LAPORAN TUGAS KECIL 02

IF2211 STRATEGI ALGORITMA

MEMBANGUN KURVA BÉZIER DENGAN ALGORITMA TITIK TENGAH BERBASIS DIVIDE AND CONQUER

Disusun oleh:

Juan Alfred Widjaya 13522073

Ivan Hendrawan Tan 13522111

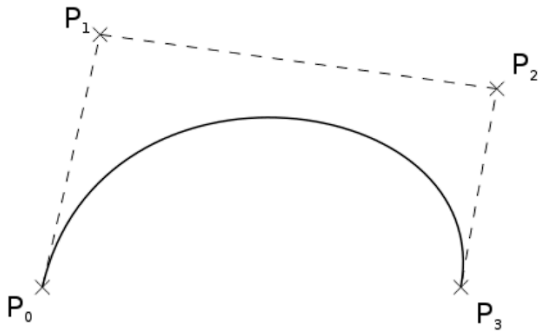
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

BAB I
DESKRIPSI MASALAH



Gambar 1 Kurva Bézier Kubik

Kurva Bézier adalah kurva halus yang sering digunakan dalam desain grafis, animasi, dan manufaktur. Kurva ini dibuat dengan menghubungkan beberapa titik kontrol, yang menentukan bentuk dan arah kurva. Cara membuatnya cukup mudah, yaitu dengan menentukan titik-titik kontrol dan menghubungkannya dengan kurva. Kurva Bézier memiliki banyak kegunaan dalam kehidupan nyata, seperti pen tool, animasi yang halus dan realistis, membuat desain produk yang kompleks dan presisi, dan membuat font yang indah dan unik. Keuntungan menggunakan kurva Bézier adalah kurva ini mudah diubah dan dimanipulasi, sehingga dapat menghasilkan desain yang presisi dan sesuai dengan kebutuhan.

Sebuah kurva Bézier didefinisikan oleh satu set titik kontrol P_0 sampai P_n , dengan n disebut order (n = 1 untuk linier, n = 2 untuk kuadrat, dan seterusnya). Titik kontrol pertama dan terakhir selalu menjadi ujung dari kurva, tetapi titik kontrol antara (jika ada) umumnya tidak terletak pada kurva. Pada gambar 1 diatas, titik kontrol pertama adalah P_0, sedangkan titik kontrol terakhir adalah P_3. Titik kontrol P_1 dan P_2 disebut sebagai titik kontrol antara yang tidak terletak dalam kurva yang terbentuk.

Mengulas lebih jauh mengenai bagaimana sebuah kurva Bézier bisa terbentuk, misalkan diberikan dua buah titik P_0 dan P_1 yang menjadi titik kontrol, maka kurva Bézier yang terbentuk adalah sebuah garis lurus antara dua titik. Kurva ini disebut dengan kurva Bézier linier. Misalkan terdapat sebuah titik Q_0 yang berada pada garis yang dibentuk oleh P0 dan P_1, maka posisinya dapat dinyatakan dengan persamaan parametrik berikut.

$$Q_0 = B(t) = (1-t)P_0 + tP_1, \qquad t \in [0,1]$$

dengan t dalam fungsi kurva Bézier linier menggambarkan seberapa jauh B(t) dari P_0 ke P_1. Misalnya ketika t = 0.25, maka B(t) adalah seperempat jalan dari titik P0 ke P1. sehingga seluruh rentang variasi nilai t dari 0 hingga 1 akan membuat persamaan B(t) membentuk sebuah garis lurus dari P_0 ke P_1.

Misalkan selain dua titik sebelumnya ditambahkan sebuah titik baru, sebut saja P_2, dengan P_0 dan P_2 sebagai titik kontrol awal dan akhir, dan P_1 menjadi titik kontrol antara. Dengan menyatakan titik Q_1 terletak diantara garis yang menghubungkan P_1 dan P_2, dan membentuk kurva Bézier linier yang berbeda dengan kurva letak Q_0 berada, maka dapat dinyatakan sebuah titik baru, R_0 yang berada diantara garis yang menghubungkan Q_0 dan Q_1 yang bergerak membentuk kurva Bézier kuadratik terhadap titik P_0 dan P_2. Berikut adalah uraian persamaannya.

$$Q_0 = B(t) = (1-t)P_0 + tP_1, \qquad t \in [0,1]$$

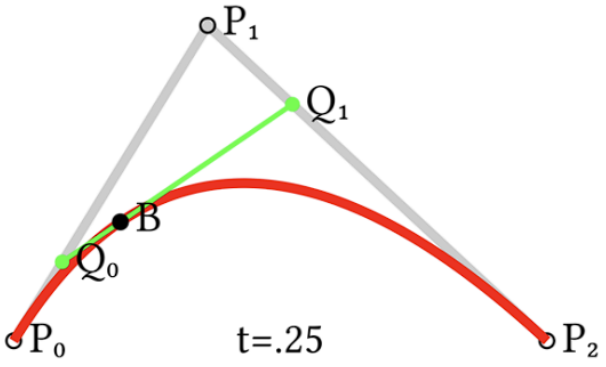
$$Q_1 = B(t) = (1-t)P_1 + tP_2, \qquad t \in [0,1]$$

$$R_0 = B(t) = (1-t)Q_0 + tQ_1, \qquad t \in [0,1]$$

dengan melakukan substitusi nilai Q_0 dan Q_1, maka diperoleh persamaan sebagai berikut.

$$R_0 = B(t) = (1-t)^2P_0 + (1-t)tP_1 + t^2P_2, \qquad t \in [0,1]$$

Berikut adalah ilustrasi dari kasus diatas.



Gambar 2 Pembentukan Kurva Bézier Kuadratik.

Proses ini dapat juga diaplikasikan untuk jumlah titik yang lebih dari tiga, misalnya empat titik akan menghasilkan kurva Bézier kubik, lima titik akan menghasilkan kurva Bézier kuartik, dan seterusnya. Berikut adalah persamaan kurva Bézier kubik dan kuartik dengan menggunakan prosedur yang sama dengan yang sebelumnya.

$$S_0 = B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3, \qquad t \in [0,1]$$

$$T_0 = B(t) = (1-t)^4 P_0 + 4(1-t)^3 t P_1 + 6(1-t)^2 t^2 P_2 + 4(1-t)t^3 P_3 + t^4 P_4, \qquad t \in [0,1]$$

Tentu saja persamaan yang terbentuk sangat panjang dan akan semakin rumit seiring bertambahnya titik. Oleh sebab itu, dalam rangka melakukan efisiensi pembuatan kurva Bézier yang sangat berguna ini, maka algoritma titik tengah berbasis Divide and Conquer dapat menjadi alternatif.

BAB II
LANDASAN TEORI

2.1 Algoritma Brute Force

Algoritma Brute Force adalah sebuah algoritma yang memecahkan persoalan dengan cara sederhana dan langsung. Algoritma Brute Force kadang disebut sebagai algoritma naif.

Pada umumnya, algoritma Brute Force tidak efisien dalam memecahkan persoalan karena algoritma ini memerlukan volume komputasi yang besar dan waktu yang lama untuk menyelesaikan sebuah permasalahan yang kompleks. Namun, algoritma Brute Force biasanya sederhana pembuatannya dan mudah dimengerti. Hampir semua permasalahan dapat diselesaikan dengan algoritma Brute Force termasuk pembuatan kurva bézier ini.

2.2 Algoritma Divide and Conquer

Algoritma Divide and Conquer adalah algoritma yang memecahkan persoalan menjadi beberapa upa-persoalan yang berukuran lebih kecil dan memiliki kemiripan terhadap persoalan awal.

Masing-masing upa-persoalan diselesaikan sehingga dari setiap upa-persoalan didapat sebuah solusi baik secara langsung jika berukuran kecil atau secara rekursif jika berukuran besar. Setiap solusi dari upa-persoalan tadi akan digabungkan untuk menghasilkan sebuah solusi untuk persoalan di awal.

2.3 Kurva Bézier

Kurva Bézier adalah sebuah kurva parametrik yang sering kali digunakan di dalam grafika komputer. Kurva ini dinamai dari seorang insinyur asal Prancis yang bernama Pierre Bézier, kurva ini pertama kali ia gunakan untuk mendesain bodi mobil Renault.

Kurva Bézier didapatkan dari menghubungkan beberapa titik kontrol yang akan menentukan bentuk dan arah dari kurva. Keuntungan dari menggunakan kurva Bézier adalah mudah untuk diubah dan dimanipulasi sehingga dapat menghasilkan kurva yang sesuai dengan kebutuhan.

BAB III
IMPLEMENTASI PROGRAM

3.1. File *util.py*

FIle *util.py* berisi fungsi valid_int_input untuk mengecek apakah input yang dimasukkan user bertipe integer atau bukan dan lebih besar dari *lower_limit*

```
def valid_int_input(msg: str, has_limit: bool = False, lower_limit: int = 0) → int:
    while True:
        try:
            user_input = int(input(msg))
            if has_limit and user_input < lower_limit:
                raise ValueError
            return user_input
        except ValueError:
            if has_limit:
                print(f"\nInvalid input, input must be an integer greater or equal to
{lower_limit}!\n")
            else:
                print("\nInvalid input, input must be an integer!\n")
```

3.2. File *bezier_curve.py*

File *bezier_curve.py* berisi fungsi - fungsi pembentukan kurva bezier.

```
from typing import Tuple, List

def midpoint(p1: Tuple[int, int], p2: Tuple[int, int]) → Tuple[int, int]:
    return ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2)

def bezier_divide_and_conquer(
    control_points: List[Tuple[int, int]],
    iterations: int
) → List[Tuple[int, int]]:
    if iterations == 0:
        return [control_points[0], control_points[-1]]
    else:
        first_half = [control_points[0]]
        second_half = [control_points[-1]]
```

```
        for _ in range(len(control_points), 1, -1):
            midpoints = [midpoint(control_points[i], control_points[i + 1]) for i in
range(len(control_points) - 1)]
            control_points = midpoints
            first_half.append(midpoints[0])
            second_half.insert(0, midpoints[-1])

        first_half_bezier = bezier_divide_and_conquer(first_half, iterations - 1)
        second_half_bezier = bezier_divide_and_conquer(second_half, iterations - 1)
        return first_half_bezier + second_half_bezier[1:]

def bezier_divide_and_conquer_animate(
    control_points: List[Tuple[int, int]],
    iterations: int,
    all_midpoints: List[List[List[Tuple[int, int]]]] | None = None
) → Tuple[List[Tuple[int, int]], List[List[List[Tuple[int, int]]]]]:
    if all_midpoints is None:
        all_midpoints = []

    if iterations == 0:
        return [control_points[0], control_points[-1]], all_midpoints
    else:
        all_mid = [control_points]
        first_half = [control_points[0]]
        second_half = [control_points[-1]]
        for _ in range(len(control_points), 1, -1):
            midpoints = [midpoint(control_points[i], control_points[i + 1]) for i in
range(len(control_points) - 1)]
            all_mid.append(midpoints)
            control_points = midpoints
            first_half.append(midpoints[0])
            second_half.insert(0, midpoints[-1])

        all_midpoints.append(all_mid)

        first_half_bezier, all_midpoints = bezier_divide_and_conquer_animate(first_half,
iterations - 1, all_midpoints)
        second_half_bezier, all_midpoints = bezier_divide_and_conquer_animate(second_half,
iterations - 1, all_midpoints)

        return first_half_bezier + second_half_bezier[1:], all_midpoints

def de_casteljau(points: List[Tuple[int, int]], t: float) → Tuple[int, int]:
```

```
    if len(points) == 1:
        return points[0]
    else:
        new_points = []
        for i in range(len(points) - 1):
            new_point = ((1 - t) * points[i][0] + t * points[i + 1][0], (1 - t) *
points[i][1] + t * points[i + 1][1])
            new_points.append(new_point)
        return de_casteljau(new_points, t)

def bezier_bruteforce(control_points: List[Tuple[int, int]], iterations: int) →
List[Tuple[int, int]]:
    curve_points = []
    num_points = 2 ** iterations + 1
    for i in range(num_points):
        t = i / (num_points - 1)
        curve_points.append(de_casteljau(control_points, t))
    return curve_points
```

3.3. File *main.py*

File *main.py* merupakan file utama program yang dapat menerima masukan pengguna dan memprosesnya dengan algoritma divide and conquer serta menampilkan hasil kurva bezier dengan animasi.

```
from matplotlib.animation import FuncAnimation
import matplotlib.pyplot as plt
from colorama import init, Fore
import time

from bezier_curve import bezier_divide_and_conquer_animate
from util import valid_int_input

# User input with color
init(autoreset=True)
print(Fore.LIGHTYELLOW_EX + "\nBézier Curve Generator\n")
n = valid_int_input(Fore.WHITE + "Number of control points (≥1): ", has_limit=True,
lower_limit=1)
control_points = []
for i in range(n):
    print(Fore.LIGHTMAGENTA_EX + f"\n== Control Point {i + 1} ==")
    x = valid_int_input(Fore.WHITE + f"x{i + 1}: ")
```

```
    y = valid_int_input(Fore.WHITE + f"y{i + 1}: ")
    control_points.append((x, y))
iterations = valid_int_input(Fore.WHITE + "\nNumber of iterations (≥1): ", has_limit=True,
lower_limit=1)

# # Without user input
# control_points = [(0, 0), (100, 200), (300, 300), (500, 0), (300, -300), (-100, -200),
(-200, 100)]
# iterations = 4

start = time.time()
curve_points, midpoints = bezier_divide_and_conquer_animate(control_points=control_points,
iterations=iterations)
end = (time.time() - start) * 1000  # milliseconds

def update(frame):
    ax.clear()

    iteration = 0
    total_levels = 0
    while total_levels + len(midpoints[iteration]) ≤ frame:
        total_levels += len(midpoints[iteration])
        iteration += 1
    level = frame - total_levels

    # Plot control points
    control_x, control_y = zip(*control_points)
    ax.scatter(control_x, control_y, color='red', label='Control Points')
    ax.plot(control_x, control_y, color='red', linestyle='dashed', alpha=0.5)

    # Plot midpoints
    for iter_index in range(iteration + 1):
        for level_index in range(len(midpoints[iter_index])):
            if iter_index == iteration and level_index > level:
                break
            mid_x, mid_y = zip(*midpoints[iter_index][level_index])
            ax.plot(mid_x, mid_y, color='green', linestyle='dotted', alpha=0.5)

    # Plot final curve
    if frame == total_frames - 1:
        x, y = zip(*curve_points)
        ax.plot(x, y, color='black', label='Bézier Curve')
        ax.legend()
```

```
    ax.set_title(f"Bézier Curve (Divide and Conquer)\nProcess Time: {end:.2f} ms\nAnimating
> 4 iterations may take times.")

fig, ax = plt.subplots()
total_frames = sum(len(levels) for levels in midpoints)
ani = FuncAnimation(fig, update, frames=range(total_frames), interval=10, repeat=False)
plt.show()
```

3.4. File *test.py*

File *test.py* berisi unit test yang diuji.

```
import time
import matplotlib.pyplot as plt
from bezier_curve import bezier_divide_and_conquer, bezier_bruteforce

# Test cases
tests = [
    ([(0, 0), (1, 1), (2, 1)], 20),
    ([(0, 0), (100, 200), (300, 300), (500, 0)], 9),
    ([(0, 0), (100, 200), (200, 0), (300, 200), (400, 0)], 9),
    ([(0, 0), (100, 300), (200, -100), (300, 400), (400, 0), (500, 300)], 13),
    ([(0, 0), (200, 200), (400, 100), (600, 300), (800, 0), (1000, 400)], 18),
    ([(0, 0), (400, 600), (1000, 400), (1000, -400), (400, -800), (-400, -400)], 20),
    ([(0, 0), (-300, 0), (0, -300), (300,0), (200, 200), (0, 200), (-100, 100)], 22),
    ([(0, 0), (200, 200), (100, -100), (-100, 200), (40, 400), (500, 100), (400, -200)],
24)
]

# Show graph toggle
show_graph = True

# Test
for test_num, (control_points, iterations) in enumerate(tests, start=1):
    print(f"\nTest {test_num}:")
    print(f"Number of Control Points: {len(control_points)}")
    print(f"Control Points: {control_points}")
    print(f"Midpoint DnC Iterations: {iterations}")



    # Divide and Conquer
    start = time.time()
```

```
    curve_dnc = bezier_divide_and_conquer(control_points=control_points,
iterations=iterations)
    end_dnc = time.time() - start
    print(f"Divide and Conquer time: {end_dnc * 1000:.2f}ms")

    # Bruteforce
    start = time.time()
    curve_bf = bezier_bruteforce(control_points=control_points, iterations=iterations)
    end_bf = time.time() - start
    print(f"Bruteforce time: {end_bf * 1000:.2f}ms")

    if show_graph:
        # Plot curve from Divide and Conquer
        x_dnc, y_dnc = zip(*curve_dnc)
        plt.plot(x_dnc, y_dnc, label=f"Bézier Curve (DnC) {end_dnc * 1000:.2f}ms",
color='black')

        # Plot curve from Bruteforce
        x_bf, y_bf = zip(*curve_bf)
        plt.plot(x_bf, y_bf, label=f"Bézier Curve (Bruteforce) {end_bf * 1000:.2f}ms",
color='yellow', linestyle='dashed')

        control_x, control_y = zip(*control_points)
        plt.scatter(control_x, control_y, color='red', label='Control Points')
        plt.plot(control_x, control_y, color='red', linestyle='dotted', alpha=0.5)

        plt.legend()
        plt.title(f"Test {test_num}")
        plt.show()
```

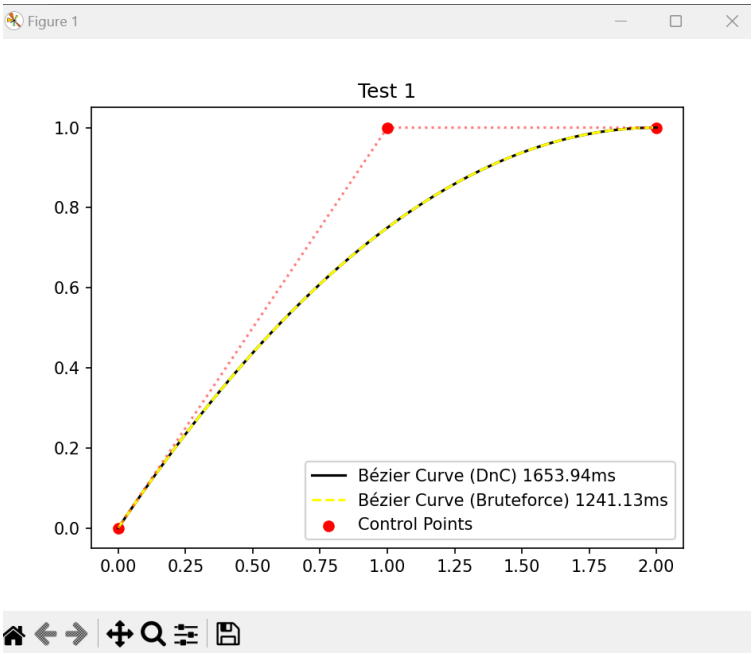
BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1. Pengujian

a. Titik kontrol: [(0, 0), (1, 1), (2, 1)], jumlah iterasi: 20

```
tests = [
    ([(0, 0), (1, 1), (2, 1)], 20)
]
```

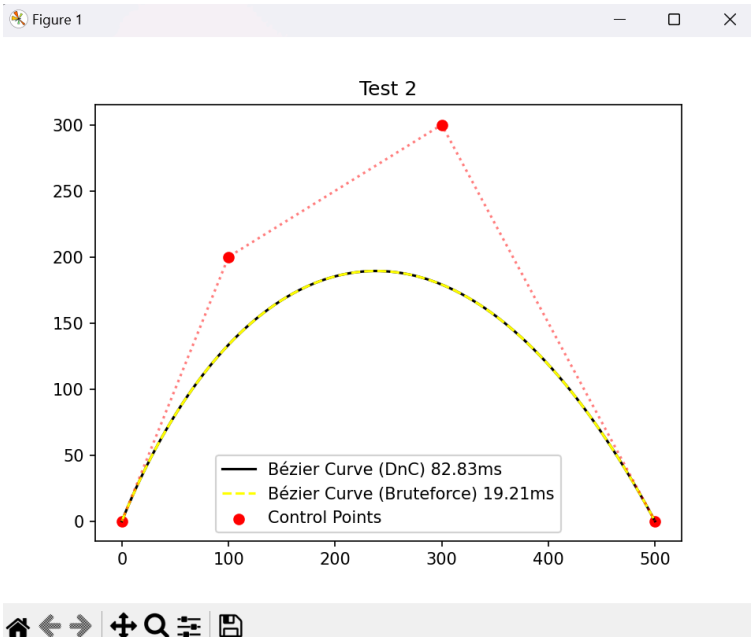
```
Test 1:
Number of Control Points: 3
Control Points: [(0, 0), (1, 1), (2, 1)]
Midpoint DnC Iterations: 20
Divide and Conquer time: 1653.94ms
Bruteforce time: 1241.13ms
```



b. Titik kontrol: [(0, 0), (100, 200), (300, 300), (500, 0)], jumlah iterasi: 9

```
tests = [
    ([(0, 0), (100, 200), (300, 300), (500, 0)], 9)
]
```

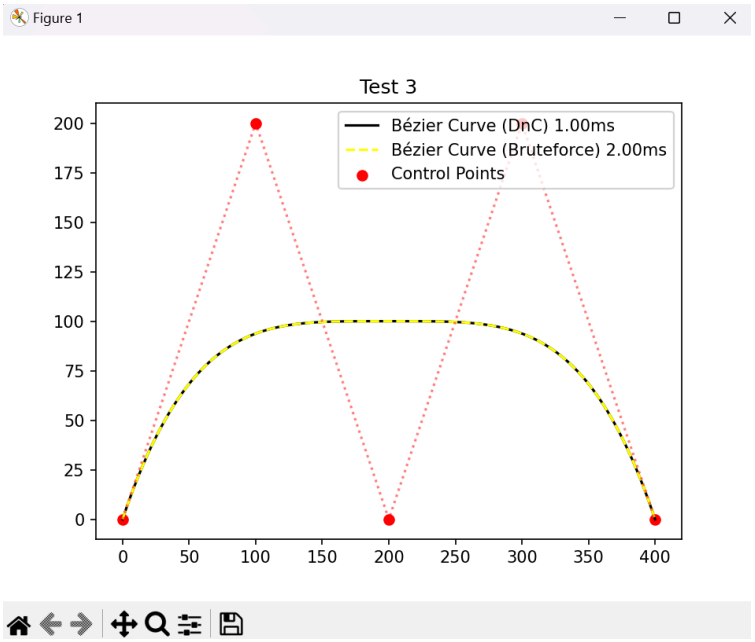
```
Test 2:
Number of Control Points: 4
Control Points: [(0, 0), (100, 200), (300, 300), (500, 0)]
Midpoint DnC Iterations: 9
Divide and Conquer time: 82.83ms
Bruteforce time: 19.21ms
```



c. Titik kontrol: [(0, 0), (100, 200), (200, 0), (300, 200), (400, 0)], jumlah iterasi: 9

```
tests = [
    ([(0, 0), (100, 200), (200, 0), (300, 200), (400, 0)], 9)
]
```

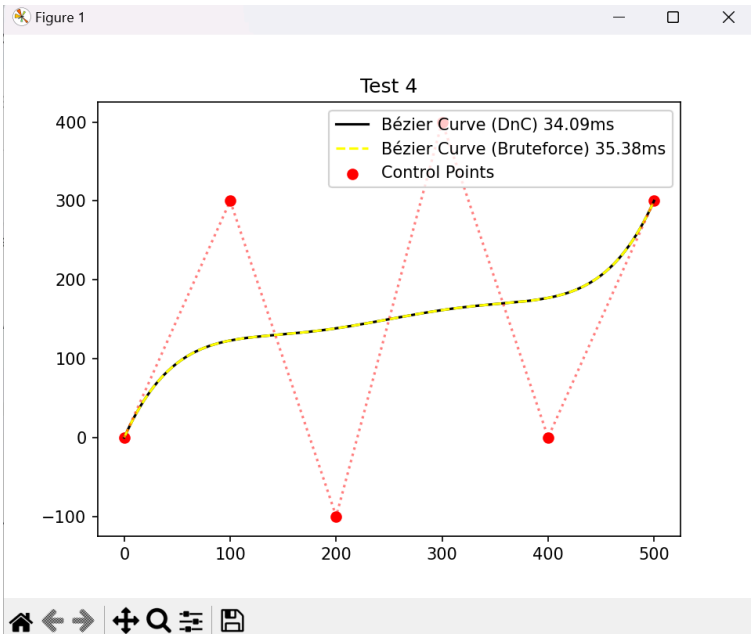
```
Test 3:
Number of Control Points: 5
Control Points: [(0, 0), (100, 200), (200, 0), (300, 200), (400, 0)]
Midpoint DnC Iterations: 9
Divide and Conquer time: 1.00ms
Bruteforce time: 2.00ms
```



d. Titik kontrol: [(0, 0), (100, 300), (200, -100), (300, 400), (400, 0), (500, 300)], jumlah iterasi: 13

```
tests = [
    ([(0, 0), (100, 300), (200, -100), (300, 400), (400, 0), (500, 300)], 13)
]
```

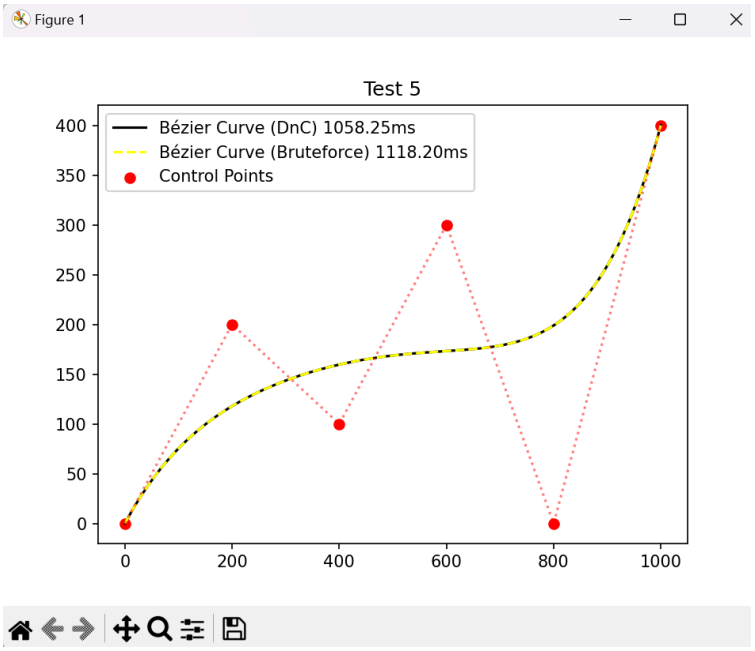
```
Test 4:
Number of Control Points: 6
Control Points: [(0, 0), (100, 300), (200, -100), (300, 400), (400, 0), (500, 300)]
Midpoint DnC Iterations: 13
Divide and Conquer time: 34.09ms
Bruteforce time: 35.38ms
```



e. Titik kontrol: [(0, 0), (200, 200), (400, 100), (600, 300), (800, 0), (1000, 400)], jumlah iterasi: 18

```
tests = [
    ([(0, 0), (200, 200), (400, 100), (600, 300), (800, 0), (1000, 400)], 18)
]
```

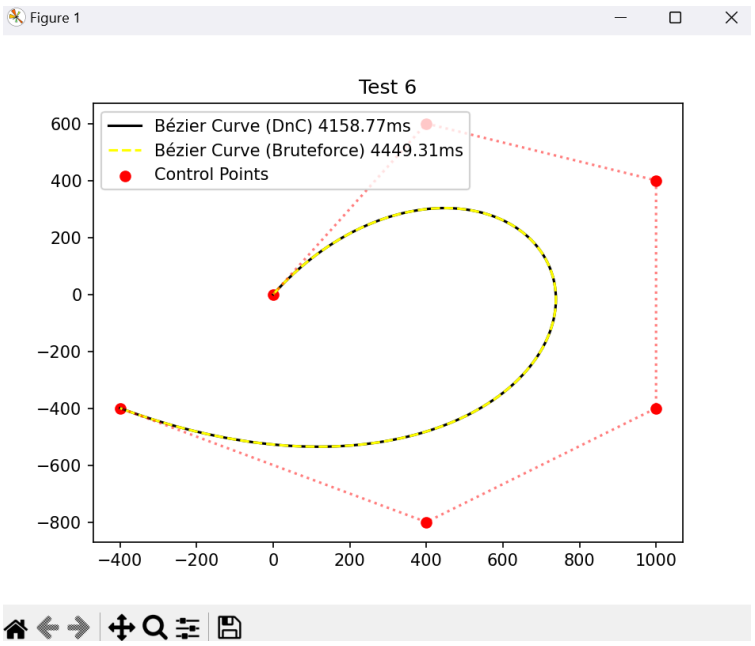
```
Test 5:
Number of Control Points: 6
Control Points: [(0, 0), (200, 200), (400, 100), (600, 300), (800, 0), (1000, 400)]
Midpoint DnC Iterations: 18
Divide and Conquer time: 1058.25ms
Bruteforce time: 1118.20ms
```



f. Titik kontrol: [(0, 0), (400, 600), (1000, 400), (1000, -400), (400, -800), (-400, -400)], jumlah iterasi: 20

```
tests = [
    ([(0, 0), (400, 600), (1000, 400), (1000, -400), (400, -800), (-400, -400)], 20)
]
```

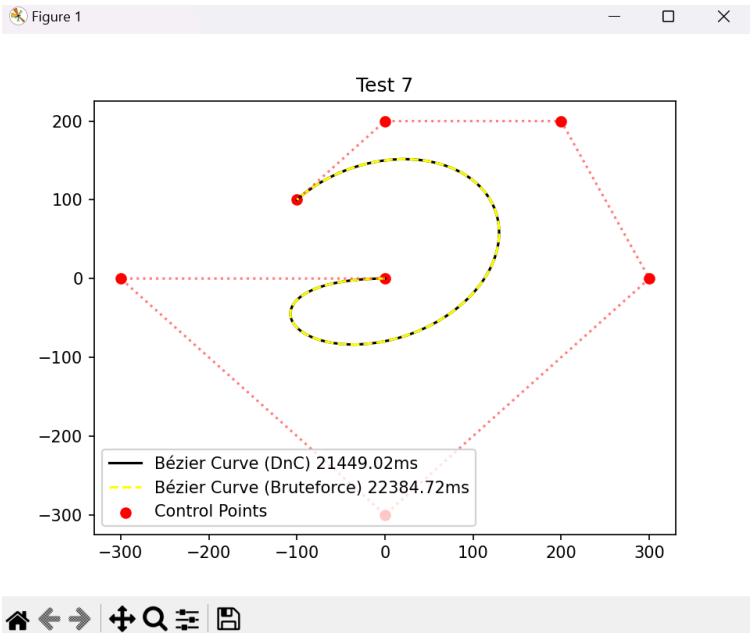
```
Test 6:
Number of Control Points: 6
Control Points: [(0, 0), (400, 600), (1000, 400), (1000, -400), (400, -800), (-400, -400)]
Midpoint DnC Iterations: 20
Divide and Conquer time: 4158.77ms
Bruteforce time: 4449.31ms
```



g. Titik kontrol: [(0, 0), (-300, 0), (0, -300), (300,0), (200, 200), (0, 200), (-100, 100)], jumlah iterasi: 22

```
tests = [
    ([(0, 0), (-300, 0), (0, -300), (300,0), (200, 200), (0, 200), (-100, 100)], 22)
]
```

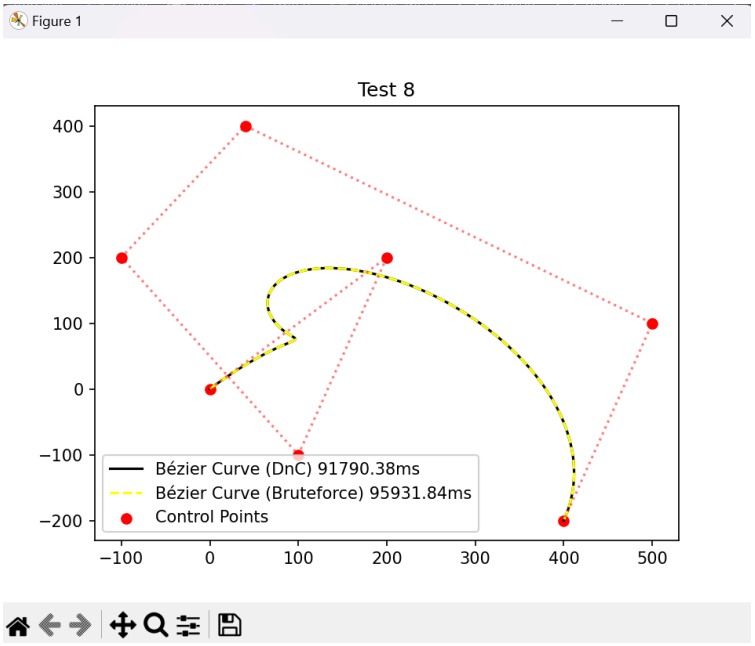
```
Test 7:
Number of Control Points: 7
Control Points: [(0, 0), (-300, 0), (0, -300), (300, 0), (200, 200), (0, 200), (-100, 100)]
Midpoint DnC Iterations: 22
Divide and Conquer time: 21449.02ms
Bruteforce time: 22384.72ms
```



h. Titik kontrol: [(0, 0), (200, 200), (100, -100), (-100, 200), (40, 400), (500, 100), (400, -200)], jumlah iterasi: 24

```
tests = [
    ([(0, 0), (200, 200), (100, -100), (-100, 200), (40, 400), (500, 100), (400, -200)], 24)
]
```

```
Test 8:
Number of Control Points: 7
Control Points: [(0, 0), (200, 200), (100, -100), (-100, 200), (40, 400), (500, 100), (400, -200)]
Midpoint DnC Iterations: 24
Divide and Conquer time: 91790.38ms
Bruteforce time: 95931.84ms
```



4.2. Keterangan

- Pengujian di atas dilakukan dengan file *test.py* karena file *main.py* digunakan untuk menampilkan animasi untuk bézier curve dengan metode Divide and Conquer. Input pada file *test.py* dilakukan dengan pengubahan kode pada “tests” secara langsung, sedangkan input file *main.py* dilakukan secara manual melalui CLI
- Seluruh pengujian dilakukan pada laptop dengan spesifikasi sebagai berikut.

4.3. Analisis

Algoritma Kurva Bézier menggunakan pendekatan Divide and Conquer dapat dijelaskan secara lengkap dengan konsep-konsep berikut:

1. Base Case

   Kasus dasar dari rekursi adalah ketika jumlah iterasi (iterations) mencapai 0. Pada kondisi ini, algoritma mengembalikan dua titik: titik awal dan titik akhir dari daftar control_points. Ini merepresentasikan kurva Bézier yang paling sederhana, yaitu garis lurus yang menghubungkan dua titik tersebut.

2. Divide

   Langkah pembagian melibatkan pemisahan titik-titik kontrol (control_points) menjadi dua bagian, first_half dan second_half. first_half diinisialisasi dengan menambahkan titik awal dari control_points, sedangkan second_half diinisialisasi dengan menambahkan titik akhirnya. Jika jumlah titik kontrol (n) lebih dari satu, algoritma menghitung titik-titik tengah dari setiap pasangan titik kontrol berturut-turut. Untuk n titik kontrol, algoritma menghitung $n - 1$ titik tengah. Titik tengah pertama dari perhitungan ini ditambahkan ke akhir first_half, dan titik tengah terakhir ditambahkan ke awal second_half. Setelah menghitung titik-titik tengah, control_points diperbarui menjadi kumpulan titik-titik tengah yang baru dihitung, mengurangi jumlah titik kontrol menjadi $n - 1$ pada tiap iterasi. Proses ini diulang hingga control_points hanya tersisa satu titik, yang bukan titik kontrol awal melainkan titik hasil dari iterasi terakhir.

3. Conquer

   Setelah titik-titik kontrol dibagi menjadi first_half dan second_half, algoritma kemudian menyelesaikan masalah ini secara rekursif. Fungsi bezier_divide_and_conquer dipanggil secara rekursif pada first_half dan

second_half, dengan mengurangi jumlah iterasi ($iterations - 1$). Rekursi ini terus berlanjut hingga mencapai kasus dasar.

4. Combine

   Setelah rekursi selesai, hasil dari pemanggilan rekursif pada first_half dan second_half dikombinasikan untuk membentuk kurva Bézier akhir. Hasil rekursi pada first_half (first_half_bezier) dan second_half (second_half_bezier) digabungkan dengan cara menambahkan semua titik dari first_half_bezier ke hasil akhir dan kemudian menambahkan semua titik dari second_half_bezier kecuali titik pertamanya (untuk menghindari duplikasi titik).

Dengan menggunakan pendekatan Divide and Conquer, algoritma kurva Bézier secara efisien memecah masalah menjadi sub-masalah yang lebih kecil, menyelesaikannya secara rekursif, dan menggabungkan hasilnya untuk mendapatkan solusi akhir. Ini memungkinkan algoritma untuk bekerja dengan cepat dan efisien, bahkan untuk kurva Bézier dengan banyak titik kontrol. Kompleksitas waktu dari algoritma kurva Bézier dengan pendekatan Divide and Conquer dapat dianalisis sebagai berikut:

- Pada setiap iterasi, algoritma menghitung titik-titik tengah dari pasangan titik kontrol berturut-turut. Jika ada n titik kontrol, maka $n - 1$ titik tengah dihitung. Karena titik-titik tengah ini menjadi titik kontrol baru untuk penghitungan berikutnya, jumlah titik yang perlu diproses berkurang secara bertahap: $n - 1$, $n - 2$, ..., hingga 1. Oleh karena itu, jumlah operasi yang dilakukan untuk satu pemanggilan fungsi adalah $(n - 1) + (n - 2) + \ldots + 1 = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$.
- Algoritma ini menggunakan pendekatan divide and conquer, setiap pemanggilan fungsi akan membagi masalah menjadi dua sub-masalah tetapi dengan jumlah titik kontrol yang sama tetapi lebih berdekatan, masing-masing dengan sekitar n titik kontrol. Kedua sub-masalah ini kemudian diselesaikan secara rekursif. Sehingga, kompleksitas waktunya dapat dihitung dengan relasi rekurens yakni $T(n, i) = \frac{n(n-1)}{2} + 2T(n, i - 1), i > 0$ dan $T(n, i) = 1, i = 0$, dimana i adalah jumlah iterasi yang dilakukan, dan n adalah jumlah titik kontrol. Dari relasi rekurens tersebut, didapat kompleksitas

waktunya yakni $T(n,\ i) = (\frac{n(n-1)}{2})(2^i - 1) + 2^i,\ i > 0$ dan $T(n,\ i) = 1,\ i = 0$. Sehingga berapapun jumlah iterasinya, kompleksitas waktu berdasarkan n hanya merupakan kelipatan dari $\frac{n(n-1)}{2}$ ditambah c yakni $k\frac{n(n-1)}{2} + c$. Dalam notasi Big O, dapat disimpulkan bahwa kompleksitas algoritma ini berdasarkan jumlah titik kontrol (n) adalah $O(n^2)$. Dan jika dilihat dari jumlah iterasi (i), kompleksitas waktunya hanya merupakan kelipatan dari $2^i - 1$ ditambah 2^i yakni $k(2^i - 1) + 2^i$. Dalam notasi Big O, kompleksitas algoritma ini berdasarkan jumlah iterasi yakni $O(2^i)$. Jika digabung kompleksitas waktunya, dalam notasi Big O adalah $O(2^i n^2)$.

- Jika dihitung dengan $n = 3$ (jumlah titik kontrol yakni 3), maka kompleksitas hanya bergantung pada i yakni $T(3,\ i) = (\frac{3(3-1)}{2})(2^i - 1) + 2^i = 3(2^i - 1) + 2^i = 4.2^i - 3$. Maka dalam notasi Big O, kompleksitas waktu algoritma dengan 3 titik kontrol yakni $O(2^i)$, dengan i adalah jumlah iterasi.

Untuk pembangunan kurva Bézier dengan algoritma Brute Force memiliki langkah sebagai berikut.

1. Pertama buat sebuah array kosong yang akan menyimpan titik-titik dari kurva Bézier.
2. Lakukan loop sebanyak jumlah num_points yang bernilai 2 pangkat jumlah iterasi ditambah 1 ($2^i + 1$).
3. Pada setiap iterasi loop, dilakukan penambahan elemen dari fungsi de_casteljau ke dalam array kosong pada langkah 1. Fungsi de_casteljau memiliki parameter berupa array yang berisi titik dan t didapat dari i pada loop dibagi dengan num_points yang dikurangi 1 terlebih dahulu.
4. Di dalam fungsi de_casteljau, terdapat basis untuk menghentikan rekursif yang ada di dalam fungsi ini. Basis tersebut adalah jika jumlah titik telah mencapai 1, maka akan mengembalikan titik tersebut.

5. Jika jumlah titik > 1, buat sebuah array kosong (new_points) untuk menyimpan titik-titik baru nantinya.
6. Lakukan iterasi sebanyak jumlah titik dikurang 1, dengan setiap loop menghasilkan sebuah titik baru dari rumus **((1 - t) * points[i][0] + t * points[i + 1][0], (1 - t) * points[i][1] + t * points[i + 1][1])**. Hasil titik baru ini ditambahkan ke dalam array new_points.
7. Setelah tiap iterasi loop, fungsi akan melakukan rekursif dengan parameter baru berupa new_points dan t.

Kompleksitas algoritma Brute Force ini dapat dianalisis sebagai berikut.

1. Pada fungsi de_casteljau, jika terdapat n jumlah titik, maka ada sejumlah n - 1 perhitungan yang dilakukan. Operasi ini dilakukan secara terus menerus dari $n-1$, $n-2$ hingga jumlah titiknya tersisa satu. Jumlah operasi yang dilakukan yakni sebesar $(n-1)+(n-2)+\ldots+1=\frac{(n-1)(n-1+1)}{2}=\frac{n(n-1)}{2}$. Oleh karena itu, fungsi de_casteljau memiliki kompleksitas waktu sebesar $T(n)=\frac{n(n-1)}{2}$. Dalam notasi Big O, fungsi de_casteljau memiliki kompleksitas $O(n^2)$.

2. Untuk fungsi bezier_bruteforce, terdapat pemanggilan fungsi de_casteljau yang memiliki kompleksitas $O(n^2)$, selain itu juga ada iterasi sebanyak num_points yang bernilai (2^i+1) sehingga fungsi bezier_bruteforce memiliki kompleksitas sebesar $T(n,\ i)=(2^i+1)(\frac{n(n-1)}{2})$ dengan n adalah jumlah titik dan i adalah jumlah iterasi. Jika i dipandang sebagai sebuah konstanta, maka kompleksitas fungsi ini sebesar $O(n^2)$. Sedangkan jika n dianggap sebagai konstanta, maka kompleksitas fungsi ini sebesar $O(2^i)$.

Kedua algoritma memiliki kompleksitas waktu yang berbeda tetapi Big O yang sama yakni $O(2^i n^2)$. Algoritma Divide and Conquer memiliki kompleksitas waktu $T(n,\ i)=(\frac{n(n-1)}{2})(2^i-1)+2^i$, sedangkan algoritma Brute Force

memiliki kompleksitas waktu $T(n, i) = (2^i + 1)(\frac{n(n-1)}{2})$. Perbedaan kedua algoritma tersebut yakni $T_2 - T_1 = n(n-1) - 2^i$, dengan T_1 adalah algoritma Brute Force, dan T_2 adalah algoritma Divide and Conquer. Jika $T_2 - T_1 > 0$, maka $n(n-1) > 2^i$ dan algoritma Brute Force lebih cepat. Jika $T_2 - T_1 < 0$, maka $n(n-1) < 2^i$ dan algoritma Divide and Conquer lebih cepat. Ini membuktikan bahwa efisiensi relatif kedua algoritma bergantung pada banyaknya jumlah titik kontrol (n) dan jumlah iterasi (i). Memungkinkan jika titik kontrol sedikit tetapi iterasi banyak, maka algoritma Divide and Conquer lebih cepat, dan sebaliknya jika titik kontrol banyak tetapi iterasi sedikit maka algoritma Brute Force lebih cepat.

Namun, dalam praktiknya, hasil pengujian mungkin tidak selalu konsisten dengan ekspektasi teoretis ini. Hal ini bisa disebabkan oleh implementasi algoritma yang tidak efisien, baik dalam hal penyimpanan maupun pengaksesan data, yang dapat mengakibatkan overhead yang tidak diinginkan. Mesin tempat algoritma dijalankan juga memainkan peran kritis dalam menentukan kecepatan eksekusi. Dua implementasi algoritma yang sama dapat memiliki kinerja yang sangat berbeda pada hardware yang berbeda. Oleh karena itu, ketika mengevaluasi atau membandingkan efisiensi algoritma, penting untuk mempertimbangkan karakteristik mesin yang digunakan.

4.4. Penjelasan Bonus

Kurva Bézier dengan n titik kontrol dapat berjalan pada program ini, tidak begitu banyak perbedaan algoritma antara kurva Bézier 3 titik kontrol (kuadratik) dengan n titik kontrol. Perbedaannya berupa untuk n buah titik kontrol, jumlah operasi yang berjalan akan semakin banyak karena jumlah operasi adalah n ditambah (n - 1) ditambah (n -2) dan seterusnya hingga n bernilai satu.

Visualisasi kurva Bézier dilakukan dengan memanfaatkan library milik python yaitu matplotlib untuk memudahkan penggambarannya. Visualisasi pada file *main.py* hanya menggambarkan penggunaan algoritma Divide and Conquer, sedangkan untuk file *test.py* menggambarkan penggunaan algoritma Divide and Conquer dan algoritma Brute Force.

BAB V
PENUTUP

5.1. Kesimpulan

Membanguin kurva bézier dengan sejumlah titik dan iterasi dapat dilakukan baik dengan algoritma Brute Force maupun algoritma Divide and Conquer. Dari hasil analisis yang didapat dari kedua algoritma tersebut menjelaskan bahwa penggunaan algoritma Divide and Conquer lebih bagus digunakan pada persoalan yang berukuran besar, sedangkan algoritma Brute Force lebih bagus digunakan pada persoalan yang lebih sederhana. Perbedaan performa dari kedua algoritma tersebut tidak terlalu besar berdasarkan hasil analisis kami sehingga kedua algoritma masih dapat digunakan dalam skenario apapun. Faktor bahasa pemrograman, optimisasi kode, atau performa dari laptop/PC dapat berpengaruh terhadap kecepatan dari algoritma Divide and Conquer dan algoritma Brute Force.

5.2. Saran

Implementasi pembangunan kurva Bézier yang kami buat masih dapat dikembangkan lebih lanjut untuk memaksimalkan kesangkilannya untuk kedua jenis algoritma. Peningkatan kesangkilan dapat dilakukan dengan memaksimalkan iterasi yang ada di dalam kode dan/atau menggunakan bahasa pemrograman lain yang lebih sangkil dan efektif.

DAFTAR REFERENSI

Munir, Rinaldi. “Algoritma Brute Force (Bagian 1).” Informatika. Diakses 15 Maret 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf

Munir, Rinaldi. “Algoritma Divide and Conquer (Bagian 1).” Informatika. Diakses 15 Maret 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian1.pdf

Munir, Rinaldi. “Algoritma Divide and Conquer (Bagian 2).” Informatika. Diakses 15 Maret 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian2.pdf

Munir, Rinaldi. “Algoritma Divide and Conquer (Bagian 3).” Informatika. Diakses 15 Maret 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian3.pdf

Munir, Rinaldi. “Algoritma Divide and Conquer (Bagian 4).” Informatika. Diakses 15 Maret 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian4.pdf

https://p2k.stekom.ac.id/ensiklopedia/Kurva_B%C3%A9zier

LAMPIRAN

Link Github: https://github.com/Bodleh/Tucil2_13522073_13522111.git

Tabel Kelayakan Program

Poin	Ya	Tidak
1. Program berhasil dijalankan	✓	
2. Program dapat melakukan visualisasi kurva Bézier.	✓	
3. Solusi yang diberikan program optimal.	✓	
4. **[Bonus]** Program dapat membuat kurva untuk n titik kontrol.	✓	
5. **[Bonus]** Program dapat melakukan visualisasi proses pembuatan kurva.	✓	