

**LAPORAN
TUGAS KECIL 2 IF2211
STRATEGI ALGORITMA**

Pencarian Pasangan Titik Terdekat Dalam Sebuah Set Titik di R^n

oleh

**Fatih Nararya R. I.
Arsa Izdihar I.**

**13521060
13521101**

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
2022/2023**

DAFTAR ISI

DAFTAR ISI	2
BAB I	2
ALGORITMA	2
1.1 Closest Pair Problem dan Solusi Naifnya	2
1.2 Algoritma Divide and Conquer Untuk Pemecahan Closest Pair Problem di Ruang R_n	2
1.3 Pola Pikir dari Algoritma DnC	1
1.4 Kompleksitas dari Algoritma DnC	3
1.5 Anomali dari Algoritma DnC	5
1.6.1 Observasi Sederhana Tentang Dimensi Tinggi	6
1.6.2 Cepatnya Pertumbuhan Maksimum Jumlah Titik yang Perlu Diperiksa	9
1.6.3 Semua Titik Berjarak Kurang Lebih Sama Pada Dimensi Tinggi	10
BAB II	12
SOURCE CODE	12
3.1 Dot.cpp	12
3.2 Dot.hpp	16
3.3 DotCollection.cpp	21
3.4 DotCollection.hpp	27
3.5 ClosestPairData.cpp	31
3.6 ClosestPairData.hpp	32
3.7 visualizer.cpp	33
3.8 visualizer.hpp	35
3.9 main.cpp	35
BAB III	37
EXPERIMEN	37
CLI	37
Bonus: Visualisasi 3D	37
BAB IV	37
HASIL	37
LAMPIRAN	37

BAB I

ALGORITMA

1.1 *Closest Pair Problem* dan Solusi Naifnya

Closest Pair Problem (CPP) adalah sebuah persoalan dalam *computational geometry* yang sederhana : diberikan n buah titik dalam sebuah ruang, temukan sepasang titik yang memiliki jarak paling dekat di antaranya.

Walau persoalan ini sederhana, terdapat banyak algoritma yang dapat digunakan untuk memecahkannya. Algoritma naif untuk persoalan ini adalah melakukan perhitungan jarak untuk semua pasangan titik yang mungkin, lalu memilih pasangan dengan nilai yang paling kecil. Algoritma tersebut memiliki kompleksitas sebagai berikut.

$$T(n) = n + (n - 1) + (n - 2) + \dots + 1$$

$$T(n) = \frac{1}{2} \cdot (n^2 + n)$$

$$\Leftrightarrow$$

$$O(n^2)$$

1.2 Algoritma *Divide and Conquer* Untuk Pemecahan *Closest Pair Problem* di Ruang R^n

Seperti yang telah disebutkan, banyak algoritma yang dapat digunakan untuk menyelesaikan CPP, salah satunya adalah algoritma *Divide and Conquer* (DnC).

Cara kerja algoritma ini untuk sebuah ruang berdimensi d adalah sebagai berikut :

1. Urutkan semua titik yang ada (S) berdasarkan salah satu sumbu ruang tempat titik itu berada, misalnya sumbu x .

2. Bagi semua titik yang ada menjadi dua kelompok yang dipisahkan oleh median nilai sumbu x dari titik-titik yang ada, sebut saja S_L dan S_R .
Selesaikan permasalahan ini pada masing-masing kelompok secara rekursif dengan basis ketika S_x hanya memiliki dua titik.
3. Ambil jarak titik paling kecil dari kedua kelompok dan simpan ke dalam suatu variabel, sebut saja δ .
4. Pada perbatasan antara kedua kelompok, buatlah dua kelompok baru lagi dari S_L dan S_R , yaitu titik-titik pada tiap kelompok yang memiliki jarak maksimum δ . Sebut kelompok baru ini S'_L dan S'_R .
5. Periksa apakah ada titik antara S'_L dan S'_R yang memiliki jarak lebih dari δ . Untuk setiap titik, maksimum titik yang diperiksa dari kelompok seberangnya adalah sebanyak $2 \cdot 3^{d-1}$.
6. Jika pada langkah ke-5 ditemukan titik dengan jarak lebih kecil dari δ , maka jarak itulah yang menjadi jarak terkecil, begitupun sebaliknya.

1.3 Pola Pikir dari Algoritma DnC

Langkah pertama dilakukan agar langkah kedua yang melibatkan penentuan median dapat dilakukan dengan cepat. Hal ini penting mengingat proses *divide* yang diwakili oleh langkah kedua akan terus dilakukan sampai algoritma mencapai basis. Pemilihan algoritma pengurutan yang cepat seperti *mergesort* ataupun *quicksort* memastikan bahwa kompleksitas langkah pertama ini tidak melebihi $O(n \log n)$.

Alasan mengapa pembagian pada langkah kedua menggunakan median dari nilai sumbu x titik adalah untuk memastikan bahwa titik-titik yang ada terbagi menjadi dua kelompok dengan jumlah anggota yang sama (± 1). Pembagian yang setara ini krusial untuk memastikan $b = 2$ di dalam rumusan *running time* dari algoritma DnC ini

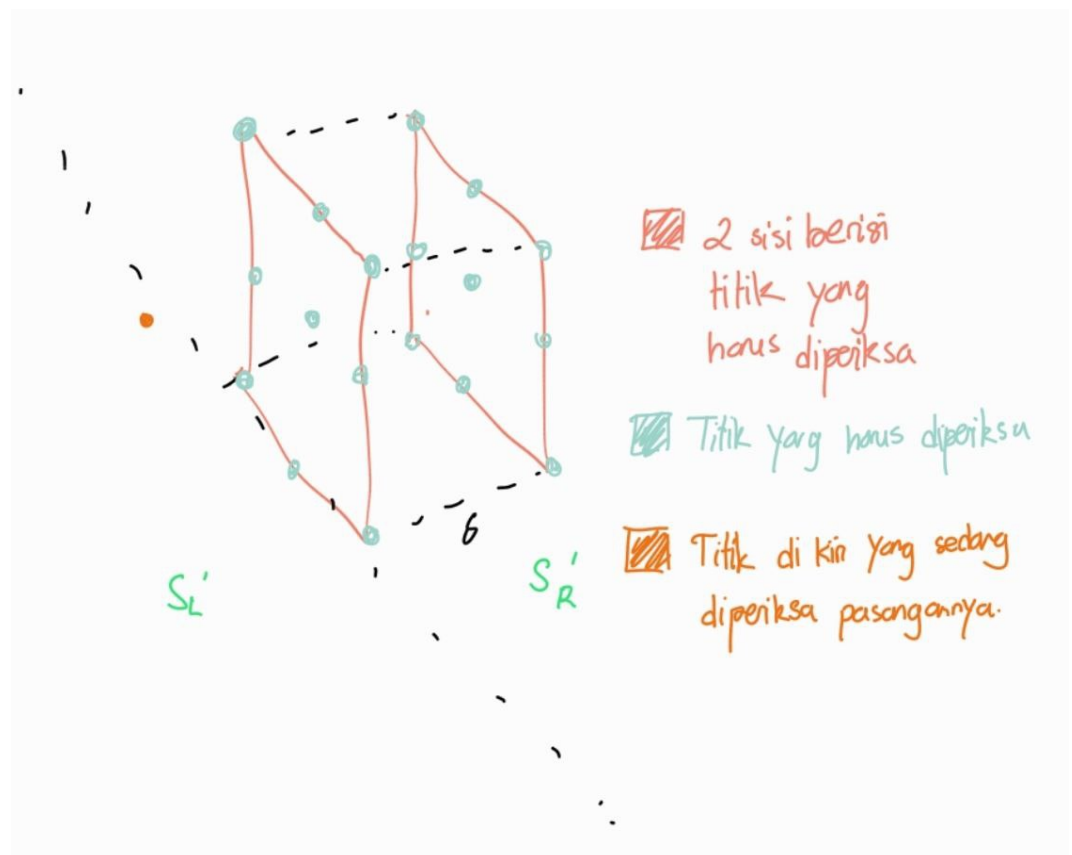
$$T(n) = aT(nb) + f(n)$$

tanpa memperdulikan bentuk dari masukan (S). Telah kita ketahui bahwa algoritma DnC kadang mengalami penurunan efisiensi jika pembagian yang ia lakukan dalam proses *divide* tidak merata (*e.g. quicksort*) sehingga langkah ini telah dibuat sedemikian rupa untuk mencegah terjadinya hal tersebut.

Langkah ke-4 dan ke-5 perlu dilakukan karena bisa saja terdapat pasangan titik antar S_L dan S_R yang memiliki jarak lebih kecil daripada pasangan titik terdekat dari masing-masing S_L dan S_R . Karena kita hanya tertarik pada pasangan titik dengan jarak lebih kecil dari δ maka kita cukup membuat S'_L dan S'_R sebesar itu saja.

Membatasi jumlah titik yang perlu diperiksa langkah ke-5 sangat penting untuk membatasi kompleksitas dari algoritma ini karena jika kita perlu memeriksa semua pasangan titik antara S'_L dan S'_R , maka kompleksitas dari algoritma ini akan sama seperti pendekatan naif yang telah disebutkan pada sub bab sebelumnya, yaitu $O(n^2)$. Untungnya, observasi geometrik menyatakan bahwa jumlah titik yang perlu diperiksa hanyalah sebanyak $2 \cdot 3^{d-1}$. Nilai tersebut didapatkan dari alur pemikiran berikut :

1. Misalkan kita sedang memeriksa sebuah titik di S'_R , maka kita bisa misalkan sebuah setengah kubus (atau *hypercube* untuk d yang tinggi) dengan “rusuk” sepanjang δ di S'_L .
2. “Sisi”-”sisi” yang menjadi pusat perhatian adalah yang menghadap ke titik. Terdapat dua “sisi” seperti itu pada S'_L : “sisi” pertama berada di perbatasan S'_L dan S'_R , “sisi” kedua adalah berada di seberang perbatasan tersebut.
3. Ilustrasi berikut menunjukkan alasan kenapa banyak titik yang perlu diperiksa adalah $2 \cdot 3^{d-1}$.



Karena terdapat dua “sisi” yang mengandung titik yang kita inginkan dan kedua “sisi” ini adalah bangun pada dimensi lebih rendah, maka digunakan $d - 1$ untuk menghitung *Moore’s Neighbourhood* “sisi” tersebut.

4. Selain titik-titik pada gambar, tidak mungkin ada titik-titik lain yang berada di dalam kubus. Sebab, andaikan titik seperti itu ada, maka jarak antara titik di dalam kubus tersebut akan lebih kecil daripada δ yang telah didapatkan.

1.4 Kompleksitas dari Algoritma DnC

Running time dari proses algoritma kami dari langkah kedua sampai terakhir mengikuti relasi rekurens berikut,

$$T_2(n) = 2 \cdot T_2\left(\frac{n}{2}\right) + f(n)$$

dengan

$$f(n) = D \cdot n$$

dan

$$D = 2 \cdot 3^{d-1}$$

$$\Leftrightarrow$$

$$f(n) \in \Theta(n^1)$$

sehingga menurut *Master’s Theorem*

$$T_2(n) \in \Theta(n \cdot \log(n))$$

karena

$$a = 2$$

$$b = 2$$

$$d = 1$$

$$\Rightarrow$$

$$a < b^d$$

Sehingga $O_2(n \cdot \log(n))$.

Mengasumsikan bahwa algoritma pengurutan titik yang dilakukan pada langkah pertama algoritma ini memiliki efisiensi waktu optimal dengan memiliki kompleksitas $O(n \cdot \log(n))$, maka kompleksitas dari algoritma kami adalah

$$\begin{aligned} O_1(n \cdot \log(n)) + O_2(n \cdot \log(n)) \\ = O(n \cdot \log(n)) \end{aligned}$$

1.5 Anomali dari Algoritma DnC

Namun, walaupun efisiensi algoritma ini tampaknya adalah $O(n \cdot \log(n))$, terdapat sebuah observasi yang menyangkal pernyataan ini pada dimensi lebih tinggi. Berikut adalah jumlah perhitungan *euclidean distance* (N) milik program kami untuk $n = 1000$ dan d yang cukup tinggi untuk masing-masing algoritma.

Dimensi	DnC	<i>Brute force</i>
25	499500	499500
50	499500	499500
100	499500	499500

Berikut pula adalah $T(n)$ untuk masing-masing algoritma dalam detik.

Dimensi	DnC	<i>Brute force</i>
25	0.41	0.33

50	0.82	0.66
100	1.62	1.33

Tampak sebuah anomali berupa algoritma DnC yang semestinya memiliki kompleksitas lebih baik daripada *brute force* malah memiliki *running time* yang lebih buruk. Lebih menariknya lagi adalah fakta bahwa N dari keduanya adalah sama persis. Terdapat beberapa hal yang dapat menjelaskan anomali ini.

Semua kode yang digunakan untuk mendapatkan data pada subbab ini ada pada *branch* `Fatih-Analysis`

1.6.1 Observasi Sederhana Tentang Dimensi Tinggi

Anomali yang telah disebutkan sebelumnya sebetulnya berakar pada kenapa N sama untuk kedua algoritma pada dimensi yang lebih tinggi (sesuatu yang tidak terjadi pada d kecil seperti ditunjukkan oleh data eksperimen yang dicantumkan pada bab 3 di mana $N_{DnC} \ll N_{bf}$).

Tetapi sebelum menelaah hal tersebut, perlu diperjelas mengapa $T_{DnC}(n) > T_{bf}(n)$ untuk $N_{DnC} = N_{bf}$ seperti yang terjadi pada d tinggi. Dari cara kerja algoritma yang telah dijelaskan pada subbab 1, tampak bahwa algoritma DnC untuk CPP adalah berjenis *easy split, hard join*. Untuk melakukan pemecahan problem menjadi subproblem, cukup mencari median dari titik yang ada. Suatu proses yang hanya memakan $O(1)$ karena titik-titik yang ada telah diurutkan terlebih dahulu. Tetapi, menggabungkan solusi dari subproblem yang telah dipecahkan memakan $O(n)$. Maka, seseorang mungkin mengasumsikan keberadaan *overhead* berdasarkan fakta sederhana bahwa algoritma DnC

melakukan relatif lebih banyak hal selain mencari *euclidean distance*, dibandingkan algoritma *brute force* yang hanya melakukan *looping* saja selain mencari *euclidean distance*.

Hal tersebut mungkin memiliki kebenaran, tetapi apakah benar mencari median titik, menentukan mana dari dua bilangan yang lebih besar, memastikan pasangan titik yang diperiksa untuk tiap titik pada proses *join* tidak melebihi D merupakan proses yang benar-benar mahal secara komputasi? Setelah penyelidikan lebih lanjut, ada satu bagian dari algoritma DnC kami yang memang mahal pada dimensi tinggi, tepatnya pada bagian langkah ke-5. Pada langkah ke-5, sebelum melakukan perhitungan jarak dari suatu pasangan titik, kami melakukan pemeriksaan dasar dulu terhadap koordinatnya. Untuk nilai $m = [1, d]$, kami memeriksa apakah ada nilai m sehingga $x_{ma} - x_{mb} > \delta$. Jika ada, maka kami tidak melanjutkan perhitungan *euclidean distance* karena jelas nilainya tidak akan kurang dari δ .

Cukup jelas kenapa komputasi ini lebih mahal untuk d besar. Tetapi dari penjelasan yang telah diberikan, tampak bahwa komputasi tersebut dilakukan untuk mencegah komputer dari melakukan penghitungan *euclidean distance* yang percuma, sehingga untuk menentukan apakah lebih baik proses tersebut kita hilangkan pada d tinggi, kita perlu menelaah apakah keuntungannya melebihi biayanya. Sebelum membicarakan *cost-benefit analysis* dari proses tersebut, kita dapat dengan mudah memperoleh jawabannya hanya dengan menjalankan program yang telah dimodifikasi untuk melewati langkah tersebut. Berikut adalah $T(n)$ untuk DnC dengan dan tanpa proses tersebut dalam detik.

Dimensi	DnC (Tanpa)	DnC (Dengan)
25	0.32	0.41
50	0.64	0.82
100	1.31	1.62

Dari analisis empiris tersebut terbukti bahwa proses tersebut memiliki biaya lebih mahal daripada keuntungannya. Kenapa begitu? Hal tersebut disebabkan oleh sebuah properti mendasar dari ruang berdimensi tinggi, yaitu ia memiliki banyak sumbu.

Misalkan kita ambil dua titik acak A dan B . Misal *euclidean distance* antara kedua titik adalah s dan jarak antara kedua titik tersebut dalam sebuah sumbu m adalah Δx_m . Menurut statistika, karena titik diambil secara acak, paling memungkinkan bahwa nilai Δx_m untuk setiap m bernilai 1 sampai d cenderung merata satu sama lain. Sehingga kita asumsikan Δx_m untuk semua m bernilai h . Titik ini akan “disaring” oleh proses tadi jika dan hanya jika $h > \delta$, yang berarti $s > \sqrt{d} \cdot \delta$. Dari sini tampak jelas mengapa keuntungan dari proses ini tidak sepadan dengan biayanya pada d besar. *Threshold* agar sebuah pasangan titik dapat tersaring $\sim \sqrt{d}$ sehingga semakin tinggi d , semakin sedikit jumlah pasangan titik yang tersaring. Akibatnya, keuntungan proses ini berkurang. Di sisi lain, seperti yang telah disebutkan, biaya proses ini $\sim d$. Maka, semakin tinggi d , waktu komputasi yang dihemat akibat proses ini menurun, sesuai dengan data yang telah didapatkan.

1.6.2 Cepatnya Pertumbuhan Maksimum Jumlah Titik yang Perlu Diperiksa

Mengulangi yang telah disampaikan pada beberapa subbab sebelumnya, D adalah jumlah pasangan titik yang perlu diperiksa untuk sebuah titik pada proses penggabungan subproblem, dirumuskan oleh

$$D = 2 \cdot 3^{d-1}$$

dengan d adalah dimensi dari ruang. Dari sini tampak bahwa walaupun untuk sebuah ruang berdimensi d jumlah maksimum titik yang perlu diperiksa adalah konstan, karena D tumbuh secara eksponensial terhadap d , untuk sebuah d sebarang, ada sebuah nilai n_0 di mana jika $n < n_0$, algoritma akan berlaku layaknya ia memeriksa semua pasangan titik yang ada pada proses penggabungan subproblem. Nilai n_0 tersebut adalah D itu sendiri, yang nilainya untuk sebuah d diberikan pada tabel berikut untuk mengilustrasikan skala dari masalah ini.

n_0	d
39,366	10
564,859,072,962	25
4.87e+23	50
3.43e+47	100

Tampak bahwa untuk dimensi tinggi, n perlu bernilai cukup besar agar algoritma DnC tidak seakan-akan memiliki kompleksitas $O(n^2)$.

1.6.3 Semua Titik Berjarak Kurang Lebih Sama Pada Dimensi Tinggi

Euclidean Distance bukanlah ukuran yang baik pada dimensi tinggi. Hal ini karena jika kita memilih suatu titik sebarang pada ruang berdimensi d , misalkan A , yang memiliki tetangga terdekat titik B dan tetangga terjauh titik C , maka

$\lim_{d \rightarrow \infty} \frac{|AB|}{|AC|} = 1$. Ini artinya, jarak antara titik pada ruang dimensi tinggi semakin seragam satu sama lain.

Misalkan sebuah ruang (atau subruang) yang dipenuhi titik memiliki sebuah rata-rata (μ_s) dan deviasi standar (σ_s) untuk jarak titik-titiknya. Dengan kondisi tersebut, algoritma kami akan (kemungkinannya paling besar) menghasilkan $\delta = \mu_s - \sigma_s$. Karena properti dari ruang berdimensi tinggi yang telah disebutkan sebelumnya $\lim_{d \rightarrow \infty} \sigma_s = 0$, sehingga $\lim_{d \rightarrow \infty} \delta = \mu_s$. Maka daerah tengah $\pm \delta$ akan mencakup semakin banyak titik semakin besarnya d . Hal ini membuat program kami perlu mencocokkan semakin banyak pasangan titik pada proses *join* ketika d tinggi.

Kami ingin membandingkan magnitudo dari δ antar d . Tetapi, semakin besar d akan semakin besar pula δ karena ada lebih banyak sumbu yang “menyumbang” jarak. Karena itu, kami membuat sebuah kuantitas baru bernama jarak ternormalisasi (δ') yang merupakan kuadrat dari selisih jarak antar titik pada suatu sumbu, sehingga didefinisikan sebagai $\delta' = \frac{\delta^2}{d}$. Kami pun memodifikasi program untuk menghasilkan statistik dasar untuk magnitudo dari δ' dari keseluruhan proses berjalannya algoritma DnC untuk $n = 10000$, d

divariasikan, dan nilai sebuah koordinat pada sebuah sumbu dalam program kami adalah 0 ± 1000 . Statistik dasar tersebut terdiri dari rata-rata jumlah titik di tengah selama proses penggabungan (μ_δ), jumlah titik di bagian tengah paling banyak selama proses penggabungan (M_δ), dan rata-rata dari 10^{th} persentil jumlah titik di tengah selama proses penggabungan (μ'_δ). Hasilnya ditunjukkan oleh tabel berikut.

d	μ_δ	M_δ	μ'_δ
3	78,963	1,030,399	357,473
6	209,336	1,177,468	538,543
10	299,889	992,308	595,798
25	430,366	973,567	649,775
50	498,256	823,562	657,661

Dapat dilihat bahwa μ_δ meningkat dan M_δ menurun seiring d meningkat, yang dapat menjadi indikasi penurunan σ_s (membuktikan bahwa ini benar-benar terjadi membutuhkan analisis matematis dan statistik mendalam yang di luar lingkup laporan maupun matkul ini).

Kami kemudian melanjutkan analisis dengan mengukur berapa banyak titik yang berada di tengah $\pm \delta (n')$ sepanjang keberjalanan algoritma ini. Maka, kami memodifikasi program lagi untuk menghasilkan statistik dasar terkait n' untuk $n = 10000$ dan d divariasikan. Statistik dasar tersebut terdiri dari rata-rata

jumlah titik di tengah selama proses penggabungan ($\mu_{n'}$), jumlah titik di bagian tengah paling banyak selama proses penggabungan ($M_{n'}$), dan rata-rata dari 10th persentil jumlah titik di tengah selama proses penggabungan ($\mu'_{n'}$). Hasilnya ditunjukkan oleh tabel berikut.

d	$\mu_{n'}$	$M_{n'}$	$\mu'_{n'}$
2	17	455	93
6	25	2500	168
10	27	5000	196
25	29	10000	210
50	29	10000	210

Hasilnya adalah semakin tinggi d , semakin banyak titik yang berada di bagian tengah, sesuai dengan analisis kami pada awal subbab ini.

BAB II

SOURCE CODE

3.1 Dot.cpp

```
#include "Dot.hpp"
#include <iostream>
#include <random>

long Dot::totalEuclidean = 0;

Dot::Dot() : Dot(3)
{
}

Dot::Dot(int dimension)
{
    std::random_device rd;
    std::default_random_engine generator(rd());
    std::uniform_real_distribution<double> distribution(-1000,
1000);

    coordinates = new double[dimension];
    this->dimension = dimension;

    for (int i = 0; i < dimension; i++)
    {
        coordinates[i] = distribution(generator);
    }
}

Dot::Dot(const Dot &givenDot)
{
    this->dimension = givenDot.dimension;
    this->coordinates = new double[dimension];
    for (int i = 0; i < dimension; i++)
    {
        coordinates[i] = givenDot[i];
    }
}
```



```

    }
}

Dot::~~Dot()
{
    delete[] coordinates;
}

Dot &Dot::operator=(const Dot &givenDot)
{
    this->dimension = givenDot.dimension;
    this->coordinates = new double[dimension];
    for (int i = 0; i < dimension; i++)
    {
        coordinates[i] = givenDot[i];
    }
    return *this;
}

double Dot::getSquaredDistance(Dot &targetDot)
{
    Dot::totalEuclidean++;
    double sum = 0;
    for (int i = 0; i < dimension; i++)
    {
        sum += pow(getCoordinateAt(i) -
targetDot.getCoordinateAt(i), 2);
    }
    return sum;
};

double Dot::getDistance(Dot &targetDot)
{
    return sqrt(getSquaredDistance(targetDot));
}

double Dot::getCoordinateAt(int dimension) const
{
    return coordinates[dimension];
}

```

```

double Dot::operator[](int nthDimension) const
{
    return coordinates[nthDimension];
}

bool Dot::operator==(const Dot &givenDot)
{
    bool same = dimension == givenDot.dimension;
    int i = 0;
    while (same && i < dimension)
    {
        same = givenDot[i] == getCoordinateAt(i);
        i += same;
    }
    return same;
};

bool Dot::operator<(const Dot &givenDot)
{
    return getCoordinateAt(0) < givenDot[0];
};

bool Dot::operator>(const Dot &givenDot)
{
    return getCoordinateAt(0) > givenDot[0];
};

bool Dot::operator<=(const Dot &givenDot)
{
    return *this < givenDot || *this == givenDot;
};

bool Dot::operator>=(const Dot &givenDot)
{
    return *this > givenDot || *this == givenDot;
};

void Dot::swap(Dot &givenDot)
{

```

```

    int tempDimension = givenDot.dimension;
    double *tempCoordinates = givenDot.coordinates;

    givenDot.dimension = dimension;
    dimension = tempDimension;
    givenDot.coordinates = this->coordinates;
    this->coordinates = tempCoordinates;
}

void Dot::print()
{
    cout << "(";
    cout << getCoordinateAt(0);
    for (int i = 1; i < dimension; i++)
    {
        cout << ", " << getCoordinateAt(i);
    }
    cout << ")" << endl;
}

bool Dot::compare(Dot &dot1, Dot &dot2)
{
    return dot1 < dot2;
}

int Dot::getTotalEuclidean()
{
    return Dot::totalEuclidean;
}

void Dot::resetTotalEuclidean()
{
    Dot::totalEuclidean = 0;
}

bool Dot::bottomBoundDistance(double delta, Dot &givenDot)
{
    bool moreThan = false;
    for (int i = 0; i < dimension; i++)
    {

```

```

        moreThan = (abs(getCoordinateAt(i) - givenDot[i]) >
delta);
        if (moreThan)
        {
            break;
        }
    }
    return moreThan;
}

```

3.2 Dot.hpp

```

#include <vector>
#include <cstdlib>
#include <cmath>

using namespace std;

#ifndef DOT
#define DOT

class Dot
{
private:
    double *coordinates;
    int dimension;
    static long totalEuclidean;

public:
    /**
     * @brief Construct a new Dot object with default dimension
of 3
     *
     */
    Dot();

    /**

```

```

    * @brief Construct a new Dot object with the given
dimension
    *
    * @param dimension
    */
Dot(int dimension);

/**
 * @brief Destroy the Dot object
 *
 */
~Dot();

/**
 * @brief Copy constructor for Dot
 *
 * @param givenCoordinate
 * @return Dot&
 */
Dot(const Dot &givenDot);

/**
 * @brief Construct a new Dot object using the assignment
operator
 *
 * @param givenCoordinate
 */
Dot &operator=(const Dot &givenCoordinate);

/**
 * @brief Determine if this dot is the same as another dot
 *
 * @param givenDot
 */
bool operator==(const Dot &);

/**
 * @brief Determine if a dot is on the left of another dot
on the x-axis
 *

```

```

    * @return true the dot on the left of the operator is on
the left of the other dot
    * @return false the dot on the left of the operator is on
the right of the other dot
    */
    bool operator<(const Dot &);

    /**
    * @brief Determine if a dot is on the right of another dot
on the x-axis
    *
    * @return true the dot on the left of the operator is on
the right of the other dot
    * @return false the dot on the left of the operator is on
the left of the other dot
    */
    bool operator>(const Dot &);

    /**
    * @brief Determine if a dot is on the left or same as of
another dot on the x-axis
    *
    * @return true the dot on the left of the operator is on
the left of the other dot
    * @return false the dot on the left of the operator is on
the right of the other dot
    */
    bool operator<=(const Dot &);

    /**
    * @brief Determine if a dot is on the right of another dot
on the x-axis
    *
    * @return true the dot on the right of the operator is on
the right of the other dot
    * @return false the dot on the right of the operator is on
the left of the other dot
    */
    bool operator>=(const Dot &);

```

```

/**
 * @brief Get the square of the distance of this dot to the
targetDot. It's assumed the target dot has the same dimension.
 *
 * @param targetDot
 * @return double
 */

double getSquaredDistance(Dot &targetDot);
/**
 * @brief Get the distance of this dot to the targetDot.
It's assumed the target dot has the same dimension.
 *
 * @param targetDot
 * @return double
 */
double getDistance(Dot &targetDot);
/**
 * @brief Get the coordinate at nthDimension
 * @param nthDimension the desired dimension of coordinate
to be fetched
 * @return double
 */
double getCoordinateAt(int nthDimentsion) const;

/**
 * @brief Get the coordinate at nthDimension
 * @param nthDimension the desired dimension of coordinate
to be fetched
 * @return double
 */
double operator[](int nthDimentsion) const;

/**
 * @brief Print the dot coordinates
 *
 */
void print();

/**

```

```

    * @brief Compare two dots (for sorting)
    *
    */
static bool compare(Dot &dot1, Dot &dot2);

/**
    * @brief Get the total euclidean variable
    *
    * @return int
    */
static int getTotalEuclidean();

/**
    * @brief Reset the total euclidean calculation to zero
    *
    * @return int
    */
static void resetTotalEuclidean();

/**
    * @brief Swap this dot and the givenDot
    *
    * @param givenDot
    */
void swap(Dot &givenDot);

/**
    * @brief Return whether this dot is at least delta
distance from givenDot
    *
    * @param givenDot compared dot
    * @param delta
    * @return true the distance between the two point is at
least delta
    * @return false the distance between the two point could
be more than delta
    */
bool bottomBoundDistance(double delta, Dot &givenDot);
};

```



```
#endif
```

3.3 DotCollection.cpp

```
#include "DotCollection.hpp"
#include <algorithm>
#include <iostream>
#include <cmath>

DotCollection::DotCollection(int numberOfDots) :
DotCollection(3, numberOfDots){};
DotCollection::DotCollection(int dimension, int numberOfDots)
: dotDimension(dimension), startIndex(0),
endIndex(numberOfDots), maxCheckedDots(2 * pow(3, dimension))
{
    this->dots = new Dot[numberOfDots];
    for (int i = 0; i < numberOfDots; i++)
    {
        this->dots[i] = Dot(dimension);
    }
    sort();
};

DotCollection::DotCollection(const DotCollection
&parentCollection, int startIndex, int endIndex) :
startIndex(startIndex), endIndex(endIndex),
dotDimension(parentCollection.dotDimension),
maxCheckedDots(parentCollection.maxCheckedDots)
{
    this->dots = parentCollection.dots;
};

DotCollection::~DotCollection(){};

void DotCollection::print()
{
    for (int i = 0; i < length(); i++)
```

```

    {
        at(i).print();
    }
};

void DotCollection::clear()
{
    delete[] this->dots;
}

Dot &DotCollection::at(int index)
{
    return this->dots[startIndex + index];
}

Dot &DotCollection::operator[](int index)
{
    return this->dots[startIndex + index];
}

bool DotCollection::inRange(int index)
{
    return index < length() && index >= 0;
}

int DotCollection::length()
{
    return endIndex - startIndex;
}

void DotCollection::sort()
{
    {
        int length = this->length();
        if (length <= 1)
        {
            return;
        }

        switch (length)
        {

```

```

case 2:
{
    if (at(1) < at(0))
    {
        at(1).swap(at(0));
    }
}
break;
default:
{
    int rightPointer = length;
    int leftPointer = 0;
    Dot pivot = at(0);

    while (leftPointer < rightPointer && leftPointer <
length && rightPointer >= 0)
    {
        do
        {
            leftPointer++;
        } while (at(leftPointer) < pivot && leftPointer <
length - 1);

        do
        {
            rightPointer--;
        } while (at(rightPointer) > pivot && rightPointer >
0);

        at(leftPointer).swap(at(rightPointer));
    }

    at(leftPointer).swap(at(rightPointer));
    at(rightPointer).swap(at(0));

    DotCollection dcLeft = createSubCollection(0,
rightPointer);
    DotCollection dcRight =
createSubCollection(rightPointer + 1, length);
    dcLeft.sort();

```

```

        dcRight.sort();
    }
    break;
}
}

DotCollection DotCollection::createSubCollection(int
startIndex, int endIndex)
{
    return DotCollection(*this, this->startIndex + startIndex,
this->startIndex + endIndex);
}

pair<DotCollection, DotCollection>
DotCollection::createCollectionWithinMiddle(double delta)
{
    int len = length();
    int middleIdx = len / 2;
    double median = at(middleIdx).getCoordinateAt(0);
    int startIdx = middleIdx;
    for (int i = 0; i < middleIdx; i++)
    {
        if (median - at(i)[0] <= delta)
        {
            startIdx = i;
            break;
        }
    }

    int endIdx;
    for (endIdx = middleIdx + 1; endIdx < len; endIdx++)
    {
        if (at(endIdx)[0] - median > delta)
        {
            break;
        }
    }

    return pair<DotCollection,
DotCollection>(createSubCollection(startIdx, middleIdx),
createSubCollection(middleIdx, endIdx));
}

```

```

}

ClosestPairData DotCollection::getClosestPair()
{
    int len = length();
    if (len <= 3)
    {
        return getClosestPairBruteForce();
    }

    int middleIdx = len / 2;
    ClosestPairData closeLeft = createSubCollection(0,
middleIdx).getClosestPair();
    ClosestPairData closeRight = createSubCollection(middleIdx,
len).getClosestPair();
    ClosestPairData &closest = closeLeft.getDistance() <
closeRight.getDistance() ? closeLeft : closeRight;
    double delta = closest.getDistance();

    pair<DotCollection, DotCollection> aroundMiddle =
createCollectionWithinMiddle(delta);

    for (int i = 0; i < aroundMiddle.first.length(); i++)
    {
        long count = 0;
        Dot &left = aroundMiddle.first[i];
        for (int j = 0; j < aroundMiddle.second.length(); j++)
        {
            Dot &right = aroundMiddle.second[j];
            if (!left.bottomBoundDistance(delta, right))
            {
                double distance = left.getDistance(right);
                if (distance < delta)
                {
                    delta = distance;
                    closest = ClosestPairData(&left, &right,
distance);

                    count++;
                    if (count == maxCheckedDots)
                        break;
                }
            }
        }
    }
}

```

```

    }

    }

}

return closest;
};

ClosestPairData DotCollection::getClosestPairBruteForce()
{
    int len = length();

    switch (len)
    {
    case 1:
    {
        throw "Can't calculate distance of 1 dot.";
    }
    break;
    case 2:
    {

        Dot &first = at(0);
        Dot &second = at(1);
        return ClosestPairData(&first, &second,
first.getDistance(second));
    }
    break;
    default:
    {
        ClosestPairData closest(NULL, NULL, INFINITY);
        for (int i = 0; i < len - 1; i++)
        {
            Dot &first = at(i);
            for (int j = i + 1; j < len; j++)
            {
                Dot &second = at(j);
                double distance = first.getDistance(second);
                if (closest.getDistance() > distance)

```

```

        closest = ClosestPairData(&first, &second,
distance);
    }
}
return closest;
}
break;
}
}

```

3.4 DotCollection.hpp

```

#include <vector>
#include <chrono>
#include "../Dot/Dot.hpp"
#include "../ClosestPairData/ClosestPairData.hpp"

using namespace std;

#ifndef DOT_COLLECTION
#define DOT_COLLECTION
class DotCollection
{
private:
    Dot *dots;
    const int startIndex;
    const int endIndex;
    const int dotDimension;
    const long maxCheckedDots;

public:
    /**
     * @brief Construct a new Dot Collection object
     *
     * @param numberOfDots
     */
    DotCollection(int numberOfDots);

```

```

/**
 * @brief Construct a new Dot Collection object
 *
 * @param parentCollection
 * @param startIndex
 * @param endIndex
 */
DotCollection(const DotCollection &parentCollection, int
startIndex, int endIndex);

/**
 * @brief Construct a new Dot Collection object
 *
 * @param dimension
 * @param numberOfDots
 */
DotCollection(int dimension, int numberOfDots);

/**
 * @brief Destroy the Dot Collection object
 *
 */
~DotCollection();

/**
 * @brief Print the dots
 *
 */
void print();

/**
 * @brief Delete the array allocated to for the collection
 *
 */
void clear();

/**
 * @brief Return the element at the given index. Undefined
behaviour when index > endIndex - startIndex
 */

```



```

    * @param index
    * @return Dot
    */
    Dot &at(int index);

    /**
     * @brief Return the element at the given index. Undefined
behaviour when index > endIndex - startIndex
     *
     * @param index
     * @return Dot
     */
    Dot &operator[] (int index);

    /**
     * @brief Determine if the index is within the range of the
collection or not
     *
     * @param index
     * @return true index is within range
     * @return false index is not within range
     */
    bool inRange(int index);

    /**
     * @brief Return the number of dots in the dot collection
     *
     * @return int
     */
    int length();

    /**
     * @brief Sort the dots inside of the dots collection
     *
     */
    void sort();

    /**

```

```

    * @brief Create a sob collection of the dot collection
    object (startIndex and endIndex is relative from the
    dotCollection)
    *
    * @param startIndex
    * @param endIndex
    * @return DotCollection&
    */
    DotCollection createSubCollection(int startIndex, int
    endIndex);

    /**
    * @brief Create two dot collections in range of middle at
    delta distance (left and right)
    *
    * @param delta sparsity
    * @return pair<DotCollection, DotCollection> left
    */
    pair<DotCollection, DotCollection>
    createCollectionWithinMiddle(double delta);

    ClosestPairData getClosestPair();

    ClosestPairData getClosestPairBruteForce();
};

#endif

```

3.5 ClosestPairData.cpp

```

#include "ClosestPairData.hpp"

ClosestPairData::ClosestPairData(Dot *dot1, Dot *dot2, double
distance)
{
    firstDot = dot1;
    secondDot = dot2;
}

```

```

        this->distance = distance;
    }

ClosestPairData::~~ClosestPairData()
{
}

Dot &ClosestPairData::getFirstDot() const
{
    return *firstDot;
}

Dot &ClosestPairData::getSecondDot() const
{
    return *secondDot;
}

double ClosestPairData::getDistance() const
{
    return distance;
}

```

3.6 ClosestPairData.hpp

```

#include "../Dot/Dot.hpp"

#ifndef CLOSEST_PAIR_DATA
#define CLOSEST_PAIR_DATA

class ClosestPairData
{
private:
    Dot *firstDot;
    Dot *secondDot;
    double distance;

public:

```

```

/**
 * @brief Construct a new Closest Pair Data object
 *
 */
ClosestPairData(Dot *firstDot, Dot *secondDot, double
distance);

/**
 * @brief Destroy the Closest Pair Data object
 *
 */
~ClosestPairData();

/**
 * @brief Get the first dot object
 *
 * @return Dot
 */
Dot &getFirstDot() const;

/**
 * @brief Get the second dot object
 *
 * @return Dot
 */
Dot &getSecondDot() const;

/**
 * @brief Get the distance between dots
 *
 * @return distance
 */
double getDistance() const;
};

#endif

```

3.7 visualizer.cpp

```
#include "visualizer.hpp"
#include "../matplotlibcpp.h"

namespace plt = matplotlibcpp;

void visualizeDots(DotCollection &dc, ClosestPairData
&closest)
{
    // create figure
    auto fig = plt::figure();
    plt::subplot("3d");

    // vector of coordinates
    vector<double> x, y, z;
    vector<double> xclose, yclose, zclose;
    for (int i = 0; i < dc.length(); i++)
    {
        Dot &dot = dc[i];

        // if not two closest dots, push to main vector
        if (&dot != &closest.getFirstDot() && &dot !=
&closest.getSecondDot())
        {
            x.push_back(dot[0]);
            y.push_back(dot[1]);
            z.push_back(dot[2]);
        }
        else
        {
            // push to close vector
            xclose.push_back(dot[0]);
            yclose.push_back(dot[1]);
            zclose.push_back(dot[2]);
        }
    }
    // scatter main vector
    plt::scatter(x, y, z, 1.0, {{ "color", "blue" }}, fig, 0.25);
    // scatter close dots
```

```

    plt::scatter(xclose, yclose, zclose, 1.0, {"color",
"red"}}, fig);
    plt::plot3(xclose, yclose, zclose, {"color", "red"}}, fig);

    plt::xlabel("X");
    plt::ylabel("Y");
    plt::set_zlabel("Z");

    plt::show();
}

```

3.8 visualizer.hpp

```

#ifndef VISUALIZER
#define VISUALIZER
#include "../DotCollection/DotCollection.hpp"
#include "../ClosestPairData/ClosestPairData.hpp"

void visualizeDots(DotCollection &dc, ClosestPairData
&closest);

#endif

```

3.9 main.cpp

```

#include "Dot/Dot.hpp"
#include "DotCollection/DotCollection.hpp"
#include "Visualizer/visualizer.hpp"
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int dimension;
    cout << "Enter the dimension of the space : ";
    cin >> dimension;
    int pointCount;
    cout << "Enter the number of points : ";

```

```

cin >> pointCount;
cout << endl;
DotCollection dc(dimension, pointCount);

using namespace chrono;
auto start = high_resolution_clock::now();
ClosestPairData closest = dc.getClosestPair();
auto end = high_resolution_clock::now();

auto duration =
chrono::duration_cast<chrono::microseconds>(end - start);
cout << "Devide et Impera\n";
cout << (double)duration.count() / 1e6 << " seconds" <<
endl;
cout << "Number of Distance Calculation : " <<
Dot::getTotalEuclidean() << endl;

closest.getFirstDot().print();
closest.getSecondDot().print();
cout << closest.getDistance() << endl;

Dot::resetTotalEuclidean();

start = high_resolution_clock::now();
ClosestPairData closestBrute =
dc.getClosestPairBruteForce();
end = high_resolution_clock::now();
cout << endl;
duration = chrono::duration_cast<chrono::microseconds>(end
- start);
cout << "Smol brain O(n^2)\n";
cout << (double)duration.count() / 1e6 << " seconds" <<
endl;
cout << "Number of Distance Calculation : " <<
Dot::getTotalEuclidean() << endl;

closest.getFirstDot().print();
closest.getSecondDot().print();
cout << closest.getDistance() << endl;

```

```
if (dimension == 3)
{
    visualizeDots(dc, closest);
}
return 0;
}
```


BAB III

EXPERIMEN

CLI

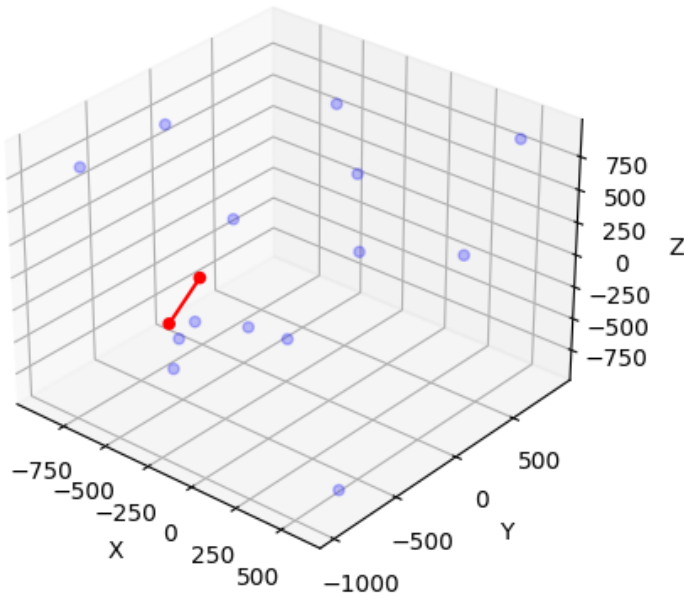
Keluaran ini dilakukan menggunakan Thinkpad X270, i5-6300U, 8GB of DDR4 RAM.

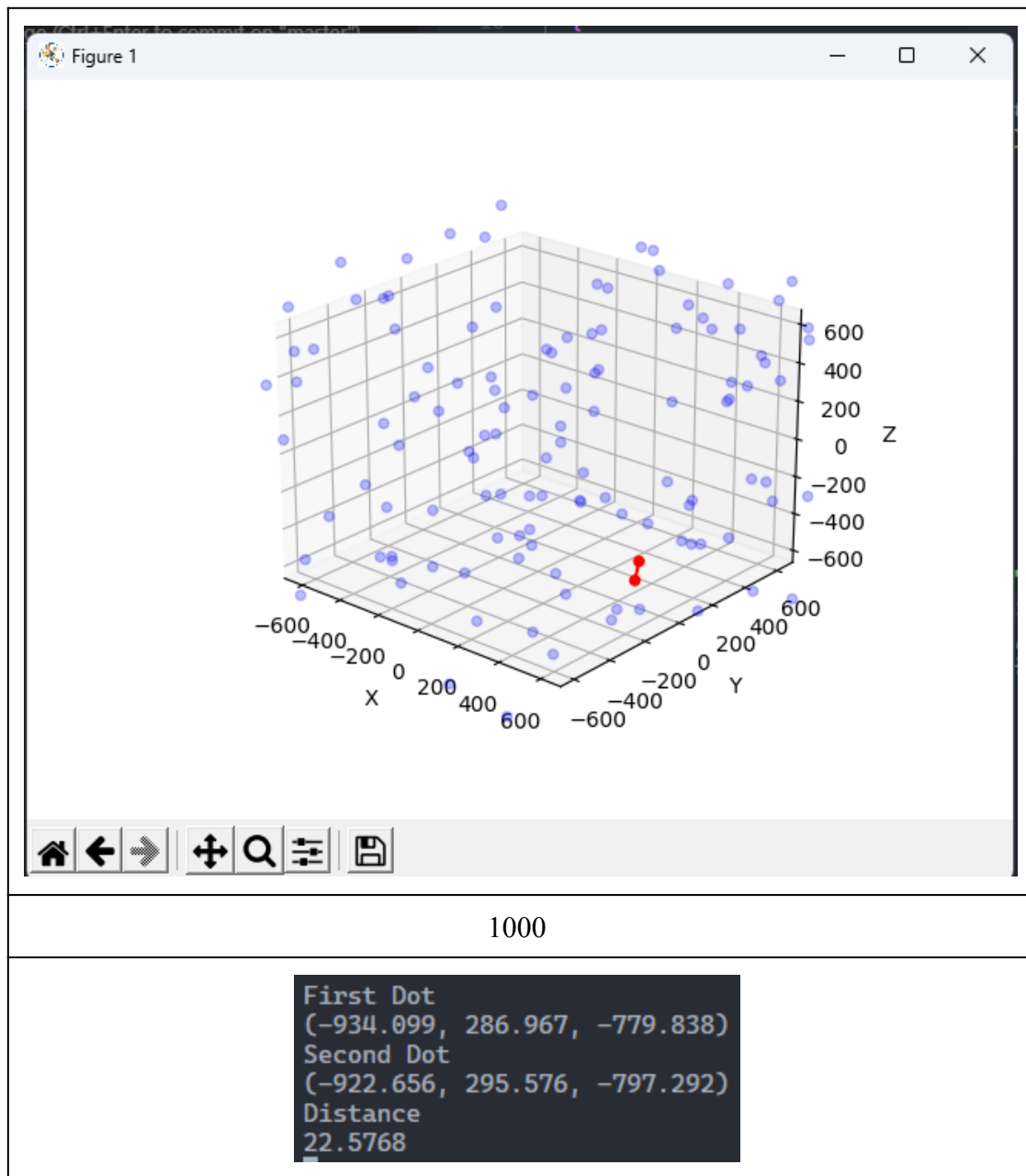
Jumlah titik	Dimensi
16	3
<pre>Enter the dimension of the space : 3 Enter the number of points : 16 Devide et Impera 5.3e-05 seconds Number of Distance Calculation : 14 First Dot (354.952, -227.329, 180.645) Second Dot (500.988, -280.981, 403.839) Distance 272.067 Smol brain O(n^2) 0.000141 seconds Number of Distance Calculation : 120 First Dot (354.952, -227.329, 180.645) Second Dot (500.988, -280.981, 403.839) Distance 272.067</pre>	
1000	3

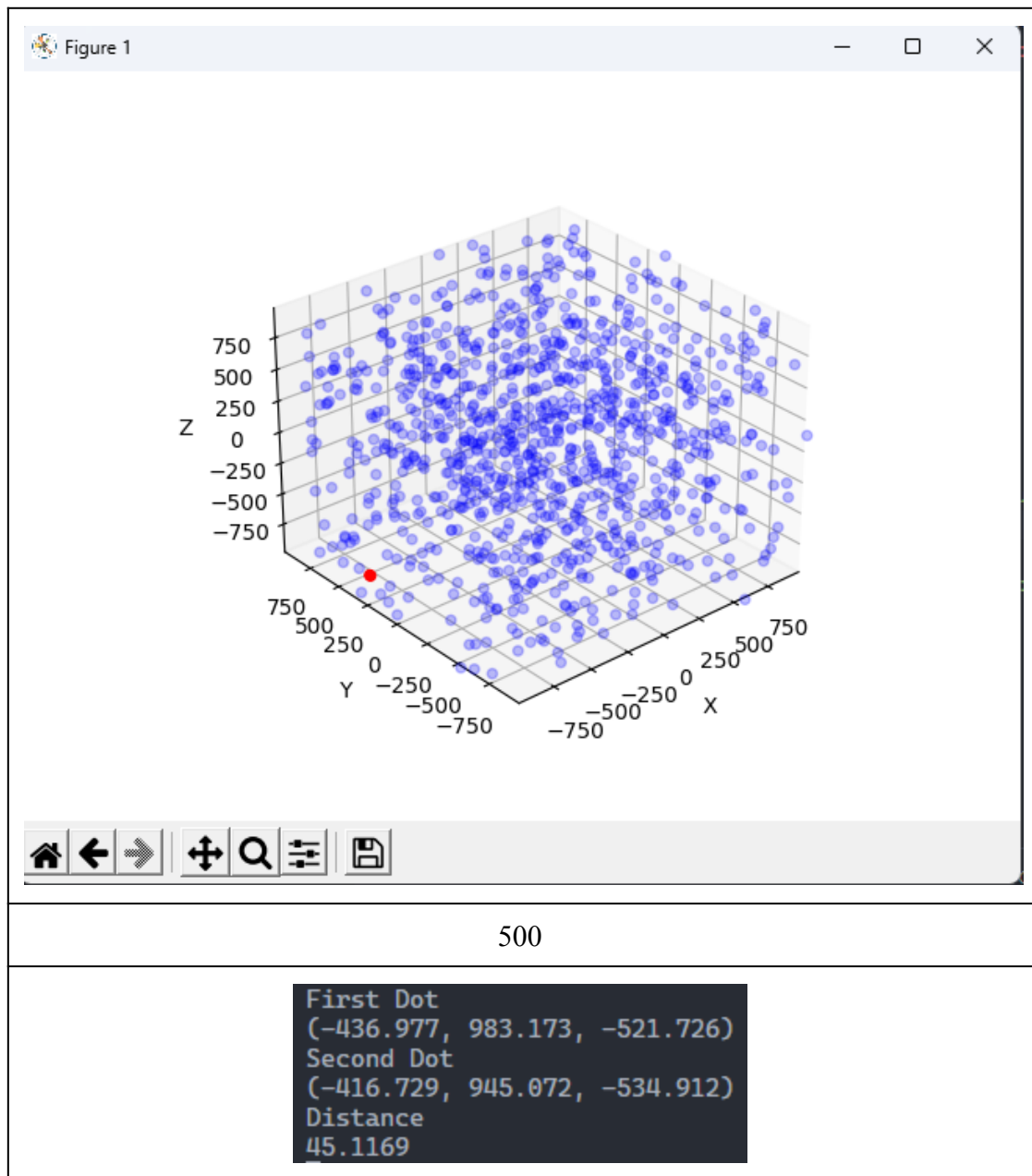
<pre> Enter the dimension of the space : 3 Enter the number of points : 10000 Devide et Impera 0.048522 seconds Number of Distance Calculation : 17967 First Dot (245.09, 629.582, -930.089) Second Dot (245.317, 622.247, -935.562) Distance 9.15416 Smol brain O(n^2) 7.51376 seconds Number of Distance Calculation : 49995000 First Dot (245.09, 629.582, -930.089) Second Dot (245.317, 622.247, -935.562) Distance 9.15416 </pre>	
1000	12
<pre> Enter the dimension of the space : 12 Enter the number of points : 1000 Devide et Impera 0.06313 seconds Number of Distance Calculation : 44125 First Dot (-988.88, 455.058, -182.337, -365.099, 713.704, -424.011, -209.196, 883.61, 53.9424, -347.127, 606.611, -497.733) Second Dot (-943.626, -125.521, -293.71, -652.423, 865.11, -308.962, -125.649, 791.965, -253.298, -751.336, 322.656, -997.847) Distance 1036.41 Smol brain O(n^2) 0.281239 seconds Number of Distance Calculation : 499500 First Dot (-988.88, 455.058, -182.337, -365.099, 713.704, -424.011, -209.196, 883.61, 53.9424, -347.127, 606.611, -497.733) Second Dot (-943.626, -125.521, -293.71, -652.423, 865.11, -308.962, -125.649, 791.965, -253.298, -751.336, 322.656, -997.847) Distance 1036.41 </pre>	
1000	8

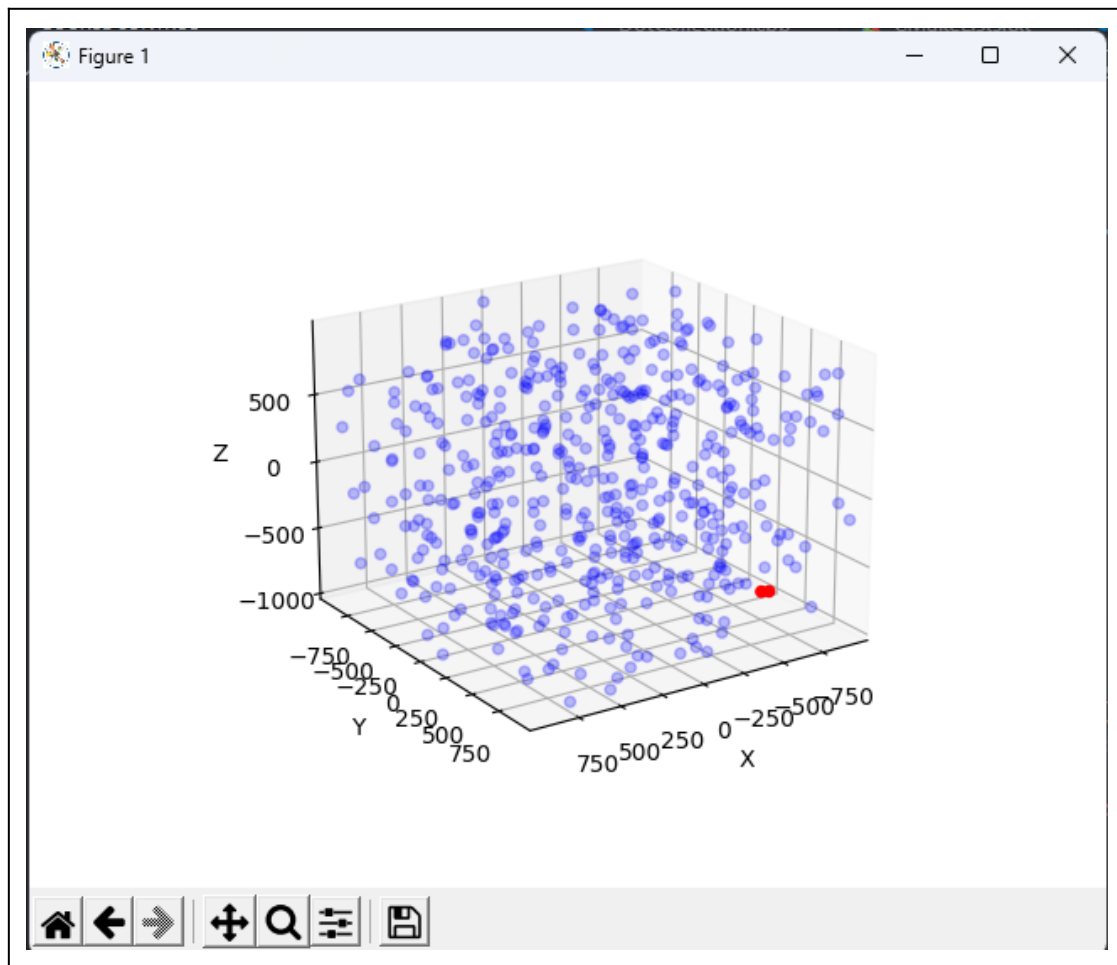
<pre> Enter the dimension of the space : 8 Enter the number of points : 1000 Devide et Impera 0.047777 seconds Number of Distance Calculation : 10216 First Dot (-425.035, 583.565, 680.876, -68.308, 476.742, 913.854, 416.613, -525.741) Second Dot (-386.149, 937.441, 694.124, 264.313, 504.674, 843.704, 562.965, -730.179) Distance 553.595 Smol brain O(n^2) 0.20381 seconds Number of Distance Calculation : 499500 First Dot (-425.035, 583.565, 680.876, -68.308, 476.742, 913.854, 416.613, -525.741) Second Dot (-386.149, 937.441, 694.124, 264.313, 504.674, 843.704, 562.965, -730.179) Distance 553.595 </pre>	
5000	6
<pre> Enter the dimension of the space : 6 Enter the number of points : 5000 Devide et Impera 0.10528 seconds Number of Distance Calculation : 19058 First Dot (-164.1, -147.727, -930.34, 68.3654, 84.9259, 692.096) Second Dot (-155.045, -25.8271, -938.223, 108.145, 62.4191, 666.912) Distance 133.143 Smol brain O(n^2) 3.47385 seconds Number of Distance Calculation : 12497500 First Dot (-164.1, -147.727, -930.34, 68.3654, 84.9259, 692.096) Second Dot (-155.045, -25.8271, -938.223, 108.145, 62.4191, 666.912) Distance 133.143 </pre>	

Bonus: Visualisasi 3D

Jumlah titik
16
<div data-bbox="598 526 1121 739"> <pre> First Dot (-575.489, -497.222, -434.301) Second Dot (-512.671, -331.101, -131.221) Distance 351.283 </pre> </div> <div data-bbox="316 743 1407 1664">  </div>
128
<div data-bbox="619 1769 1099 1971"> <pre> First Dot (352.85, 178.913, -553.931) Second Dot (366.419, 138.968, -633.527) Distance 90.0852 </pre> </div>







BAB IV

HASIL

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

LAMPIRAN

S. Suri. *Closest Pair Problem*. UC Santa Barbara. Diakses pada 27 Februari 2023, dari <https://sites.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf>

Sycorax (<https://stats.stackexchange.com/users/22311/sycorax>), Why is Euclidean distance not a good metric in high dimensions?, URL (version: 2022-02-14): <https://stats.stackexchange.com/q/99191>

Aggarwall C. C., . Hinneburg, A., Keim, D. A., On the Surprising Behavior of Distance Metrics in High Dimensional Space. Institute of Computer Science, University of Halle Kurt-Mothes-Str.1, 06120 Halle (Saale), Germany. Diakses pada 1 Maret 2023, dari <https://bib.dbvis.de/uploadedFiles/155.pdf>

[Link Repository Program](#)