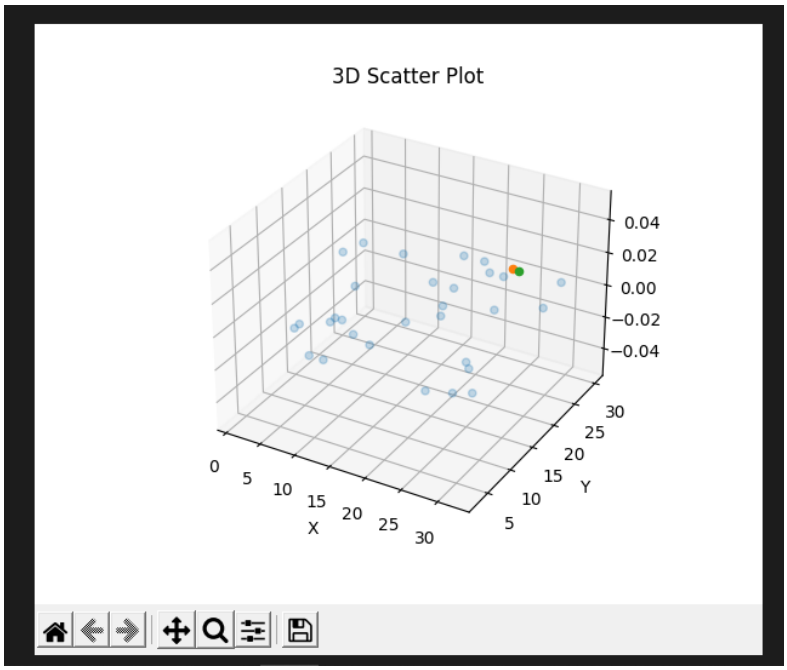


Gambar 5.6.2 Plot untuk n = 16 di Dimensi 1

5.7 Kasus 32 Titik pada Dimensi 2

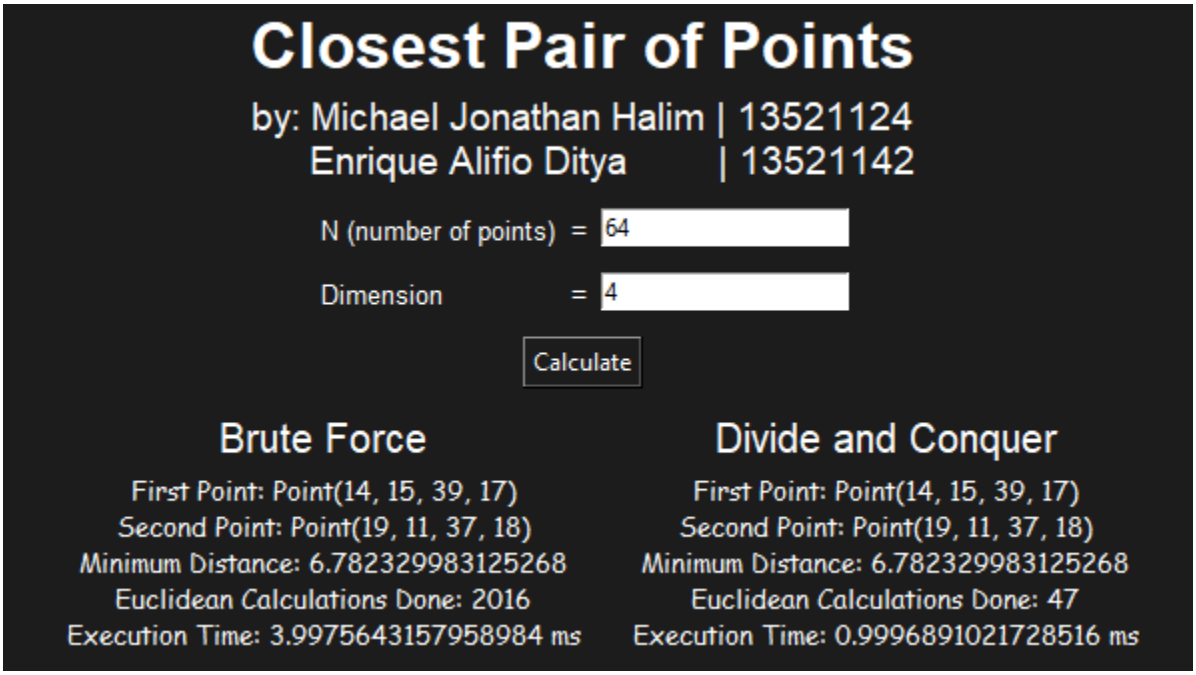


Gambar 5.7.1 Solusi untuk n = 32 di Dimensi 2



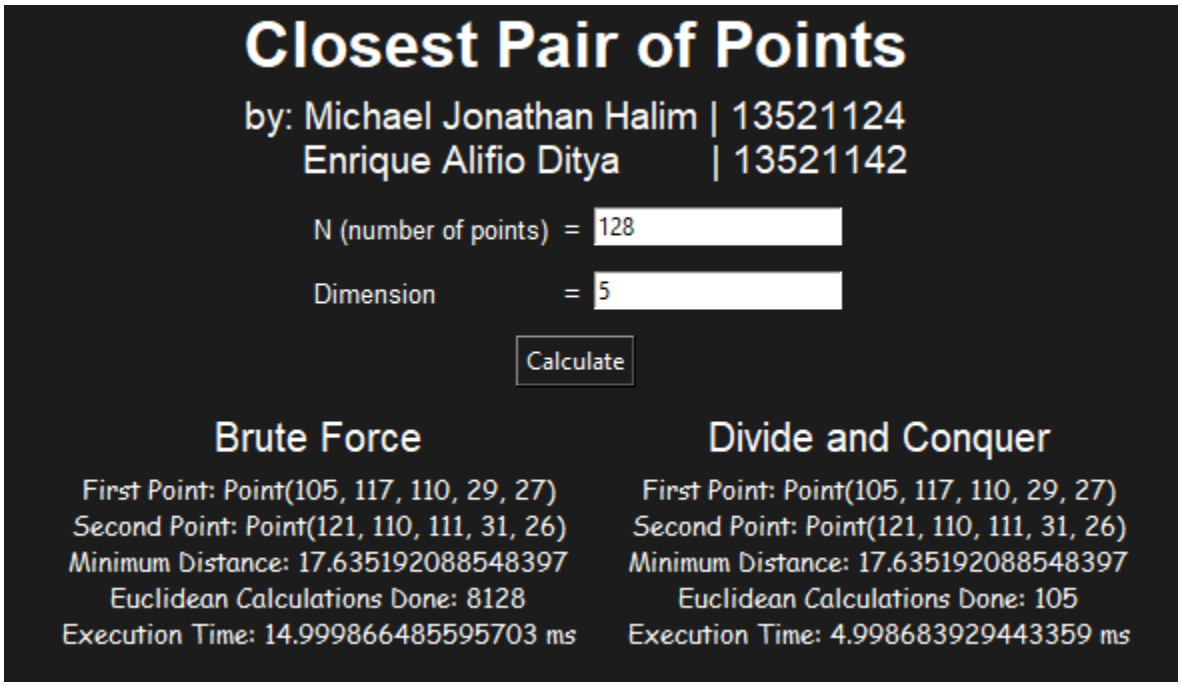
Gambar 5.7.2 Plot untuk n = 32 di Dimensi 2

5.8 Kasus 64 Titik pada Dimensi 4



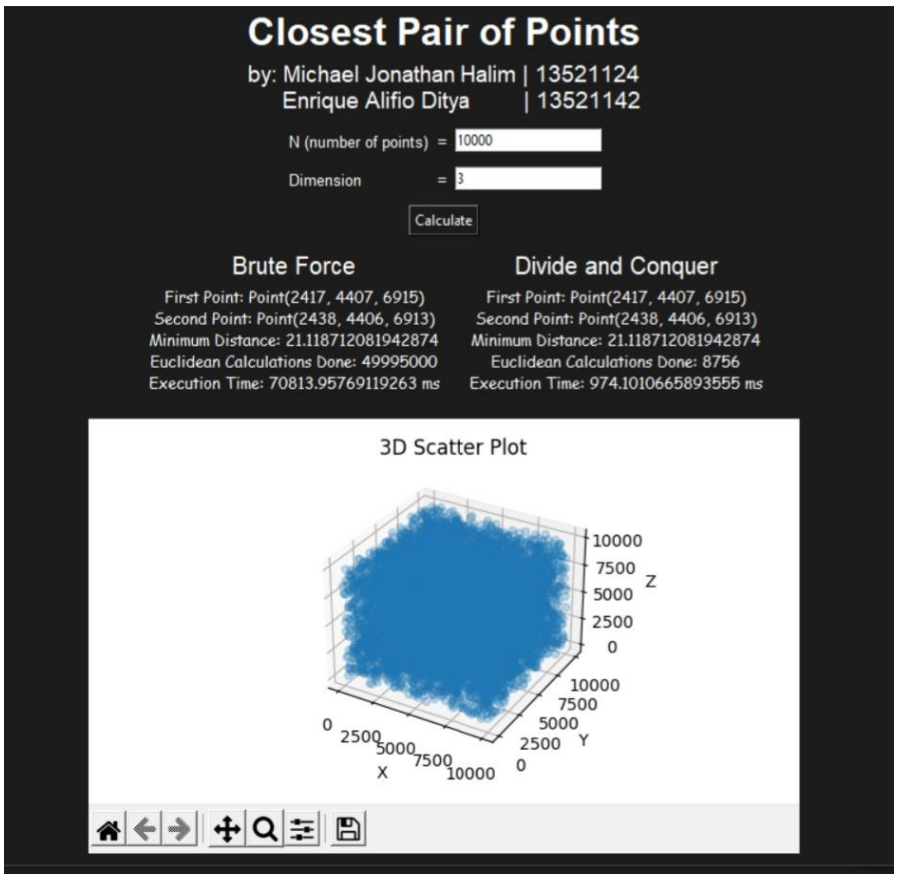
Gambar 5.8.1 Plot untuk n = 64 di Dimensi 4

5.9 Kasus 128 Titik pada Dimensi 5



Gambar 5.9.1 Plot untuk n = 128 di Dimensi 5

5.10 Kasus 10000 Titik pada Dimensi 3



Gambar 5.10.1 Plot untuk n = 10000 di Dimensi 3

5.11 Tampilan CLI

```
Solution by brute force :
First Point: Point(14, 15, 39, 17)
Second Point: Point(19, 11, 37, 18)
Minimum Distance: 6.782329983125268
Euclidean Calculations Done: 2016
Execution Time: 3.9975643157958984 ms

Solution by divide and conquer :
First Point: Point(14, 15, 39, 17)
Second Point: Point(19, 11, 37, 18)
Minimum Distance: 6.782329983125268
Euclidean Calculations Done: 47
Execution Time: 0.9996891021728516 ms
```

Gambar 5.11.1 Contoh Solusi yang Ditampilkan di CLI

BAB VI
ANALISIS KOMPLEKSITAS ALGORITMA

6.1 Analisis Kompleksitas Algoritma *Divide and Conquer*

Dalam pencarian solusi pada pasangan titik terdekat untuk orde R^n, program ini melewati empat fase utama, yakni pra proses sorting, partisi, evaluasi jarak pasangan titik di strip zone, dan evaluasi jarak pasangan titik yang memenuhi syarat tertentu. Empat fase ini didukung oleh fungsi-fungsi utama 'quicksort', 'splitPoints', 'findClosestPairInStrip', dan 'findClosestPair'.

Pada tahap pengurutan, kami memilih algoritma *Quick Sort* agar tidak memperbesar kompleksitas algoritma *divide and conquer* secara keseluruhan. Kompleksitas waktu rata-rata dari *Quick Sort* adalah $O(n\,log(n))$, dengan n sebagai jumlah elemen yang akan diurutkan. Pada setiap rekursi, list dipecah menjadi dua dengan kompleksitas waktu $O(n)$, kemudian dilakukan partisi dan *swap* dengan kompleksitas waktu yang sama. Total kompleksitas waktu adalah $O(n\,log(n))$. Namun, pada kasus terburuk, kompleksitas waktu *Quick Sort* dapat mencapai $O(n^2)$ saat list sudah terurut atau hampir terurut secara terbalik, sehingga partisi tidak efektif dalam membagi list menjadi dua. Pada sisi lain, untuk kasus terbaik, yakni ketika setiap partisi dapat dibagi menjadi dua bagian yang sama ukurannya, kompleksitas waktu *Quick Sort* adalah $O(n\,log(n))$, kontras dengan algoritma pengurutan pada umumnya yang memiliki kompleksitas $O(n^2)$.

Fungsi 'splitPoints' merupakan fungsi untuk mendapatkan hasil partisi dari sebuah himpunan titik menjadi subset bagian kiri dan subset bagian kanan. Fungsi ini memiliki kompleksitas $O(n)$ dengan n berupa jumlah titik. Hal ini disebabkan fungsi hanya perlu menjalankan iterasi keseluruhan titik sebanyak satu kali saja untuk menentukan apakah titik itu tergolong di subset kiri atau subset kanan. Penentuan bagian tersebut dilakukan dengan membandingkan apakah absis dari titik tersebut lebih kecil atau sama dengan absis dari pembatas. Jika iya, maka titik tersebut tergolong ke subset kiri, dan sebaliknya.

Fungsi 'findClosestPairInStrip' merupakan fungsi untuk mencari pasangan titik terdekat di strip zone yang memiliki jarak lebih kecil dari jarak terkecil di subset kiri dan subset kanan. Fungsi ini memiliki kompleksitas waktu $O(n)$ dengan n berupa jumlah titik pada subset kiri. Hal ini disebabkan karena kita tidak perlu memeriksa jarak dari sebuah titik di subset kiri ke seluruh titik di subset kanan. Perlu diingat bahwa kita ingin mencari jarak yang lebih kecil dari *d* (jarak terkecil dari pasangan titik di subset kiri dan subset kanan), sehingga kita tidak perlu menghitung seluruh jarak yang akan melebihi *d.* Hal ini mengakibatkan $T(n)$ = Cn dengan *C* adalah sebuah konstanta dan n adalah jumlah titik di subset kiri. Nilai *C* bergantung pada dimensi dari titik tersebut. Untuk tiga dimensi, nilai *C* terbesarnya adalah 18 yang artinya untuk setiap titik di subset kiri, maksimal perhitungan jarak yang dilakukan adalah sebanyak 18 kali.

Fungsi `findClosestPair' merupakan fungsi utama dari pencarian solusi permasalahan. Fungsi ini merupakan fungsi rekursif yang memanggil dirinya sendiri untuk pencarian solusi di subset kiri dan subset kanan. Ia juga akan memanggil fungsi 'findClosestPairInStrip' untuk menangani kasus strip zone. Pada akhirnya, fungsi ini mengembalikan solusi yang diinginkan. Kompleksitas waktunya adalah $O(n\,(log(n))^{d-1})$ karena himpunan titik yang selalu dibagi dua (proses *divide*) hingga menyentuh basis yaitu ketika jumlah titik dalam himpunan lebih kurang atau sama dengan dua dan proses *conquer* yaitu membandingkan pasangan titik mana yang jaraknya lebih kecil dari tiga kasus kemungkinan, yaitu ketika yang terdekat adalah pasangan titik di subset kiri, pasangan titik di subset kanan, atau pasangan titik melewati pembatas.

BAB VII
LAMPIRAN

7.1 Link Repository Github

Link: https://github.com/AlifioDitya/Tucil2_13521142_13521124

7.2 Checklist Tabel Progress

Poin	Ya	Tidak
1. Program berhasil dikompilasi tanpa ada kesalahan.	✓	
2. Program berhasil running.	✓	
3. Program dapat menerima masukan dan dan menuliskan luaran.	✓	
4. Luaran program sudah benar (solusi closest pair benar).	✓	
5. Bonus 1 dikerjakan.	✓	
6. Bonus 2 dikerjakan.	✓	