

IF2211 Strategi Algoritma

**PENCARIAN PASANGAN TITIK TERDEKAT PADA
RUANG TIGA DIMENSI DENGAN ALGORITMA *DIVIDE
AND CONQUER***

Laporan Tugas Kecil 2

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



Oleh

Akbar Maulana Ridho

13521093

Chiquita Ahsanunnisa

13521129

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2023**

DAFTAR ISI

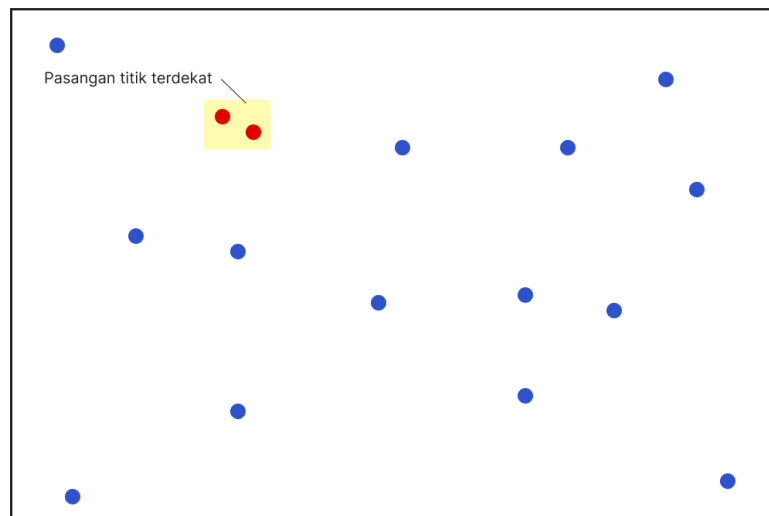
DAFTAR ISI	1
BAB I	
DESKRIPSI MASALAH	3
1.1 Permasalahan Pencarian Pasangan Titik Terdekat	3
1.2 Spesifikasi Program	4
1.2.1 <i>Input</i>	4
1.2.2 <i>Output</i>	4
1.2.3 Bonus	4
BAB II	
LANDASAN TEORI	5
2.1 Jarak <i>Euclidean</i>	5
2.2 Algoritma	5
2.3 Algoritma <i>Divide and Conquer</i>	5
BAB III	
IMPLEMENTASI	8
3.1 Algoritma Pengurutan Larik dengan Metode <i>Quick Sort</i>	8
3.2 Algoritma Pencarian Pasangan Titik Terdekat dengan Memanfaatkan Algoritma <i>Brute Force</i>	8
3.3 Algoritma Pencarian Pasangan Titik Terdekat dengan Memanfaatkan Algoritma <i>Divide and Conquer</i>	9
3.4 Implementasi Algoritma	10
3.4.1 Struktur Data	11
3.4.2 <i>Generator</i> Titik Acak	11
3.4.2 <i>Input dan Output</i>	13
3.4.3 Perhitungan Jarak <i>Euclidean</i>	15
3.4.4 Pencarian Pasangan Titik Terdekat dengan Algoritma <i>Brute Force</i>	16
3.4.5 Pengurutan Larik dengan <i>Quick Sort</i>	19
3.4.6 Pencarian Pasangan Titik Terdekat dengan Algoritma <i>Divide and Conquer</i>	21
3.4.7 Program Utama	23
3.4.8 <i>Visualizer</i>	26
BAB IV	
UJI COBA	28
4.1 Spesifikasi Komputer Uji Coba	28
4.2 Hasil Uji Coba	28
4.2.1 Uji Coba dengan 16 Titik pada Ruang Tiga Dimensi	28
4.2.2 Uji Coba dengan 64 Titik pada Ruang Tiga Dimensi	28

4.2.3 Uji Coba dengan 128 Titik pada Ruang Tiga Dimensi	29
4.2.4 Uji Coba dengan 1000 Titik pada Ruang Tiga Dimensi	29
4.2.5 Uji Coba Lainnya	30
4.3 Analisis Hasil Uji Coba	31
LAMPIRAN	34
Lampiran 1 <i>Link Repository</i> GitHub	34
Lampiran 2 Tampilan Program	34
Lampiran 3 Tabel <i>Check List</i> Poin	36
DAFTAR REFERENSI	37

BAB I DESKRIPSI MASALAH

1.1 Permasalahan Pencarian Pasangan Titik Terdekat

Permasalahan pencarian pasangan titik terdekat adalah salah satu permasalahan *computational geometry* yang cukup populer. Inti dari permasalahan ini adalah “Diberikan sebanyak n titik pada ruang berdimensi k . Bagaimana cara untuk mencari pasangan titik yang memiliki jarak terdekat?”



Gambar 1.1.1 Ilustrasi Permasalahan Pencarian Pasangan Titik Terdekat
(Sumber: Dokumentasi Penulis)

Permasalahan ini dapat diselesaikan dengan berbagai strategi. Strategi paling naif adalah dengan menghitung jarak antartitik dari seluruh kemungkinan pasangan titik dan mencari nilai minimumnya. Namun, strategi ini memiliki kompleksitas waktu $O(kn^2)$, yang berarti strategi ini relatif tidak efektif. Oleh karena itu, dibutuhkan strategi lain yang lebih efektif. Strategi lain yang dapat digunakan adalah dengan algoritma *divide and conquer*.

Tugas mahasiswa adalah mengimplementasikan algoritma *divide and conquer* untuk menyelesaikan permasalahan pencarian pasangan titik terdekat sesuai dengan spesifikasi yang diberikan. Namun, pada tugas ini, ruang yang digunakan dibatasi, yaitu ruang tiga dimensi. Adapun penyelesaian permasalahan untuk ruang multidimensi yang lebih tinggi merupakan kriteria bonus.

1.2 Spesifikasi Program

Program yang dibangun memanfaatkan algoritma *divide and conquer* untuk mencari pasangan titik terdekat. Program ditulis dalam salah satu bahasa di antara C/C++/Java/Python/Golang/Ruby/Perl. Berikut adalah detail dari spesifikasi program yang harus dibuat.

1.2.1 Input

Pengguna pertama memasukkan banyaknya titik (n). Kemudian program akan membangkitkan titik-titik secara acak dalam koordinat titik.

1.2.2 Output

Program menampilkan:

1. Pasangan titik yang jarak antara keduanya paling dekat beserta nilai jarak terdekat tersebut
2. Banyaknya operasi perhitungan jarak *euclidean*
3. Waktu eksekusi program dalam detik (beserta spesifikasi komputer yang digunakan)

1.2.3 Bonus

Berikut adalah spesifikasi bonus dari program yang dibangun.

1. Visualisasi seluruh titik pada bidang tiga dimensi
2. Generalisasi solusi untuk menyelesaikan permasalahan yang sama pada ruang multidimensi

BAB II LANDASAN TEORI

2.1 Jarak *Euclidean*

Jarak yang dimaksud pada bagian sebelumnya adalah jarak *euclidean*. Jarak *euclidean* antara dua titik $p(p_1, p_2, \dots, p_k)$ dan $q(q_1, q_2, \dots, q_k)$ pada ruang berdimensi k dinyatakan dengan notasi $d(p, q)$ yang dapat dihitung dengan rumus di bawah ini.

$$d(p, q) = \sqrt{\sum_{i=1}^k (p_i - q_i)^2} \dots (1)$$

Rumus di atas juga dapat dijabarkan menjadi seperti di bawah ini.

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_k - q_k)^2} \dots (2)$$

2.2 Algoritma

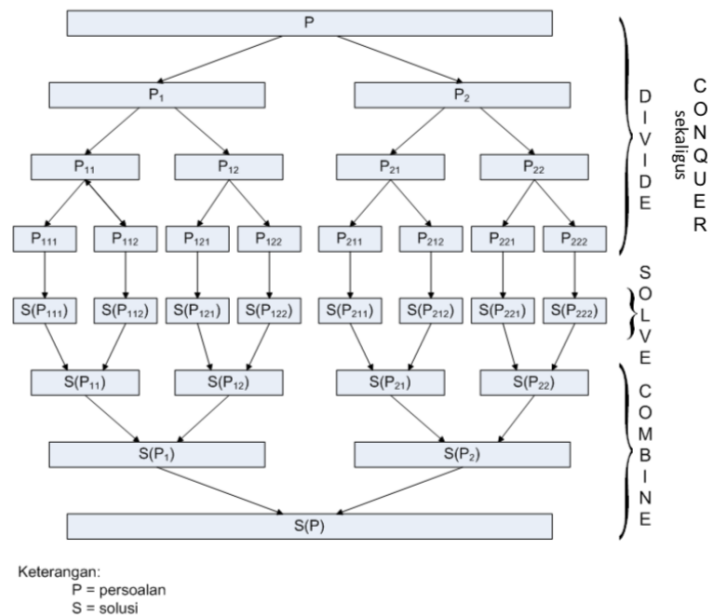
Algoritma adalah urutan langkah-langkah yang jelas dan terstruktur dalam menyelesaikan suatu permasalahan.

2.3 Algoritma *Divide and Conquer*

Algoritma *divide and conquer* adalah salah satu teknik dalam mendesain algoritma. Algoritma ini terbagi menjadi tiga langkah utama, yaitu sebagai berikut.

1. Membagi (*divide*) persoalan menjadi beberapa upapersoalan yang memiliki kemiripan dengan persoalan semula namun berukuran lebih kecil (idealnya berukuran hampir sama)
2. Menyelesaikan (*conquer* atau *solve*) masing-masing upapersoalan (secara langsung jika sudah berukuran kecil atau secara rekursif jika masih berukuran besar)
3. Menggabungkan solusi masing-masing upapersoalan sehingga membentuk solusi persoalan semula

Keseluruhan langkah di atas dapat diilustrasikan melalui gambar di bawah ini.



Gambar 2.3.1 Ilustrasi Algoritma *Divide and Conquer*

(Sumber: <https://informatika.stei.itb.ac.id/>)

Algoritma *divide and conquer* memiliki skema umum sebagai berikut.

```

procedure DIVIDEandCONQUER(input P : problem, n : integer)
{ Menyelesaikan persoalan P dengan algoritma divide and conquer
  Masukan: masukan persoalan P berukuran n
  Luaran: solusi dari persoalan semula }

Deklarasi
  r : integer

Algoritma
  if n ≤ n0 then {ukuran persoalan P sudah cukup kecil }
    SOLVE persoalan P yang berukuran n ini
  else
    DIVIDE menjadi r upa-persoalan, P1, P2, ..., Pr, yang masing-masing
    berukuran n1, n2, ..., nr
    for masing-masing P1, P2, ..., Pr, do
      DIVIDEandCONQUER(Pi, ni)
    endfor
    COMBINE solusi dari P1, P2, ..., Pr menjadi solusi persoalan semula
  endif

```

Berdasarkan skema di atas, algoritma *divide and conquer* memiliki kompleksitas waktu:

$$T(n) = g(n) \text{ untuk } n \leq n_0$$

$$T(n) = T(n_1) + T(n_2) + \dots + T(n_r) \text{ untuk } n > n_0 \dots (3)$$

dengan $T(n)$ adalah kompleksitas waktu penyelesaian persoalan P yang berukuran n , $g(n)$ adalah kompleksitas waktu penyelesaian jika n sudah berukuran kecil, $T(n_1) + T(n_2) + \dots + T(n_r)$ adalah gabungan kompleksitas waktu dari setiap upapersoalan, serta $f(n)$ adalah kompleksitas waktu untuk menggabungkan solusi dari masing-masing upapersoalan menjadi solusi persoalan semula.

2.4 Teorema Master

Teorema Master adalah teorema yang digunakan untuk menentukan notasi asimtotik dari kompleksitas waktu yang berbentuk relasi rekurens. Dengan Teorema Master, relasi rekurens tersebut tidak perlu diselesaikan secara iteratif.

Misalkan $T(n)$ adalah fungsi monoton menaik yang memenuhi relasi rekurens

$$T(n) = aT\left(\frac{n}{b}\right) + cn^d \dots (4)$$

dengan $n = b^k$, $k = 1, 2, \dots$, $a \geq 1$, $b \geq 2$, dan $d \geq 0$. Menurut Teorema Master, dapat disimpulkan bahwa

$$\begin{aligned} T(n) &= O(n^d) \text{ jika } a < b^d \\ T(n) &= O(n^d \log n) \text{ jika } a = b^d \\ T(n) &= O(n^{\log_b a}) \text{ jika } a > b^d \dots (5) \end{aligned}$$

Teorema Master sering digunakan untuk menentukan notasi asimtotik dari kompleksitas waktu pada algoritma *divide and conquer*. Hal ini terjadi karena pada umumnya algoritma *divide and conquer* diimplementasikan dalam skema rekursif.

BAB III IMPLEMENTASI

3.1 Algoritma Pengurutan Larik dengan Metode *Quick Sort*

Untuk mengimplementasikan algoritma *divide and conquer* pada permasalahan pencarian pasangan titik terdekat, dibutuhkan proses pengurutan larik titik berdasarkan sumbu tertentu. Proses pengurutan larik ini diimplementasikan dengan memanfaatkan algoritma *divide and conquer*, yaitu dengan metode *quick sort*.

Metode *quick sort* sangat bergantung pada algoritma partisi larik. Awalnya, dipilih sebuah nilai dari larik sebagai *pivot*. Kemudian, algoritma ini membagi (mempartisi) larik menjadi dua bagian, yaitu bagian di sebelah kiri *pivot* dan sebelah kanan *pivot*. Bagian sebelah kiri diperuntukkan bagi elemen larik yang bernilai lebih kecil daripada *pivot*, sedangkan bagian sebelah kanan diperuntukkan bagi elemen larik yang bernilai lebih besar daripada *pivot*.

Berikut adalah langkah-langkah pada algoritma partisi untuk larik A .

1. Pilih $x \in \{A[1], A[2], \dots, A[n]\}$ sebagai *pivot*. Pada implementasi yang dilakukan, dipilih elemen tengah larik sebagai *pivot*.
2. Pindai larik dari kiri hingga ditemukan elemen $A[p] \geq x$.
3. Pindai larik dari kanan hingga ditemukan elemen $A[q] \leq x$.
4. Tukar posisi $A[p]$ dengan $A[q]$.
5. Ulangi langkah 2 dari posisi $p + 1$ dan langkah 3 dari posisi $q - 1$ hingga pemindaian bertemu di tengah larik ($p \geq q$).

Berikut adalah langkah-langkah pada metode *quick sort*.

1. Partisi larik sehingga elemen yang menjadi *pivot* sudah berada di urutan yang tepat.
2. Perlakukan upalarik di sebelah kiri dan kanan *pivot* sebagai sebuah larik tunggal. Lakukan langkah 1 dan 2 secara rekursif.
3. Basis dari rekursi terjadi saat upalarik yang didapat berisi nol atau satu elemen. Pada kasus basis, tidak dilakukan apa-apa.

Metode ini memiliki kompleksitas waktu asimtotik $O(n \log n)$.

3.2 Algoritma Pencarian Pasangan Titik Terdekat dengan Memanfaatkan Algoritma *Brute Force*

Permasalahan pencarian pasangan titik terdekat sebenarnya dapat diselesaikan dengan algoritma *brute force*. Dengan algoritma *brute force*, semua kemungkinan pasangan titik akan dicek lalu dicari jarak minimumnya.

Berikut adalah *pseudocode* untuk pencarian pasangan titik terdekat dengan algoritma *brute force*.

```

function brute_force(points) → closest pair of point
  for i in range(num_of_points - 1)
    for j in range(i + 1, num_of_points)
      current_min ← euclidean_distance(points[i], points[j])
      if current_min < min then
        min ← current_min
        pair ← (points[i], points[j])
  → pair

```

Algoritma ini menghitung setiap kemungkinan pasangan titik untuk mencari jarak terdekat. Bila terdapat n buah titik, maka akan membutuhkan ${}_nC_2 = \frac{n(n-1)}{2}$ buah perbandingan, sehingga pendekatan ini memiliki kompleksitas waktu $O(n^2)$.

3.3 Algoritma Pencarian Pasangan Titik Terdekat dengan Memanfaatkan Algoritma *Divide and Conquer*

Permasalahan pencarian pasangan titik terdekat juga dapat diselesaikan dengan algoritma *divide and conquer*. Dengan algoritma *divide and conquer*, semua kemungkinan pasangan titik yang harus dicek dapat dikurangi sehingga lebih efektif.

Berikut adalah skema algoritma pencarian pasangan titik terdekat yang diimplementasikan.

1. Urutkan titik secara membesar berdasarkan komponen sumbu x.
2. Partisi (*divide*) larik titik di median sehingga dihasilkan dua upalarik titik. Perlakukan masing-masing upalarik sebagai larik tunggal. Cari jarak minimum pada masing-masing upalarik dengan pemanggilan rekursif. Carilah δ , yaitu minimum dari jarak minimum masing-masing upalarik.
3. Satukan solusi dari masing-masing upalarik dengan cara memilih titik-titik yang memiliki jarak komponen sumbu x ke median lebih kecil daripada δ .
4. Lakukan pemanggilan rekursif untuk mencari jarak pasangan titik minimum pada titik-titik yang terpilih pada langkah 3, dengan catatan pencarian diproyeksikan ke komponen sumbu selanjutnya (y, z, ...).
5. Jarak terkecil dari keseluruhan titik adalah minimum dari δ dan jarak pasangan titik minimum yang dihasilkan dari langkah 4.

Berikut adalah *pseudocode* untuk pencarian pasangan titik terdekat dengan algoritma *divide and conquer*.

```

function divide_and_conquer(points, depth) → closest pair of point
  if num_of_points ≤ 3 or last_dimension then

```

```

// base case, ketika hasil proyeksi di dimensi satu atau
// terdapat kurang dari tiga titik
→ brute_force(points)

// sort berdasarkan sumbu tertentu
quicksort(points, depth)

points_s1 ← slice points from start to median point
points_s2 ← slice points from median point to end

// divide
pairs1 ← divide_and_conquer(points_s1, depth)
pairs2 ← divide_and_conquer(points_s2, depth)

delta ← min(pairs1, pairs2)

// conquer
points_s12 ← based on current axis (depth), filter points so
              so that only points that have distance at some
              axis to median are less than delta

// proyeksikan ke dimensi berikutnya terhadap axis sekarang
pairs12 ← divide_and_conquer(points_s12, depth+1)

→ min(delta, pairs12)

```

Berdasarkan algoritma di atas, persamaan kompleksitas waktu untuk pendekatan *divide and conquer* adalah

$$\begin{aligned}
 T(n, d) &= 2T(n/2, d) + U(n, d - 1) + O(n) \\
 &= 2T(n/2, d) + O(n(\log n)^{d-2}) + O(n) \\
 &= O(n(\log n)^{d-1})
 \end{aligned}$$

dengan $2 * T(n/2, d)$ terjadi saat proses *divide* yang membagi data menjadi dua. Lalu, $U(n, d - 1)$ terjadi saat proses *conquer*, yakni memproyeksikan titik-titik kepada salah satu sumbu sehingga menjadi satu dimensi lebih rendah. Terdapat pula proses pengurutan larik titik berdasarkan sumbu dengan kompleksitas $O(n \log n)$, tetapi pada persamaan ini dapat diabaikan.

3.4 Implementasi Algoritma

Algoritma yang telah dipaparkan pada bagian sebelumnya diimplementasikan pada bahasa pemrograman C++. Visualisasi dengan grafik tiga dimensi diimplementasikan pada bahasa pemrograman Python dengan

memanfaatkan library Numpy dan Matplotlib. *Passing* antara bahasa C++ ke Python untuk visualisasi memanfaatkan mekanisme *log* melalui *file* .txt.

3.4.1 Struktur Data

Kode untuk mendeklarasikan struktur data yang digunakan ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan nama *file* *datatypes.h*.

```
datatypes.h

#include <vector>
#include <tuple>

typedef std::vector<double> point_t;
typedef std::vector<point_t> points_t;
typedef std::tuple<point_t, point_t> point_pair_t;
typedef std::vector<point_pair_t> pairs_t;
typedef std::tuple<pairs_t, double> closest_pair_t;
```

3.4.2 Generator Titik Acak

Kode untuk *generator* titik acak ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan dua *file*: *generator.h* sebagai *header* dan *generator.cpp* sebagai implementasi.

```
generator.h

#include "datatypes.h"

/**
 * @brief
 *
 *
 *
 * @param dimensions
 * @param number_of_points
 * @param range
 * @return
 */
points_t generate_points(int dimensions, int number_of_points, int range);

/**
 * Print point to cout
 * @param point
 */
void print_point(point_t point);

/**
```

```

* Print pair result
* @param closest_pair
*/
void print_result(const closest_pair_t& closest_pair);

```

generator.cpp

```

#include "generator.h"
#include <iostream>
#include <iomanip>
#include <random>

using std::cout;
using std::get;
using std::endl;
using std::mt19937_64;
using std::random_device;
using std::setprecision;
using std::uniform_real_distribution;

points_t generate_points(int dimensions, int number_of_points, int
range)
{
    random_device rd;
    mt19937_64 generator(rd());
    uniform_real_distribution<> distributor(-1 * (range / 2.0), range /
2.0);

    points_t points;

    for (int i = 0; i < number_of_points; i++)
    {
        point_t point;
        for (int j = 0; j < dimensions; j++)
        {
            point.push_back(distributor(generator));
        }

        points.push_back(point);
    }

    return points;
}

void print_point(point_t point)
{
    cout << '(' << point[0];
    for (int i = 1; i < point.size(); i++)
    {
        cout << ", " << point[i];
    }
    cout << ')' << endl;
}

```

```

void print_pair(point_pair_t p)
{
    print_point(get<0>(p));
    print_point(get<1>(p));
}

void print_result(const closest_pair_t& closest_pair)
{
    auto &[pairs_list, dist] = closest_pair;

    cout << setprecision(8);

    cout << "Closest Pairs: " << endl;

    for (int i = 0; i < pairs_list.size(); i++){
        cout << "----- " << i + 1 << " ----" << endl;
        print_pair(pairs_list[i]);
    }

    cout << endl
         << "Closest Distance: " << dist << endl;
}

```

3.4.2 Input dan Output

Kode untuk mengurus *input* dan *output* ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan dua *file*: *io.h* sebagai *header* dan *io.cpp* sebagai implementasi.

io.h

```

#include <iostream>
#include <fstream>
#include <string>
#include <filesystem>
#include "datatypes.h"

using namespace std;

/* INPUT */
int input_dimensions();

int input_num_of_points();

/* OUTPUT */
void write_to_file(points_t points, string filename);

void write_to_file(pairs_t pairs, string filename);

```

io.cpp

```

#include "io.h"

/* INPUT */
int input_dimensions()
{
    int dims;

    do
    {
        cout << "Dimensions: ";
        cin >> dims;

        if (dims < 1)
        {
            cout << "Invalid dimensions. Please try again." << endl;
        }
    }
    while (dims < 1);

    return dims;
}

int input_num_of_points()
{
    int num_of_points;

    do
    {
        cout << "Number of points: ";
        cin >> num_of_points;

        if (num_of_points < 2)
        {
            cout << "Invalid number of points. There should be at least
two points. Please try again." << endl;
        }
    }
    while (num_of_points < 2);

    return num_of_points;
}

/* OUTPUT */
void write_to_file(points_t points, string filename)
{
    filesystem::remove(filename);

    ofstream file(filename);
    int dims = points[0].size();
    for (int i = 0; i < points.size(); i++)
    {
        for (int j = 0; j < dims; j++)
        {

```

```

        file << points[i][j];
        if (j != dims - 1)
        {
            file << " ";
        }
    }
    file << "\n";
}

file.close();
}

void write_to_file(pairs_t pairs, string filename)
{
    filesystem::remove(filename);

    ofstream file(filename);
    int dims = get<0>(pairs[0]).size();
    for (int i = 0; i < pairs.size(); i++)
    {
        for (int j = 0; j < dims; j++)
        {
            file << get<0>(pairs[i])[j];
            if (j != dims - 1)
            {
                file << " ";
            }
        }
        file << "\n";

        for (int j = 0; j < dims; j++)
        {
            file << get<1>(pairs[i])[j];
            if (j != dims - 1)
            {
                file << " ";
            }
        }
        file << "\n";
    }

    file.close();
}

```

3.4.3 Perhitungan Jarak *Euclidean*

Kode untuk perhitungan jarak *euclidean* ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan dua *file*: *euclidean.h* sebagai *header* dan *euclidean.cpp* sebagai implementasi.

```
euclidean.h
```



```

#include "datatypes.h"

/**
 * Calculate euclidean distance between two points with d dimension
 * @return
 */
double calculate_euclidean_distance(point_t &, point_t &);

```

euclidean.cpp

```

#include "euclidean.h"
#include <cmath>

double calculate_euclidean_distance(point_t &a, point_t &b)
{
    double squared_sum = 0.0;
    int dimension = a.size();

    for (int i = 0; i < dimension; i++)
    {
        squared_sum += pow((a[i] - b[i]), 2);
    }

    return sqrt(squared_sum);
}

```

3.4.4 Pencarian Pasangan Titik Terdekat dengan Algoritma *Brute Force*

Kode untuk pencarian pasangan titik terdekat dengan dengan algoritma brute force ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan dua file: bfsolver.h sebagai *header* dan bfsolver.cpp sebagai implementasi.

bfsolver.h

```

#include "datatypes.h"

extern int bf_counter;
extern int dnc_counter;

/**
 * Check if two point are equal
 * @param p1
 * @param p2
 * @return
 */
bool compare_point(point_t p1, point_t p2);

/**
 * Check if two point pair are equal
 * @param p1
 * @param p2

```

```

* @return
*/
bool compare_pair(point_pair_t pair1, point_pair_t pair2);

/**
 * Combine unique pair from source to destination
 * @param dest
 * @param source
 */
void combine_pairs(pairs_t &dest, pairs_t &source);

/**
 * Solve closest pair problem with brute force method
 * @param points
 * @param from_dnc
 * @return
 */
closest_pair_t closest_pair_brute_force(points_t &points, bool from_dnc
= false);

```

bfsolver.cpp

```

#include "bfsolver.h"

#include <utility>
#include "euclidean.h"
using namespace std;

int bf_counter = 0;
int dnc_counter = 0;

bool compare_point(point_t p1, point_t p2)
{
    bool equal = true;
    int i = 0;
    int dims = p1.size();

    while (i < dims && equal)
    {
        if (p1[i] != p2[i])
        {
            equal = false;
        }

        i++;
    }

    return equal;
}

bool compare_pair(point_pair_t pair1, point_pair_t pair2)
{
    auto [p11, p12] = std::move(pair1);
    auto [p21, p22] = std::move(pair2);

```

```

        return (compare_point(p11, p21) && compare_point(p12, p22)) ||
               (compare_point(p11, p22) && compare_point(p12, p21));
    }

void combine_pairs(pairs_t &dest, pairs_t &source)
{
    for (auto &src : source)
    {
        int i = 0;
        bool match = false;

        while (i < dest.size() && !match)
        {
            auto to_cmp = dest[i];

            if (compare_pair(src, to_cmp))
            {
                match = true;
            }

            i++;
        }

        if (!match) {
            dest.push_back(src);
        }
    }
}

closest_pair_t    closest_pair_brute_force(points_t    &points,    bool
from_dnc)
{
    int n = points.size();

    double min = 1 << 25;
    pairs_t pairs_list;

    // Iterate every point pair and select smallest pair (brute force)
    // Time complexity O(N^2)
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            double distance = calculate_euclidean_distance(points[i],
points[j]);

            if (from_dnc)
            {
                dnc_counter++;
            }
            else
            {

```

```

        bf_counter++;
    }

    if (distance == min)
    {
        pairs_list.emplace_back(points[i], points[j]);
    }
    else if (distance < min)
    {
        min = distance;
        pairs_list.clear();
        pairs_list.emplace_back(points[i], points[j]);
    }
}

return closest_pair_t{pairs_list, min};
}

```

3.4.5 Pengurutan Larik dengan *Quick Sort*

Kode untuk pengurutan larik titik dengan algoritma *quick sort* ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan dua *file*: *sorter.h* sebagai *header* dan *sorter.cpp* sebagai implementasi.

sorter.h

```

#include "datatypes.h"

/**
 * Part array of points into two parts using the middle element as its
 * pivot
 */
int partition(points_t &points, int depth, int low, int high);

void _quicksort(points_t &points, int depth, int low, int high);

/**
 * Sort points based on some axis with quicksort
 */
void quicksort(points_t &, int);

```

sorter.cpp

```

#include "sorter.h"

int partition(points_t &points, int depth, int low, int high)
{
    double pivot_depth_val;
    int left_ptr, right_ptr;
    point_t temp;

```

```

        pivot_depth_val = points[(low + high) / 2][depth]; // pivot at
middle
        left_ptr = low;
        right_ptr = high;

        while (left_ptr <= right_ptr)
        {
            while (points[left_ptr][depth] < pivot_depth_val)
            {
                left_ptr++;
            }

            while (points[right_ptr][depth] > pivot_depth_val)
            {
                right_ptr--;
            }

            if (left_ptr <= right_ptr)
            {
                temp = points[left_ptr];
                points[left_ptr] = points[right_ptr];
                points[right_ptr] = temp;
                left_ptr++;
                right_ptr--;
            }
        }

        return left_ptr;
    }

void _quicksort(points_t &points, int depth, int low, int high)
{
    int part_idx = partition(points, depth, low, high);

    if (low < part_idx - 1)
    {
        _quicksort(points, depth, low, part_idx - 1);
    }

    if (part_idx < high)
    {
        _quicksort(points, depth, part_idx, high);
    }
}

void quicksort(points_t &points, int depth)
{
    if (points.size() > 1)
    {
        _quicksort(points, depth, 0, points.size() - 1);
    }
}

```

3.4.6 Pencarian Pasangan Titik Terdekat dengan Algoritma *Divide and Conquer*

Kode untuk pencarian pasangan titik terdekat dengan algoritma *divide and conquer* ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan dua file: `dncsolver.h` sebagai *header* dan `dncsolver.cpp` sebagai implementasi.

dncsolver.h
<pre>#include "datatypes.h" /** * Solve closest pair problem with divide and conquer method * @param points * @param depth start from 0, denote current projection axis * @return */ closest_pair_t closest_pair_divide_conquer(points_t &points, int depth);</pre>
dncsolver.cpp
<pre>#include "datatypes.h" #include "dncsolver.h" #include "sorter.h" #include "euclidean.h" #include "bfsolver.h" #include <cmath> closest_pair_t closest_pair_divide_conquer(points_t &points, int depth) { bool lastDims = points[0].size() - 1 == depth; /** * Base case * * Jika banyaknya titik kurang dari sama dengan tiga atau hasil * proyeksi bersisa satu dimensi, * cari solusi pasangan titik terdekat dengan menggunakan metode * brute force */ if (points.size() <= 3 lastDims) { return closest_pair_brute_force(points, true); } /** * Lakukan sort berdasarkan axis tertentu, terurut membesar * Memiliki kompleksitas waktu O(NlogN) */</pre>

```

    quicksort(points, depth);

    int median_idx = points.size() / 2;

    /*
     * Bagi titik menjadi dua bagian, S1 dan S2 berdasarkan median
     */
    auto points_s1 = points_t(points.begin(), points.begin() +
median_idx);
    auto points_s2 = points_t(points.begin() + median_idx + 1,
points.end());

    /*
     * Divide step,
     *
     * panggil fungsi rekursif dengan titik-titik telah dibagi dua
     *
     * Kompleksitas waktu  $2 * T(N/2, d)$  dengan d adalah dimensi
     */
    auto [s1_pairs_list, s1_dist] =
closest_pair_divide_conquer(points_s1, depth);
    auto [s2_pairs_list, s2_dist] =
closest_pair_divide_conquer(points_s2, depth);

    double delta;
    pairs_t pairs_list;

    // cari delta dan pasangan titik terdekat berdasarkan hasil divide

    if (s1_dist < s2_dist)
    {
        delta = s1_dist;
        pairs_list = s1_pairs_list;
    }
    else if (s1_dist > s2_dist)
    {
        delta = s2_dist;
        pairs_list = s2_pairs_list;
    }
    else
    {
        delta = s1_dist;
        pairs_list = s1_pairs_list;
        combine_pairs(pairs_list, s2_pairs_list);
    }

    point_t median_point = points[median_idx];

    // conquer step
    // pada axis tertentu, ambil titik titik yang pada axis tertentu
    memenuhi
    //  $x_i - x_{\text{median}} < \text{delta}$ 

```

```

points_t points_s12;

for (auto &point : points)
{
    if (std::abs(point[depth] - median_point[depth]) < delta)
    {
        points_s12.push_back(point);
    }
}

if (points_s12.size() <= 1)
{
    return closest_pair_t{pairs_list, delta};
}

/*
 * Conquer step
 *
 * for every points that are closer than delta in an axis,
 * project the points into some axis
 *
 * This step have time complexity
 * T(N, d-1)
 */
auto [s12_pairs_list, s12_dist] =
closest_pair_divide_conquer(points_s12, depth + 1);

/*
 * Bandingkan delta dengan titik terdekat pada hasil conquer, lalu
 * pilih jarak dan pasangan titik terdekat
 */
if (s12_dist == delta)
{
    combine_pairs(pairs_list, s12_pairs_list);
    return closest_pair_t{pairs_list, delta};
}
else if (s12_dist < delta)
{
    return closest_pair_t{s12_pairs_list, s12_dist};
}

return closest_pair_t{pairs_list, delta};
}

```

3.4.7 Program Utama

Kode untuk program utama ditulis dalam bahasa C++. Kode terdapat pada direktori src dengan nama *file* main.cpp.

```
main.cpp
```



```

#include <iostream>
#include <chrono>
#include "datatypes.h"
#include "bfsolver.h"
#include "dncsolver.h"
#include "generator.h"
#include "io.h"

using std::cin;
using std::cout;
using std::endl;
using std::chrono::duration;
using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;

bool run(int dimension, int n, bool debug = false)
{
    bf_counter = 0;
    dnc_counter = 0;

    int range = 1e4;
    points_t points1 = generate_points(dimension, n, range);
    if (dimension == 3)
    {
        write_to_file(points1, "../log/points.txt");
    }

    points_t points2 = points1;

    if (debug)
    {
        cout << endl
              << "Dims " << dimension << " num of points " << n <<
endl;
    }

    cout << endl
         << "Divide And Conquer" << endl;
    auto start_dnc = high_resolution_clock::now();
    auto dnc_result = closest_pair_divide_conquer(points1, 0);
    auto stop_dnc = high_resolution_clock::now();
    print_result(dnc_result);

    duration<double> dnc_duration =
duration_cast<duration<double>>(stop_dnc - start_dnc);
    cout << dnc_duration.count() << " seconds elapsed" << endl;
    cout << dnc_counter << " euclidean distance comparisons" << endl;

    bf_counter = 0;

    cout << endl
         << "Brute Force" << endl;
    auto start_bf = high_resolution_clock::now();

```

```

    auto bf_result = closest_pair_brute_force(points2);
    auto stop_bf = high_resolution_clock::now();
    print_result(bf_result);

    duration<double> bf_duration =
duration_cast<duration<double>>(stop_bf - start_bf);
    cout << bf_duration.count() << " seconds elapsed" << endl;
    cout << bf_counter << " euclidean distance comparisons" << endl;

    if (dimension == 3 && !debug)
    {
        auto [pairs_list, dist] = dnc_result;
        write_to_file(pairs_list, "../log/pairs_list.txt");
    }

    auto [dncres, dnclist] = dnc_result;
    auto [bfres, bfdist] = bf_result;

    return dnclist == bfdist;
}

int main()
{
    // DEBUG PURPOSES
    bool DEBUG = false;

    if (DEBUG)
    {
        bool shouldStop = false;
        int counter = 0;
        int limit = 100;

        while (!shouldStop)
        {
            int dims[] = {1, 2, 3, 4, 5};
            int ns[] = {10, 32, 128, 1000, 2000};

            int i = 0;

            while (i < 5 && !shouldStop)
            {
                int j = 0;
                while (j < 5 && !shouldStop)
                {
                    bool correctResult = run(dims[i], ns[j], true);

                    if (!correctResult)
                    {
                        shouldStop = true;
                    }
                    j++;
                    counter++;
                }
            }
        }
    }
}

```

```

        if (counter == limit)
        {
            shouldStop = true;
        }
    }
    i++;
}

if (counter < limit)
{
    cout << "Total " << counter << " run before error" <<
endl;
}
else
{
    cout << "No errors found within " << limit << " run" <<
endl;
}
}
else
{
    int dimensions, num_of_points;

    dimensions = input_dimensions();
    num_of_points = input_num_of_points();

    run(dimensions, num_of_points);
}

return 0;
}

```

3.4.8 Visualizer

Kode untuk *visualizer* ditulis dalam bahasa Python. Kode terdapat pada direktori src dengan nama *file* visualizer.py.

```

visualizer.py

import numpy as np
from numpy.typing import NDArray
import matplotlib.pyplot as plt
import itertools

# read log files
def read_file(path):
    f = open(path, "r")
    points = f.readlines()
    for i in range(len(points)):

```

```

        points[i] = points[i].strip().split()

    points = np.asarray(points).astype(float)
    f.close()

    return points

# visualization in 3D scatter plot
def visualize(points: NDArray, pairs_list: NDArray):
    for point in pairs_list:
        new_point = points
        for i in range(len(points)):
            if np.array_equal(points[i], point):
                new_point = np.delete(points, i, axis=0)

        points = new_point

    fig = plt.figure("3D Scatter Plot")
    ax = fig.add_subplot(projection='3d')

    ax.scatter(points[:, 0], points[:, 1], points[:, 2], marker='o')

    # change color for each pair of closest points
    colors = itertools.cycle(["tab:orange", "tab:green", "tab:red",
                              "tab:purple", "tab:brown", "tab:pink", "tab:gray", "tab:olive",
                              "tab:cyan", "r", "g", "b", "c", "m", "y", "k"])
    for i in range(0, len(pairs_list), 2):
        pairs_color = next(colors)
        ax.scatter(pairs_list[i, 0], pairs_list[i, 1],
                   pairs_list[i, 2], marker='^', color=pairs_color)
        ax.scatter(pairs_list[i + 1, 0], pairs_list[i + 1, 1],
                   pairs_list[i + 1, 2], marker='^', color=pairs_color)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    plt.show()

# main program for visualization
if __name__ == "__main__":
    points = read_file("../log/points.txt")
    pairs_list = read_file("../log/pairs_list.txt")
    visualize(points, pairs_list)

```

BAB IV UJI COBA

4.1 Spesifikasi Komputer Uji Coba

Uji coba dilakukan pada komputer dengan spesifikasi sebagai berikut.

Tabel 4.1.1 Spesifikasi Komputer Uji Coba

Nama	ASUS TUF Gaming A15 FA506IC
Prosesor	AMD Ryzen 7 4800H
RAM	16GB DDR4 3200MHz dual channel
OS	Ubuntu 22.04 WSL2 (Windows 11)

4.2 Hasil Uji Coba

4.2.1 Uji Coba dengan 16 Titik pada Ruang Tiga Dimensi

```
Divide And Conquer
Closest Pairs:
----- 1 -----
(-505.77261, 1991.8725, 3763.0764)
(-398.74687, 2105.2746, 1961.6229)

Closest Distance: 1808.1895
8.9507e-05 seconds elapsed
52 euclidean distance comparisons

Brute Force
Closest Pairs:
----- 1 -----
(-505.77261, 1991.8725, 3763.0764)
(-398.74687, 2105.2746, 1961.6229)

Closest Distance: 1808.1895
1.4096e-05 seconds elapsed
120 euclidean distance comparisons
```

4.2.2 Uji Coba dengan 64 Titik pada Ruang Tiga Dimensi

```
Divide And Conquer
Closest Pairs:
```

```

----- 1 -----
(4399.031, 3323.8795, -363.0479)
(3990.0235, 3629.2336, -951.4916)

Closest Distance: 778.97
0.000444392 seconds elapsed
323 euclidean distance comparisons

Brute Force
Closest Pairs:
----- 1 -----
(4399.031, 3323.8795, -363.0479)
(3990.0235, 3629.2336, -951.4916)

Closest Distance: 778.97
0.000159739 seconds elapsed
2016 euclidean distance comparisons

```

4.2.3 Uji Coba dengan 128 Titik pada Ruang Tiga Dimensi

```

Divide And Conquer
Closest Pairs:
----- 1 -----
(4414.3651, -1214.698, 2973.2615)
(4474.8494, -1176.853, 2866.4456)

Closest Distance: 128.45324
0.000861272 seconds elapsed
579 euclidean distance comparisons

Brute Force
Closest Pairs:
----- 1 -----
(4414.3651, -1214.698, 2973.2615)
(4474.8494, -1176.853, 2866.4456)

Closest Distance: 128.45324
0.001681037 seconds elapsed
8128 euclidean distance comparisons

```

4.2.4 Uji Coba dengan 1000 Titik pada Ruang Tiga Dimensi

```

Divide And Conquer
Closest Pairs:
----- 1 -----

```

```

(2379.1123, 4019.7166, 1091.4971)
(2318.4006, 4021.9928, 1150.6499)

Closest Distance: 84.794711
0.012350469 seconds elapsed
9234 euclidean comparisons

Brute Force
Closest Pairs:
----- 1 -----
(2379.1123, 4019.7166, 1091.4971)
(2318.4006, 4021.9928, 1150.6499)

Closest Distance: 84.794711
0.10315231 seconds elapsed
499500 euclidean distance comparisons

```

4.2.5 Uji Coba Lainnya

Berikut adalah hasil uji coba dengan dimensi dan jumlah titik yang lebih variatif dan terkait program yang telah dibuat.

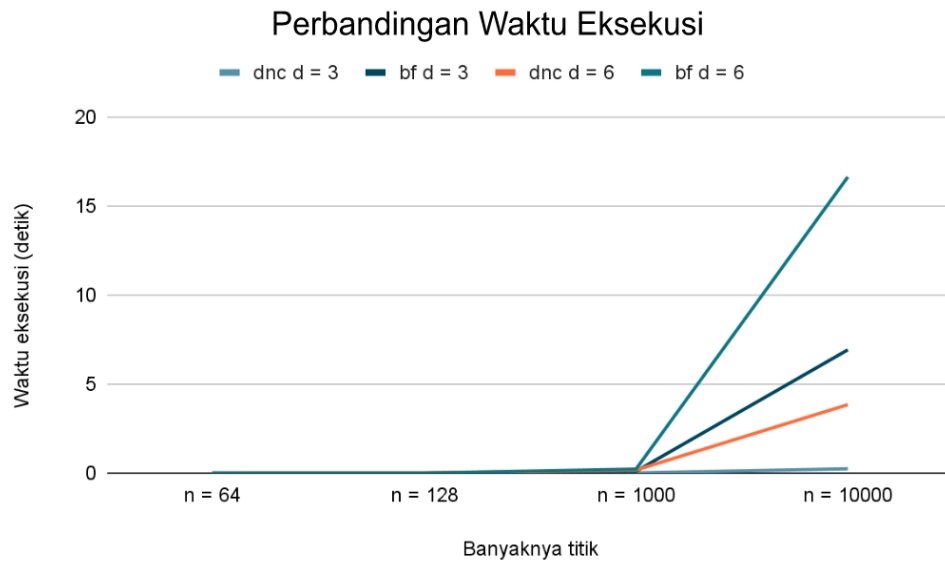
Tabel 4.2.5.1 Hasil Uji Coba Lainnya

Banyak Titik	Dimensi	<i>Divide and Conquer</i>		<i>Brute Force</i>	
		Waktu (s)	Banyak Perbandingan Jarak Euclidean	Waktu (s)	Banyak Perbandingan Jarak Euclidean
16	2	5.2799e-05	29	1.8014e-05	120
	3	8.9507e-05	52	1.4096e-05	120
	4	0.00014052	58	2.2953e-05	120
	6	0.00021495	131	2.0909e-05	120
64	2	0.00019059	183	0.00011759	2016
	3	0.00044439	323	0.00015973	2016
	4	0.00097996	775	0.00021200	2016
	6	0.00336093	2449	0.00028793	2016
128	2	0.00047366	528	0.00061991	8128

	3	0.00086127	579	0.00168103	8128
	4	0.00194080	1116	0.00087444	8128
	6	0.00796576	4905	0.00115366	8128
1000	2	0.00543404	11017	0.05444396	499500
	3	0.01235046	9234	0.10315231	499500
	4	0.02866976	16863	0.06968595	499500
	6	0.17379463	111714	0.22556121	499500
10000	2	0.08605574	271120	5.2103163	49995000
	3	0.24326819	182909	6.9192667	49995000
	4	0.61626154	322483	11.00568	49995000
	6	3.8410908	2068809	16.618325	49995000

4.3 Analisis Hasil Uji Coba

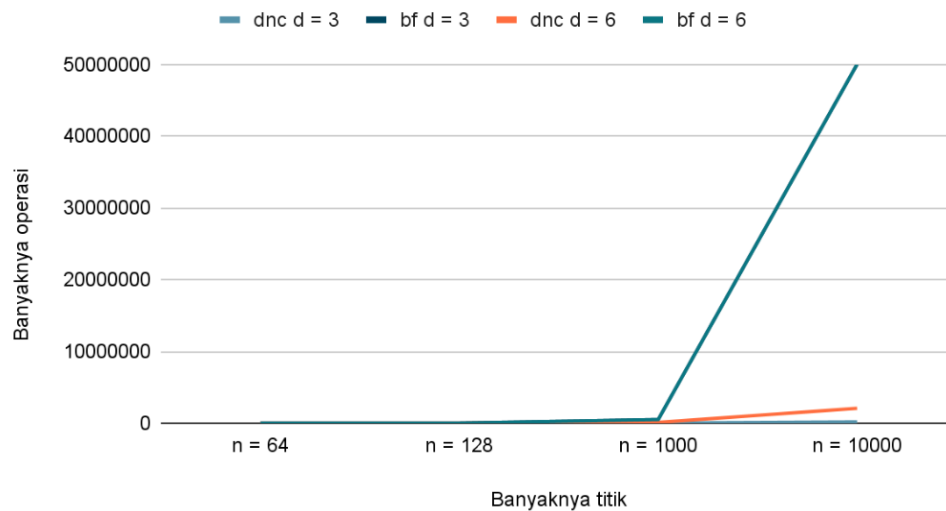
Berdasarkan hasil uji coba yang telah dibahas pada bagian sebelumnya, dapat dilihat bahwa kedua algoritma—*brute force* dan *divide and conquer*—menghasilkan solusi yang sama. Namun, terdapat perbedaan yang signifikan pada waktu eksekusi dan banyaknya perbandingan jarak *euclidean* di antara keduanya. Perbedaan tersebut dapat terlihat pada grafik di bawah ini.



* bf melambangkan *brute force*, dnc melambangkan *divide and conquer*, d melambangkan dimensi
 Gambar 4.3.1 Grafik Perbandingan Waktu Eksekusi antara Algoritma *Brute Force* dan *Divide and Conquer*

Berdasarkan grafik di atas, dapat dilihat bahwa metode *divide and conquer* memiliki waktu eksekusi program jauh lebih cepat dibandingkan waktu eksekusi algoritma *brute force*. Untuk keduanya, semakin besar jumlah titik dan dimensi, semakin lama pula waktu eksekusinya. Perbedaannya adalah waktu eksekusi pada metode *brute force* membesar secara sangat signifikan. Selain itu, dapat dilihat bahwa untuk metode *divide and conquer*, dimensi sangat berpengaruh terhadap waktu eksekusi. Hal ini terjadi karena kompleksitas waktu dari algoritma *divide and conquer* yang diimplementasikan adalah $O(n(\log n)^{d-1})$ sehingga lama eksekusi juga akan meningkat seiring dengan bertambahnya dimensi.

Perbandingan Banyaknya Operasi Perhitungan Jarak *Euclidean*



* bf melambangkan *brute force*, dnc melambangkan *divide and conquer*, d melambangkan dimensi

Gambar 4.3.2 Grafik Perbandingan Banyaknya Perhitungan Jarak *Euclidean* antara Algoritma *Brute Force* dan *Divide and Conquer*

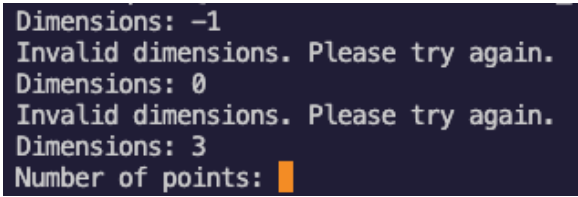
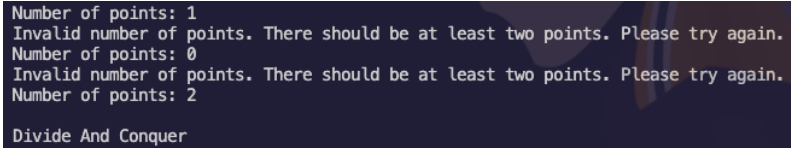
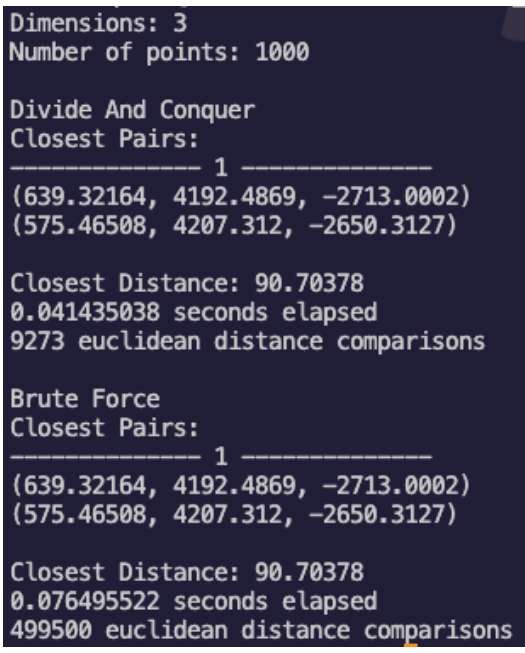
Hal yang serupa juga terjadi pada grafik banyaknya operasi perhitungan jarak *euclidean*. Secara umum, algoritma *divide and conquer* akan memiliki banyak operasi yang jauh lebih sedikit dibandingkan dengan algoritma *brute force*. Banyaknya operasi juga meningkat seiring dengan peningkatan dimensi. Akan tetapi, algoritma *brute force* memiliki banyak operasi yang sama terlepas dari banyaknya dimensi masukan.

LAMPIRAN

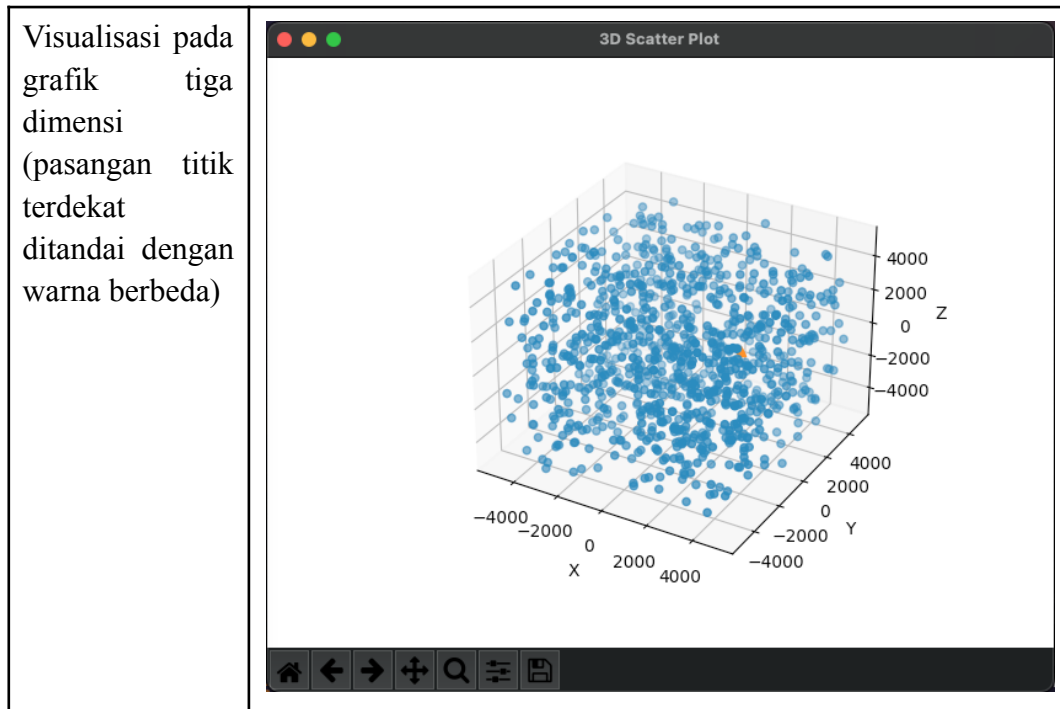
Lampiran 1 Link Repository GitHub

https://github.com/akbarmridho/Tucil2_13521093_13521129

Lampiran 2 Tampilan Program

Nama	Tampilan
Validasi masukan dimensi	 <pre> Dimensions: -1 Invalid dimensions. Please try again. Dimensions: 0 Invalid dimensions. Please try again. Dimensions: 3 Number of points: 0 </pre>
Validasi masukan jumlah titik	 <pre> Number of points: 1 Invalid number of points. There should be at least two points. Please try again. Number of points: 0 Invalid number of points. There should be at least two points. Please try again. Number of points: 2 Divide And Conquer </pre>
Contoh percobaan pada ruang tiga dimensi	 <pre> Dimensions: 3 Number of points: 1000 Divide And Conquer Closest Pairs: ----- 1 ----- (639.32164, 4192.4869, -2713.0002) (575.46508, 4207.312, -2650.3127) Closest Distance: 90.70378 0.041435038 seconds elapsed 9273 euclidean distance comparisons Brute Force Closest Pairs: ----- 1 ----- (639.32164, 4192.4869, -2713.0002) (575.46508, 4207.312, -2650.3127) Closest Distance: 90.70378 0.076495522 seconds elapsed 499500 euclidean distance comparisons </pre>

<p>Contoh percobaan pada ruang multidimensi</p>	<pre> Dimensions: 5 Number of points: 1000 Divide And Conquer Closest Pairs: ----- 1 ----- (1122.2762, -82.184105, 3310.7698, 1543.9385, -2450.95) (1203.381, -369.18365, 3456.5491, 1666.4019, -2415.7232) Closest Distance: 355.57915 0.18569501 seconds elapsed 35973 euclidean distance comparisons Brute Force Closest Pairs: ----- 1 ----- (1203.381, -369.18365, 3456.5491, 1666.4019, -2415.7232) (1122.2762, -82.184105, 3310.7698, 1543.9385, -2450.95) Closest Distance: 355.57915 0.10119466 seconds elapsed 499500 euclidean distance comparisons </pre>
<p><i>File</i> <i>log</i> points.txt (untuk kebutuhan visualisasi 3D)</p>	<pre> log > ≡ points.txt 1 -1835.26 -566.307 452.346 2 -1942.37 1358.02 -2808.63 3 -4604.88 -676.097 4499.01 4 4429.98 4628.06 -259.014 5 2010.62 -3764.47 -327.821 6 -1415.2 1873.05 -299.107 7 3801.15 3601.21 3603.23 </pre>
<p><i>File</i> <i>log</i> pairs_list.txt (untuk kebutuhan visualisasi 3D)</p>	<pre> log > ≡ pairs_list.txt 1 639.322 4192.49 -2713 2 575.465 4207.31 -2650.31 3 </pre>



Lampiran 3 Tabel *Check List* Poin

No.	Poin	Ya	Tidak
1.	Program dapat dikompilasi tanpa kesalahan	√	
2.	Program berhasil <i>running</i>	√	
3.	Program dapat menerima masukan dan menuliskan luaran	√	
4.	Luaran program sudah benar (solusi <i>closest pair</i> benar)	√	
5.	Bonus 1 dikerjakan	√	
6.	Bonus 2 dikerjakan	√	

DAFTAR REFERENSI

- Bentley, John dan Michael Shamos. (1976). *Divide and Conquer in Multidimensional Space*.
- Munir, R. (2022). *Strategi Algoritma: Algoritma Divide and Conquer (Bagian 1)*. Dilansir dari Homepage Rinaldi Munir: [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-\(2021\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian1.pdf). Diakses pada 23 Februari 2023.
- Munir, R. (2022). *Strategi Algoritma: Algoritma Divide and Conquer (Bagian 2)*. Dilansir dari Homepage Rinaldi Munir: [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-\(2021\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian2.pdf). Diakses pada 23 Februari 2023.
- Suri, Subhash. (2021). *Closest Pair*. Dilansir dari <https://sites.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf>. Diakses pada 28 Februari 2023.
- Wengrow, Jay. (2020). *A Common-Sense Guide to Data Structures and Algorithms*. Raleigh: The Pragmatic Bookshelf.