TUGAS KECIL 2

IF2211 STRATEGI ALGORITMA

SEMESTER II TAHUN 2023/2024

"Aproksimasi Kurva Bezier dengan Algoritma *Divide and Conquer*"

OLEH:

Irfan Sidiq Permana 13522007

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

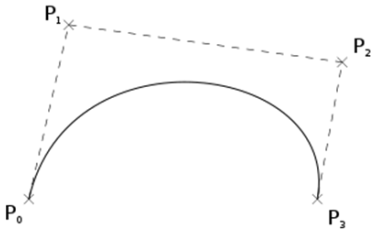
INSTITUT TEKNOLOGI BANDUNG

2024

BAB I
DESKRIPSI MASALAH

1.1. Abstraksi

Kurva Bezier adalah kurva yang sangat sering digunakan dalam desain grafis, animasi, dan desain rekayasa. Kurva ini dibuat dengan menetapkan beberapa titik kontrol yang menentukan arah dan lengkungan kurva. Kurva berawal dari titik kontrol pertama dan bergerak melengkung mengikuti arah posisi relatif titik kontrol lain (walaupun tidak tepat mengenai titik kontrol), sebelum akhirnya berhenti tepat pada titik kontrol terakhir. Kurva Bezier merupakan salah satu penemuan terpenting dalam dunia desain digital dan memiliki banyak aplikasi di dunia nyata, seperti *pen tool*, animasi yang halus dan realistis, membuat desain produk yang kompleks dan presisi, dan membuat font yang indah dan unik. Kelebihan dari kurva bezier ini adalah kurva dapat dibuat serta dimanipulasi dengan mudah serta memiliki presisi yang tinggi.



Gambar 1.1 Contoh Kurva Bezier dengan 4 titik kontrol

Pada Tugas Kecil 2 ini, penulis merancang program untuk menerapkan algoritma Divide and Conquer yang telah dipelajari di kelas untuk melakukan aproksimasi hasil kurva bezier. Program juga memiliki opsi untuk melakukan aproksimasi menggunakan algoritma brute force, sehingga dapat dibandingkan antara proses perhitungan secara *brute force* dengan secara *divide and conquer* serta waktu eksekusi masing-masing. Program kecil ini dibuat menggunakan bahasa Python dan dijalankan melalui CLI, serta menerima input berupa: 1) Pilihan algoritma yang digunakan, yaitu pendekatan *divide and conquer* atau *brute force*, 2) Jumlah titik kontrol *n* (harus lebih dari sama dengan 3 titik), 3) Jumlah iterasi (harus lebih dari sama dengan 1 iterasi), serta 4) Koordinat masing-masing titik kontrol, yaitu (x_1, y_1), (x_2, y_2), …, (x_n, y_n). Setelah menerima input, program akan melakukan kalkulasi lalu menampilkan animasi pembentukan aproksimasi kurva bezier menggunakan kakas *matplotlib* sesuai dengan pendekatan algoritma yang dipilih.

BAB II
IMPLEMENTASI ALGORITMA

2.1. Algoritma *Divide and Conquer*

Algoritma *divide and conquer* adalah salah satu pendekatan algoritma untuk menyelesaikan berbagai persoalan komputasi. Ide utama dari algoritma ini yaitu membagi persoalan (*divide*) menjadi beberapa upa-persoalan hingga upa-persoalan tersebut dapat diselesaikan secara langsung (*solve*), lalu kemudian menggabungkan (*combine*) semua solusi dari upa-persoalan tersebut untuk menjawab persoalan awal. Kelebihan dari pendekatan ini adalah implementasi dari algoritmanya umumnya intuitif, dan memiliki kompleksitas waktu yang efisien untuk beberapa persoalan sehingga menjadi pendekatan yang standar untuk persoalan tersebut. Sebagai pembanding, algoritma *divide and conquer* biasanya lebih efisien dari algoritma *brute force* untuk persoalan tertentu dilihat dari kompleksitas waktunya. Karena sifatnya yang terus membagi upa-persoalan hingga ukurannya cukup kecil untuk langsung diselesaikan, maka pendekatan ini umumnya diimplementasikan secara rekursif.

Pada program ini, algoritma *divide and conquer* untuk mengaproksimasi kurva bezier diimplementasikan dengan tahapan berikut. Misalkan terdapat 3 buah titik kontrol, yaitu P_1, P_2, dan P_3, dengan P_2 sebagai titik antara serta jumlah aproksimasi sebanyak 2 kali.

1) Pertama, buat garis yang menghubungkan setiap titik kontrol secara berurutan, yaitu garis yang menghubungkan P_1 dengan P_2 serta garis yang menghubungkan P_2 dengan P_3.
2) Buat titik yang berada di tengah masing-masing garis, misalnya Q_1 yang berada di tengah garis yang menghubungkan P_1 dengan P_2, serta Q_2 yang berada di tengah garis yang menghubungkan P_2 dengan P_3.
3) Kemudian, buat kembali garis yang menghubungkan setiap titik tengah tersebut, yaitu garis yang menghubungkan Q_1 dengan Q_2. Serta buat kembali titik yang berada di tengah garis tersebut, misalnya R_1.
4) Buat garis yang menghubungkan titik kontrol awal yaitu P_1 dengan titik R_1, serta garis yang menghubungkan R_1 dengan titik kontrol akhir yaitu P_3. Kedua garis inilah yang menjadi hasil dari aproksimasi kurva bezier pada iterasi pertama.
5) Untuk iterasi selanjutnya, ulangi keempat langkah di atas dengan 2 kumpulan titik-titik yang sebelumnya terbentuk, yaitu titik (P_1, Q_1, R_1), serta titik (R_1, Q_2, P_3). Setelah

dilakukan kembali keempat langkah pada kedua kumpulan titik diatas, maka diperoleh hasil aproksimasi kurva bezier pada iterasi kedua.

Konsep pendekatan *divide and conquer* yang dicerminkan pada tahapan-tahapan diatas yaitu:

1) Memproses (*conquer*) persoalan awal (P_1, P_2, P_3) untuk memperoleh solusi awal R_1,
2) Membagi (*divide*) persoalan tersebut menjadi dua upa-persoalan (P_1, Q_1, R_1) dan (R_1, Q_2, P_3) dimana masing-masing upa-persoalan kemudian menghasilkan solusi upa-persoalan R_2 dan R_3 (*solve*),
3) Menggabungkan (*combine*) solusi R_2 dan R_3 dengan solusi R_1 sebelumnya melalui pembuatan garis yang menghubungkan R_1 dengan R_2 serta R_2 dengan R_3 untuk memperoleh jawaban dari persoalan awal, yaitu aproksimasi kurva bezier untuk titik kontrol (P_1, P_2, P_3) dengan iterasi sebanyak 2 kali.

Proses pembentukan aproksimasi kurva diatas dapat digeneralisasi untuk n buah titik kontrol dengan iterasi sebanyak c kali. Misalkan terdapat n buah titik kontrol yaitu (P_1, P_2, P_3, …, P_n) dan akan dilakukan iterasi sebanyak c kali.

1) Buat garis yang menghubungkan tiap pasangan titik kontrol secara berurutan, yaitu garis yang menghubungkan P_1 dengan P_2, P_2 dengan P_3, P_3 dengan P_4, dan seterusnya hingga $P_{n\text{-}1}$ dengan P_n.
2) Buat titik tengah (A_{11}, A_{12}, A_{13}, …, $A_{1(n\text{-}1)}$) yang masing-masing titik A_{1k} berada pada tengah garis yang menghubungkan titik antara P_k dengan P_{k+1}. Contohnya, A_{11} berada pada garis yang menghubungkan P_1 dengan P_2, A_{12} berada pada garis yang menghubungkan P_2 dengan P_3, dan seterusnya.
3) Buat kembali garis yang menghubungkan tiap pasangan titik (A_{11}, A_{12}, A_{13}, …, $A_{1(n\text{-}1)}$) secara berurutan, yaitu garis yang menghubungkan A_{11} dengan A_{12}, A_{12} dengan A_{23}, dan seterusnya.
4) Buat kembali titik tengah (A_{21}, A_{22}, A_{23}, …, $A_{2(n\text{-}2)}$) yang masing-masing titik A_{2k} berada pada tengah garis yang menghubungkan titik antara A_{1k} dengan $A_{1(k+1)}$. Contohnya, A_{21} berada pada garis yang menghubungkan A_{11} dengan A_{12}, A_{22} berada pada garis yang menghubungkan A_{12} dengan A_{13}, dan seterusnya.
5) Kedua langkah diatas (membuat garis dan titik tengah) terus dilakukan sebanyak n-1 kali hingga diperoleh satu titik terakhir, yaitu $A_{(n\text{-}1)1}$.

6) Sebagai langkah terakhir dari iterasi pertama, buat garis yang menghubungkan antara titik kontrol awal P_1 dengan titik $A_{(n\text{-}1)1}$ serta garis yang menghubungkan antara titik $A_{(n\text{-}1)1}$ dengan titik kontrol akhir P_n. Garis inilah yang menjadi hasil akhir dari iterasi pertama.
7) Untuk iterasi selanjutnya, lakukan kembali langkah-langkah diatas untuk dua kelompok titik yang masing-masing berjumlah *n* titik, yaitu (P_1, A_{11}, A_{21}, A_{31}, …, $A_{(n\text{-}1)1}$) dan ($A_{(n\text{-}1)1}$, $A_{(n\text{-}2)2}$, $A_{(n\text{-}3)3}$, …, $A_{1(n\text{-}1)}$, P_n). Hasil akhir dari dua kelompok titik tersebut adalah hasil akhir dari iterasi kedua.

8) Proses diatas dapat dilakukan sebanyak *c* kali untuk mendapatkan hasil akhir dari iterasi ke-*c*.

2.2. Algoritma *Brute Force*

Sebagai pembanding, diimplementasikan pula algoritma aproksimasi kurva bezier menggunakan pendekatan *brute force*. Algoritma *brute force* adalah pendekatan yang lempang (*straightforward*) untuk memecahkan suatu persoalan. Algoritma *brute force* umumnya didasarkan pada pernyataan pada persoalan (*problem statement*), dan definisi/konsep yang dilibatkan. Algoritma *brute force* memecahkan persoalan dengan sangat sederhana, langsung, dan jelas caranya (*obvious way*). Kelemahan utamanya dari algoritma *brute force* yaitu umumnya tidak mangkus dan membutuhkan volume komputasi yang besar.

Pada program ini, algoritma *brute force* sebagai algoritma pembanding untuk mengaproksimasi kurva bezier diimplementasikan dengan memanfaatkan rumus parametrik kurva bezier kuadratik, yaitu:

$$\mathbf{B}(t) = (1 - t)^2 \mathbf{P}_1 + 2 (1 - t) t \mathbf{P}_2 + t^2 \mathbf{P}_3, \quad 0 \le t \le 1$$

Misalkan terdapat 3 buah titik kontrol, yaitu P_1, P_2, dan P_3, dengan P_2 sebagai titik antara. Maka untuk mendapatkan titik R_1 seperti pada pendekatan *divide and conquer* sebelumnya, caranya cukup dengan mensubstitusikan nilai t = 0,5 pada persamaan diatas. Rumus diatas juga dapat digeneralisasi untuk *n* buah titik kontrol, yaitu:

$$\begin{aligned}\mathbf{B}(t) &= \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i \\ &= (1-t)^n \mathbf{P}_0 + \binom{n}{1} (1-t)^{n-1} t \mathbf{P}_1 + \cdots +\end{aligned}$$

$$\binom{n}{n-1}(1-t)t^{n-1}\mathbf{P}_{n-1} + t^n\mathbf{P}_n, \qquad 0 \leqslant t \leqslant 1$$

Implementasi dari rumus diatas untuk pendekatan *brute force* dirancang pada program dengan tahapan sebagai berikut. Misalkan terdapat *n* buah titik kontrol yaitu (P_1, P_2, P_3, …, P_n) dan akan dilakukan iterasi sebanyak *c* kali.

1) Inisialisasi nilai (R_1, R_2, R_3, …, R_k) masing-masing bernilai 0, dengan $k = 2^C + 1$. Dengan kata lain, iterasi sebanyak *c* kali akan menghasilkan $2^C + 1$ titik akhir (termasuk titik awal dan titik akhir kurva).
2) Inisialisasi $R_1 = P_1$ sebagai titik awal dan $R_k = P_n$ sebagai titik akhir kurva.
3) Hitung nilai koefisien setiap suku dalam persamaan terlebih dahulu, dimana tiap suku ke-*i* memiliki nilai koefisien $\binom{n}{i}$.
4) Hitung nilai $\triangle t$, dimana $\triangle t = 1 / 2^C$ (*c* adalah jumlah iterasi). Inisialisasi nilai *t* dengan $t = \triangle t$
5) Hitung nilai $(1-t)^{n-i}t^i$, mulai dari $i = 1$ hingga $i = n$. Kemudian, kalikan masing-masing hasilnya dengan koefisien yang telah dihitung tadi (sehingga diperoleh nilai $\binom{n}{i}(1-t)^{n-i}t^i$).
6) Jumlahkan nilai $\binom{n}{i}(1-t)^{n-i}t^i\mathbf{P}_i$ mulai dari $i = 1$ hingga $i = n$, kemudian *assign* nilai tersebut menjadi nilai R_2 yang baru.
7) Untuk memperoleh nilai R_3, ubah nilai *t* sebelumnya menjadi $t = 2$ lalu lakukan kembali langkah 5 dan 6. Begitu juga untuk R_4, R_5, dan seterusnya hingga R_{k-1}.
8) Sebagai langkah terakhir, tarik garis yang menghubungkan tiap pasangan titik (R_1, R_2, R_3, …, R_k) secara berurutan, yaitu R_1 dengan R_2, R_2 dengan R_3, dan seterusnya hingga $R_{(k-1)}$ dengan R_k. Hasil garis inilah yang menjadi hasil akhir aproksimasi kurva pada iterasi ke-*c*.

BAB III
IMPLEMENTASI KODE PROGRAM

3.1 Implementasi Kode *Main Program*

Berikut merupakan implementasi kode program utama yaitu “main.py”:

main.py

```
# ====================================================
# Input data
# ====================================================

prompt_method_choice = """
Pilih metode yang digunakan:
------------------------------
| 1. Divide and Conquer      |
| 2. Brute Force             |
------------------------------

Pilihan metode: """
method_choice = int(InputValidation.input_number_validation(prompt_method_choice, "integer", 1, 2))
num_of_control_points = int(InputValidation.input_number_validation("\nMasukkan jumlah titik kontrol: ", "integer",
num_of_iteration = int(InputValidation.input_number_validation("\nMasukkan jumlah iterasi: ", "integer", 1))

control_points = np.zeros((num_of_control_points, 2))
print(f"\nMasukkan koordinat {num_of_control_points} titik:")
for i in range(num_of_control_points):
    control_points[i][0] = float(InputValidation.input_number_validation(f"x{i+1}: "))
    control_points[i][1] = float(InputValidation.input_number_validation(f"y{i+1}: "))
```

```
# ====================================================
# Process data
# ====================================================

plot = Plot(num_of_iteration, subplot_=111, figsize_=(800/120, 800/120), dpi_=120)
plot.plot(control_points)

execution_time = ExecutionTime()
bezier_curve = BezierCurve(num_of_iteration, control_points)

if method_choice == 1:
    bezier_curve.divide_and_conquer(num_of_iteration, control_points, 0, len(bezier_curve.result_points)-1, plot, execution_time)
    plot.remove_saved_lines()
else:
    bezier_curve.brute_force(plot, execution_time)

plot.plot(bezier_curve.result_points)
plot.show()

print(f"\nWaktu eksekusi: {execution_time.total_time} ms")
print("\nTerima kasih dan sampai jumpa lagi!")
```

3.2 Implementasi Kode *Divide and Conquer*

Berikut merupakan implementasi kode dari algoritma *divide and conquer* yang terletak pada file "bezier_curve.py":

bezier_curve.py

```
def divide_and_conquer(self, num_of_iteration: int, control_points: List[Point], l: int, r: int, plot: Plot, execution_tim
    if num_of_iteration == 0:
        return

    execution_time.start()
    next_control_points_1, next_control_points_2 = np.zeros((len(control_points), 2)), np.zeros((len(control_points), 2))
    next_control_points_1[0] = control_points[0]
    next_control_points_2[len(next_control_points_2)-1] = control_points[len(control_points)-1]
    execution_time.stop()

    # Variables for displaying dots and lines
    temp = np.zeros((math.ceil(len(control_points)*(len(control_points)-1)/2), 2))
    temp_idx, last_temp_idx = 0, 0
    current_iteration = self.num_of_iteration - num_of_iteration
```

```
for i in range(1, len(control_points)):
    for j in range(len(control_points)-i):
        execution_time.start()
        control_points[j] = self.find_middle_point(control_points[j], control_points[j+1])
        execution_time.stop()

        if current_iteration < 4:
            plot.add_dot(control_points[j], color_='0.5')
            temp[temp_idx] = control_points[j]
            temp_idx += 1

    # Display dashed lines
    if current_iteration < 4:
        delay_time = current_iteration+2 if current_iteration <= 2 else current_iteration**2
        plot.pause(1 / 2**delay_time)
        if temp_idx - last_temp_idx > 1:
            line = temp[last_temp_idx:temp_idx].transpose()
            plot.add_line(line, linestyle_='dashed', color_='0.5')
            plot.pause(1 / 2**delay_time)

    execution_time.start()
    next_control_points_1[i] = control_points[0]
    next_control_points_2[len(next_control_points_2)-i-1] = control_points[len(control_points)-i-1]
    execution_time.stop()

    last_temp_idx = temp_idx
```

```
plot.remove_saved_dots()
plot.add_dot(control_points[0], None, save=False)

execution_time.start()
mid = (l+r) // 2
self.result_points[mid] = control_points[0]
execution_time.stop()

self.divide_and_conquer(num_of_iteration-1, next_control_points_1, l, mid, plot, execution_time)
self.divide_and_conquer(num_of_iteration-1, next_control_points_2, mid+1, r, plot, execution_time)
```

3.3 Implementasi Kode *Brute Force*

Berikut merupakan implementasi kode dari algoritma *brute force* yang terletak pada file "bezier_curve.py":

bezier_curve.py

```
def brute_force(self, plot: Plot, execution_time: ExecutionTime):
    execution_time.start()
    coefficient = np.zeros(len(self.control_points))
    for i in range(len(self.control_points)):
        coefficient[i] = math.comb(len(self.control_points)-1, i)
    t = 0
    delta_t = 1 / 2**self.num_of_iteration
    execution_time.stop()

    for i in range(1, 2**self.num_of_iteration):
        execution_time.start()
        t += delta_t
        for j in range(len(self.control_points)):
            coefficient_result = coefficient[j] * (1-t)**(len(self.control_points)-j-1) * (t**j)
            self.result_points[i] += coefficient_result * self.control_points[j]
        execution_time.stop()

        plot.add_dot(self.result_points[i])
        plot.pause(1 / 2**self.num_of_iteration)
```

BAB IV
HASIL PENGUJIAN DAN ANALISIS

4.1 Hasil Pengujian Algoritma

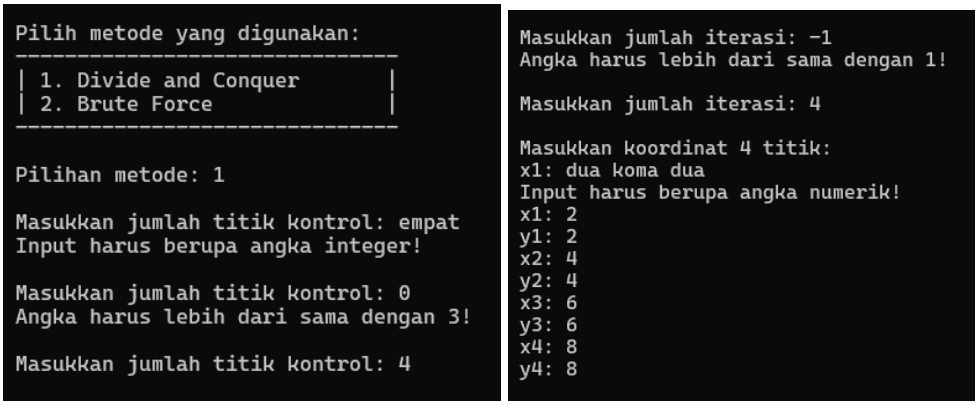
Program dijalankan dengan memasukkan command “python main.py” pada folder src. Tampilan awal program adalah sebagai berikut:

Gambar 4.1 Tampilan awal program

Program akan meminta input berupa:

1) Pilihan algoritma yang digunakan, yaitu pendekatan *divide and conquer* atau *brute force*,
2) Jumlah titik kontrol *n* (harus lebih dari sama dengan 3 titik),
3) Jumlah iterasi *c* (harus lebih dari sama dengan 1 iterasi), serta
4) Koordinat masing-masing titik kontrol, yaitu (x_1, y_1), (x_2, y_2), …, (x_n, y_n).

Untuk setiap input akan dilakukan validasi apakah input memiliki tipe data yang valid atau tidak, serta apakah input yang dimasukkan berada pada *range* yang sesuai atau tidak. Berikut contoh alur validasi input program:



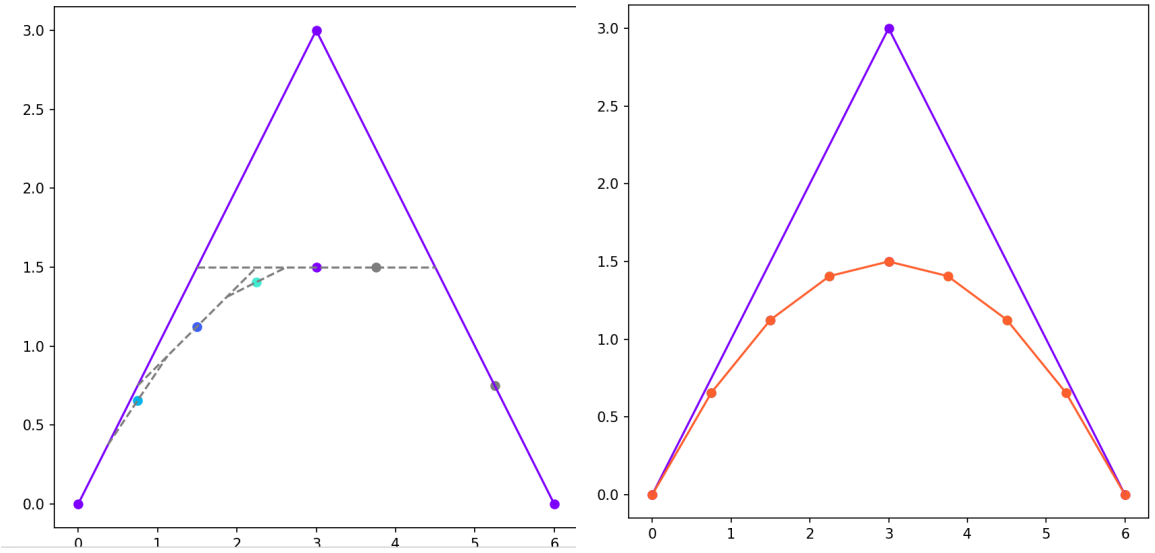
Gambar 4.2 Alur Validasi Input Program

Untuk melakukan pengetesan algoritma aproksimasi kurva bezier, dilakukan uji coba dengan 6 data uji yang masing-masing diujikan baik dengan algoritma *divide and conquer* maupun algoritma *brute force*.

1) Data Uji 1

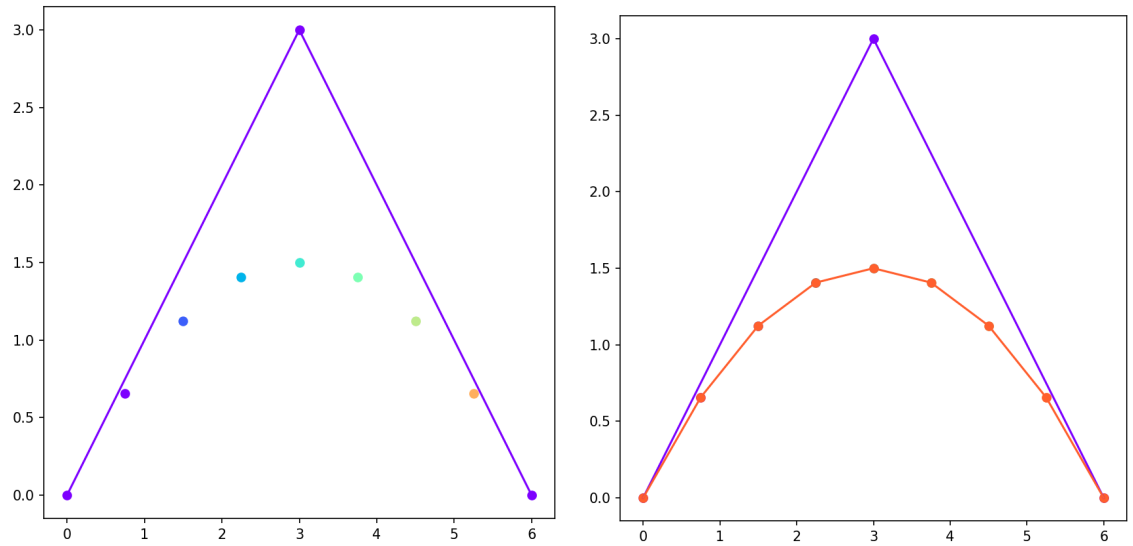
```
Masukkan jumlah titik kontrol: 3

Masukkan jumlah iterasi: 3

Masukkan koordinat 3 titik:
x1: 0
y1: 0
x2: 3
y2: 3
x3: 6
y3: 0
```

Gambar 4.3.1.1 Data Uji 1




```
Waktu eksekusi: 0.0 ms
```

Gambar 4.3.1.2 Hasil Data Uji 1 menggunakan pendekatan divide and conquer

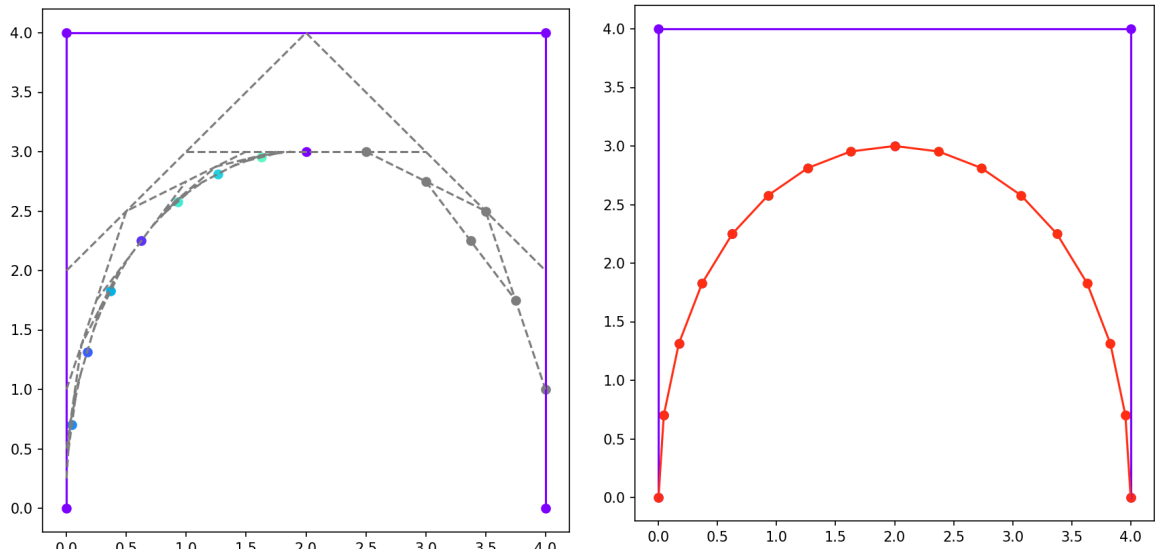


Gambar 4.3.1.3 Hasil Data Uji 1 menggunakan pendekatan *brute force*

2) Data Uji 2

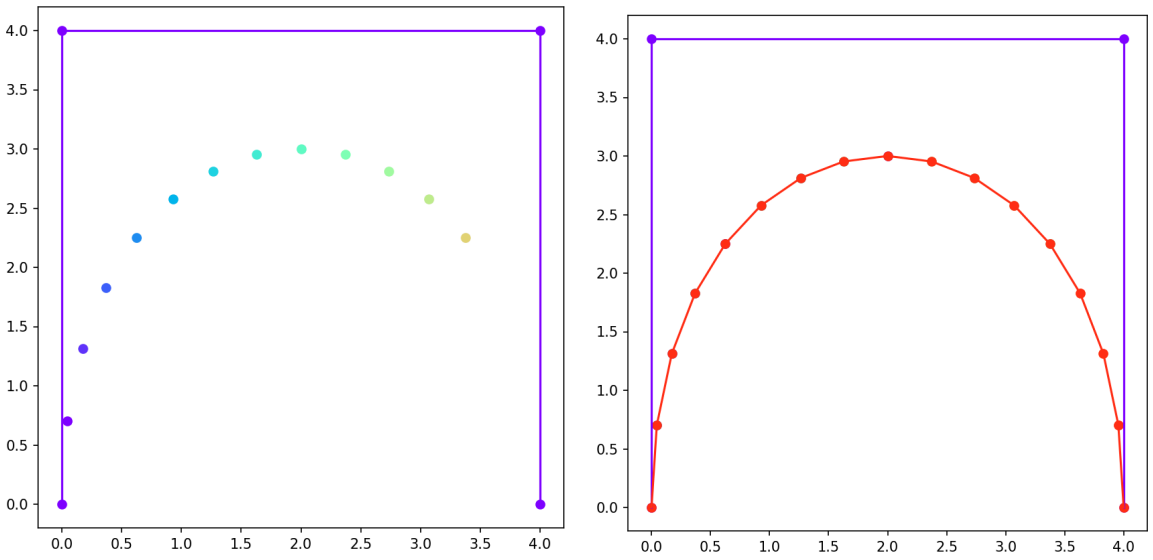
```
Masukkan jumlah titik kontrol: 4

Masukkan jumlah iterasi: 4

Masukkan koordinat 4 titik:
x1: 0
y1: 0
x2: 0
y2: 4
x3: 4
y3: 4
x4: 4
y4: 0
```

Gambar 4.3.2.1 Data Uji 2



```
Waktu eksekusi: 2.014892578125 ms
```

Gambar 4.3.2.2 Hasil Data Uji 2 menggunakan pendekatan divide and conquer



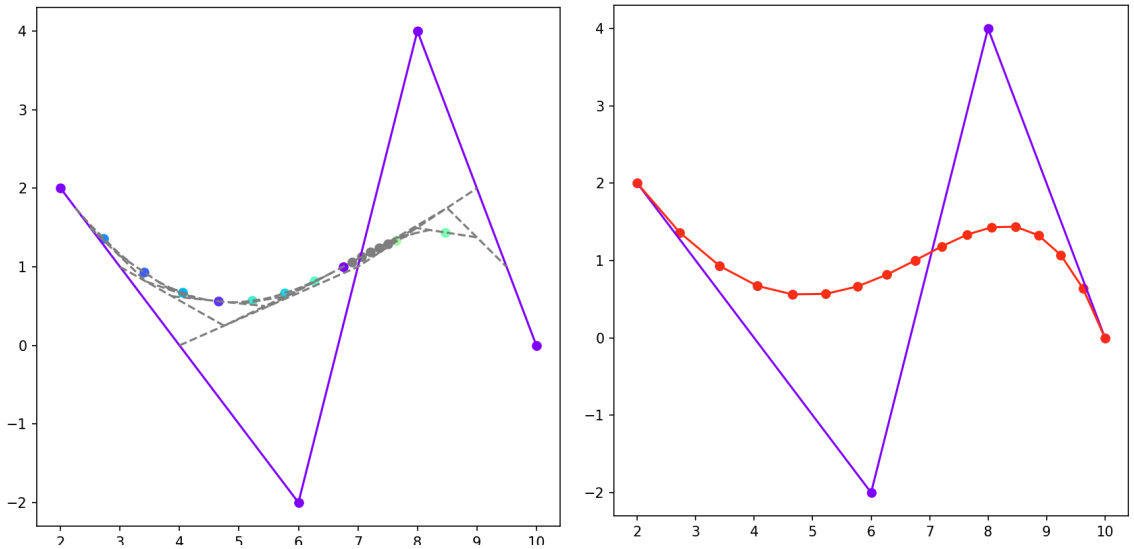

```
Waktu eksekusi: 0.992431640625 ms
```

Gambar 4.3.2.3 Hasil Data Uji 2 menggunakan pendekatan brute force

3) Data Uji 3

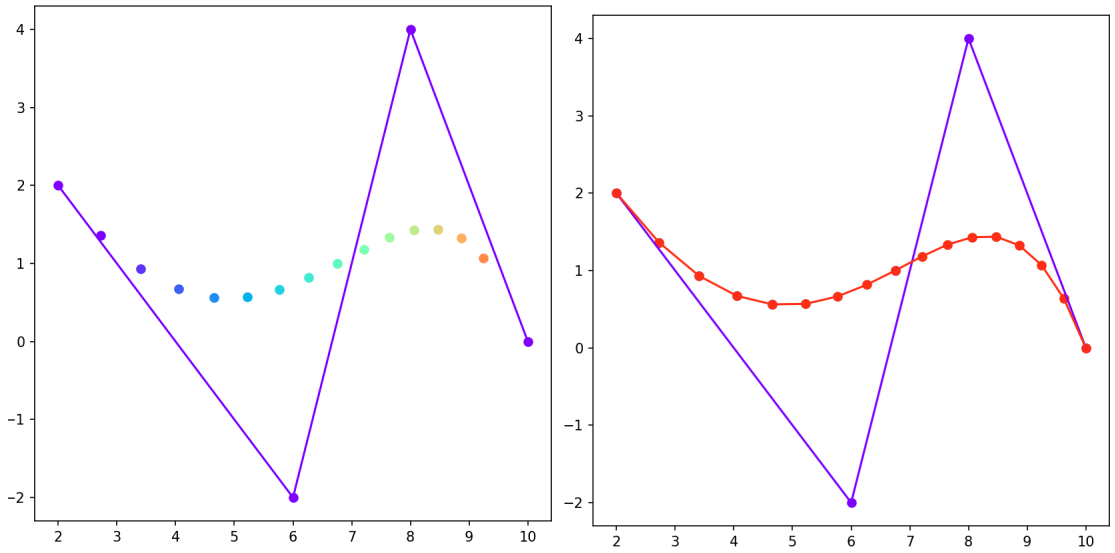
```
Masukkan jumlah titik kontrol: 4

Masukkan jumlah iterasi: 4

Masukkan koordinat 4 titik:
x1: 2
y1: 2
x2: 6
y2: -2
x3: 8
y3: 4
x4: 10
y4: 0
```

Gambar 4.3.3.1 Data Uji 3



```
Waktu eksekusi: 0.992431640625 ms
```

Gambar 4.3.3.2 Hasil Data Uji 3 menggunakan pendekatan divide and conquer



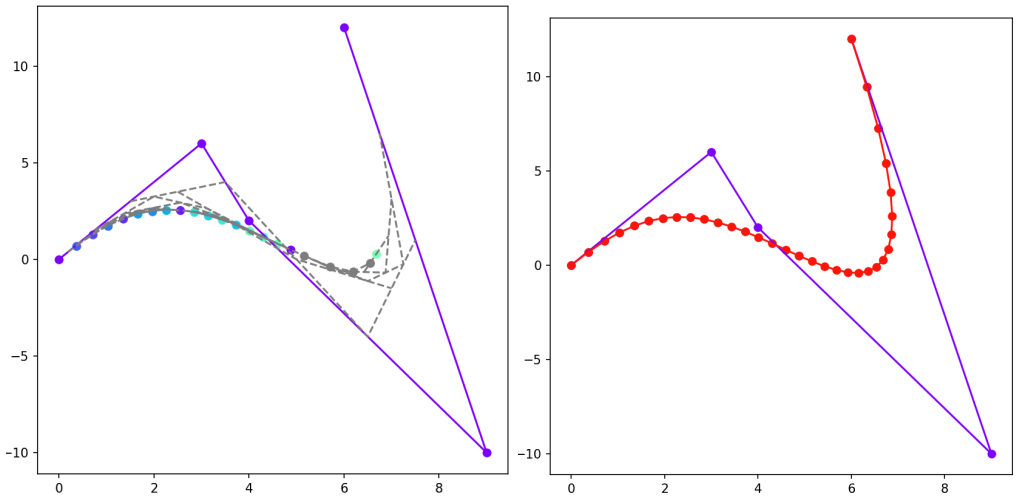
```
Waktu eksekusi: 1.0185546875 ms
```

Gambar 4.3.3.3 Hasil Data Uji 3 menggunakan pendekatan brute force

4) Data Uji 4

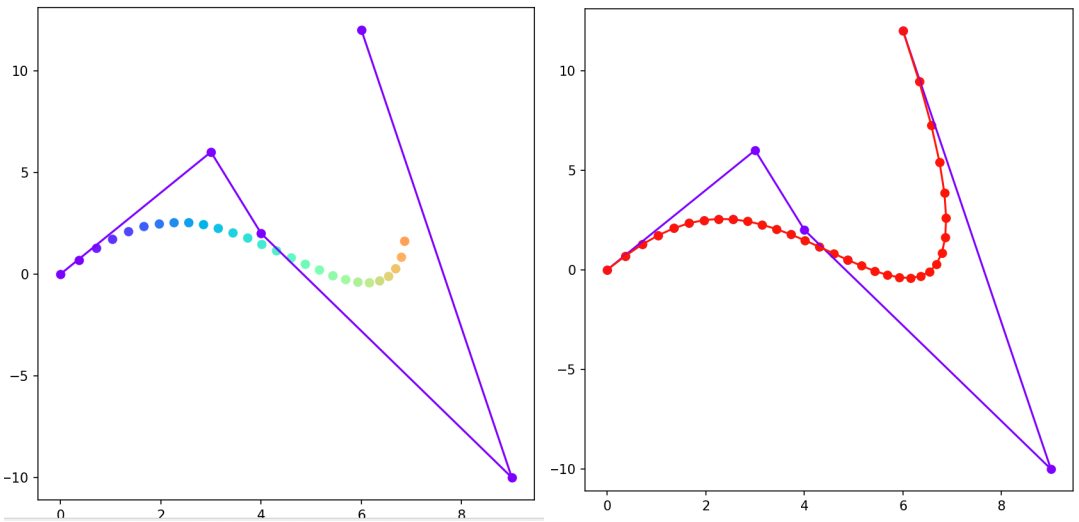
```
Masukkan jumlah titik kontrol: 5

Masukkan jumlah iterasi: 5

Masukkan koordinat 5 titik:
x1: 0
y1: 0
x2: 3
y2: 6
x3: 4
y3: 2
x4: 9
y4: -10
x5: 6
y5: 12
```

Gambar 4.3.4.1 Data Uji 4



```
Waktu eksekusi: 2.007080078125 ms
```

Gambar 4.3.4.2 Hasil Data Uji 4 menggunakan pendekatan divide and conquer



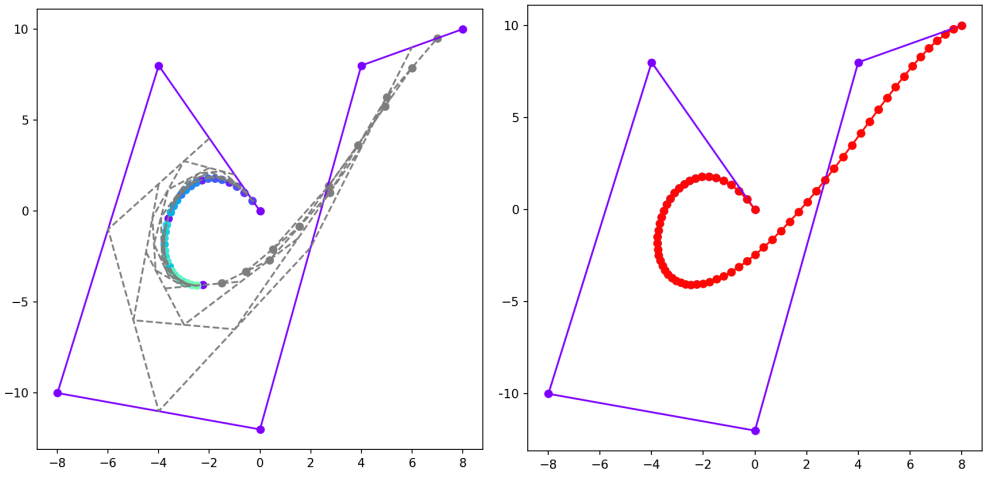
```
Waktu eksekusi: 3.999755859375 ms
```

Gambar 4.3.4.3 Hasil Data Uji 4 menggunakan pendekatan brute force

5) Data Uji 5

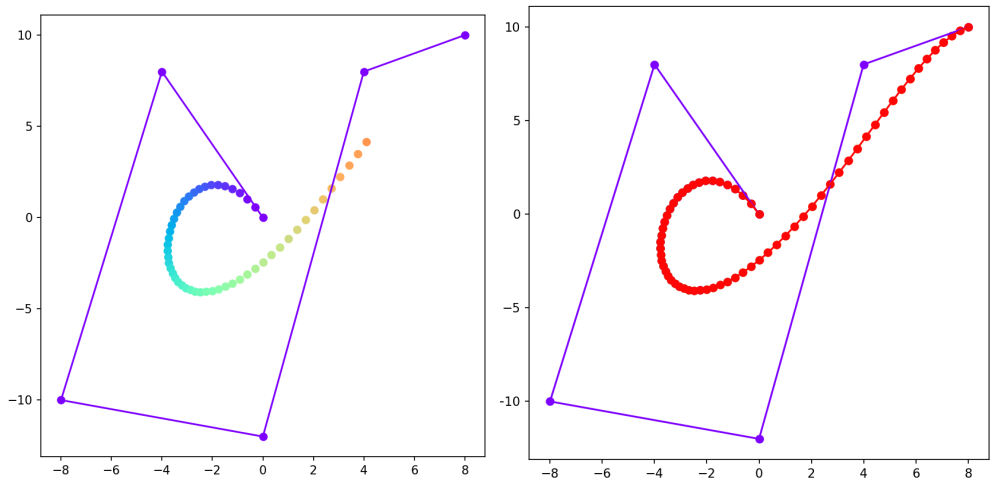
```
Masukkan jumlah titik kontrol: 6

Masukkan jumlah iterasi: 6

Masukkan koordinat 6 titik:
x1: 0
y1: 0
x2: -4
y2: 8
x3: -8
y3: -10
x4: 0
y4: -12
x5: 4
y5: 8
x6: 8
y6: 10
```

Gambar 4.3.5.1 Data Uji 5



```
Waktu eksekusi: 2.00244140625 ms
```

Gambar 4.3.5.2 Hasil Data Uji 5 menggunakan pendekatan divide and conquer



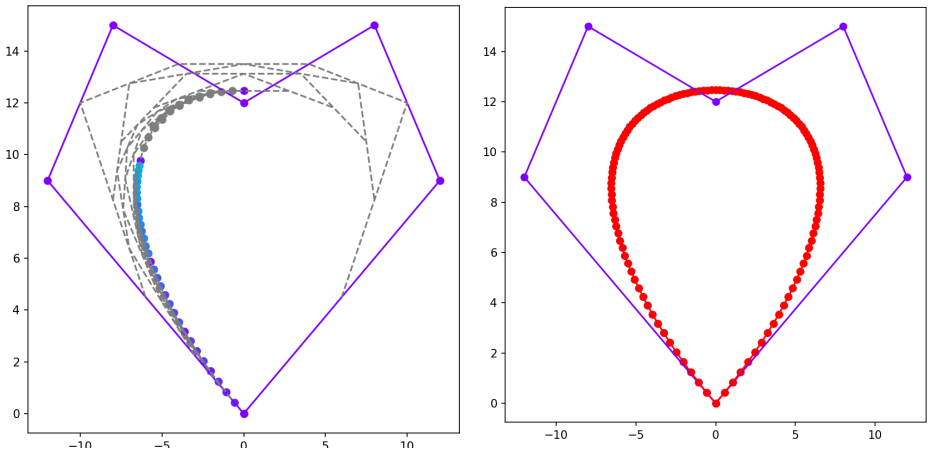
```
Waktu eksekusi: 3.97607421875 ms
```

Gambar 4.3.5.3 Hasil Data Uji 5 menggunakan pendekatan brute force

6) Data Uji 6

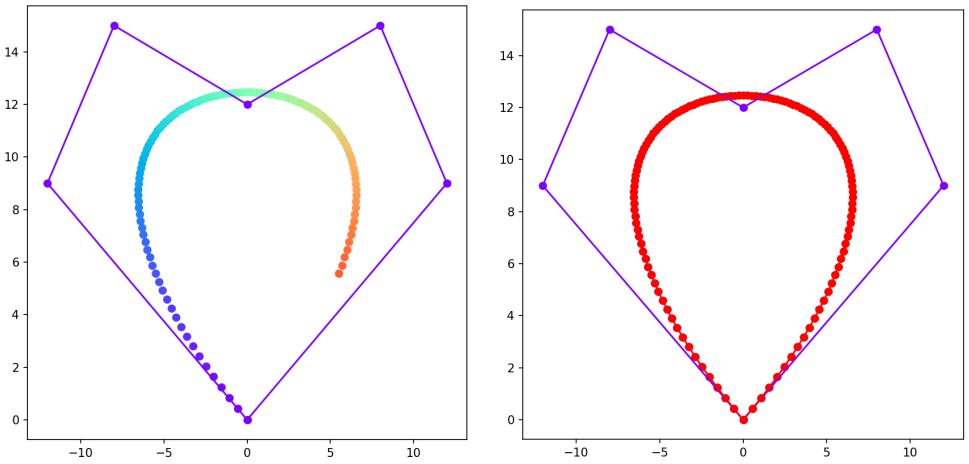
```
Masukkan jumlah titik kontrol: 7

Masukkan jumlah iterasi: 7

Masukkan koordinat 7 titik:
x1: 0
y1: 0
x2: -12
y2: 9
x3: -8
y3: 15
x4: 0
y4: 12
x5: 8
y5: 15
x6: 12
y6: 9
x7: 0
y7: 0
```

Gambar 4.3.5.1 Data Uji 6



```
Waktu eksekusi: 6.999267578125 ms
```

Gambar 4.3.5.2 Hasil Data Uji 6 menggunakan pendekatan divide and conquer



```
Waktu eksekusi: 6.8818359375 ms
```

Gambar 4.3.5.3 Hasil Data Uji 6 menggunakan pendekatan brute force

4.2 Analisis Kompleksitas Algoritma

Setelah melakukan uji coba dengan beberapa data uji, diperoleh hasil gambar aproksimasi kurva dan waktu eksekusi masing-masing algoritma. Sehingga dapat dilakukan analisis pada implementasi kedua algoritma untuk membandingkan apakah hasil dari uji coba sesuai dengan perkiraan dari implementasi algoritma. Perbandingan ini dapat dilakukan dengan menggunakan kompleksitas waktu. Misalkan jumlah iterasi dinotasikan dengan *n*, dan jumlah titik kontrol dinotasikan dengan *k*.

```python
def brute_force(self, plot: Plot, execution_time: ExecutionTime):
    execution_time.start()
    coefficient = np.zeros(len(self.control_points))
    for i in range(len(self.control_points)):
        coefficient[i] = math.comb(len(self.control_points)-1, i)
    t = 0
    delta_t = 1 / 2**self.num_of_iteration
    execution_time.stop()

    for i in range(1, 2**self.num_of_iteration):
        execution_time.start()
        t += delta_t
        for j in range(len(self.control_points)):
            coefficient_result = coefficient[j] * (1-t)**(len(self.control_points)-j-1) * (t**j)
            self.result_points[i] += coefficient_result * self.control_points[j]
        execution_time.stop()
```

Gambar 4.2.1 Implementasi algoritma pendekatan brute force

Pada implementasi diatas, terlihat bahwa perhitungan yang paling mendominasi adalah perhitungan *coefficient_result*, yang menghitung nilai $\binom{n}{i}(1-t)^{n-i}t^i$. Nilai dari $\binom{n}{i}$ sudah dihitung sebelumnya, dimana perhitungan nilai kombinasi menggunakan '*math.comb()*' adalah O(*n*), sehingga kompleksitas waktunya dilakukan sebanyak *k* kali adalah O(*n***k*). Selanjutnya, bila dipandang dari jumlah perkalian, perpangkatan $a^n = a$ x a x a x ... memiliki jumlah operasi perkalian sebanyak *n*-1 kali, sehingga kompleksitas waktu dari perhitungan nilai $(1\text{-}t)^{n\text{-}i}\ t^i$ adalah O(*n*). Perhitungan ini dilakukan sebanyak $2^{n\text{-}1}$ * *k* kali (*loop* luar dikali *loop* dalam), sehingga kompleksitas totalnya adalah O(2^n **k** *n*). Karena perhitungan tersebut yang paling dominan dilakukan dalam implementasi algoritma, maka kompleksitas total algoritma brute force dilihat dari jumlah operasi perkaliannya adalah O(2^n * *k* * *n*).

```python
def divide_and_conquer(self, num_of_iteration: int, control_points: List[Point], l: int, r: int, plot: Plot, execution_time
    if num_of_iteration == 0:
        return

    execution_time.start()
    next_control_points_1, next_control_points_2 = np.zeros((len(control_points), 2)), np.zeros((len(control_points), 2))
    next_control_points_1[0] = control_points[0]
    next_control_points_2[len(next_control_points_2)-1] = control_points[len(control_points)-1]
    execution_time.stop()
```

```
for i in range(1, len(control_points)):
    for j in range(len(control_points)-i):
        execution_time.start()
        control_points[j] = self.find_middle_point(control_points[j], control_points[j+1])
        execution_time.stop()

        if current_iteration < 4:
            plot.add_dot(control_points[j], color_='0.5')
            temp[temp_idx] = control_points[j]
            temp_idx += 1

    # Display dashed lines
    if current_iteration < 4:
        delay_time = current_iteration+2 if current_iteration <= 2 else current_iteration**2
        plot.pause(1 / 2**delay_time)
        if temp_idx - last_temp_idx > 1:
            line = temp[last_temp_idx:temp_idx].transpose()
            plot.add_line(line, linestyle_='dashed', color_='0.5')
            plot.pause(1 / 2**delay_time)

    execution_time.start()
    next_control_points_1[i] = control_points[0]
    next_control_points_2[len(next_control_points_2)-i-1] = control_points[len(control_points)-i-1]
    execution_time.stop()

    last_temp_idx = temp_idx
```

```
execution_time.start()
mid = (l+r) // 2
self.result_points[mid] = control_points[0]
execution_time.stop()

self.divide_and_conquer(num_of_iteration-1, next_control_points_1, l, mid, plot, execution_time)
self.divide_and_conquer(num_of_iteration-1, next_control_points_2, mid+1, r, plot, execution_time)
```

```
def find_middle_point(self, p1: Point, p2: Point) -> Point:
    return (p1 + p2) / 2
```

Gambar 4.2.2 Implementasi algoritma pendekatan *divide and conquer*

Sedangkan pada implementasi *divide and conquer*, operasi yang paling mendominasi adalah perhitungan titik tengah garis. Perhitungan titik tengah garis memiliki kompleksitas waktu O(1). Untuk mendapatkan satu titik akhir aproksimasi diperlukan perhitungan titik tengah sebanyak $k(k\text{-}1)/2$ kali, sehingga total kompleksitas waktunya adalah $O(k^2)$. Tiap kali pemanggilan fungsi divide and conquer akan menghasilkan satu titik akhir, sehingga fungsi setidaknya akan memanggil dirinya sendiri sebanyak 2^{n-1} ditambah ketika fungsi sampai pada kasus basis $n = 0$ dimana fungsi langsung melakukan *return* tanpa melakukan perhitungan. Maka total kompleksitas waktu untuk menghitung semua titik aproksimasi menggunakan divide and conquer adalah $O(2^n * k^2)$.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Algoritma *divide and conquer* adalah salah satu pendekatan algoritma untuk menyelesaikan berbagai persoalan komputasi dengan cara membagi persoalan (*divide*) menjadi beberapa upa-persoalan hingga upa-persoalan tersebut dapat diselesaikan secara langsung (*solve*), lalu kemudian menggabungkan (*combine*) semua solusi dari upa-persoalan tersebut untuk menjawab persoalan awal. Pada kasus aproksimasi kurva bezier ini, algoritma divide and conquer dilakukan dengan mencari titik tengah dari k buah titik kontrol sebanyak $k(k\text{-}1)/2$ kali hingga didapatkan satu titik akhir hasil aproksimasi P, lalu membagi sisa titik aproksimasi menjadi dua wilayah, yaitu titik aproksimasi yang ada di kiri titik P dan di kanan titik P untuk kemudian diselesaikan secara rekursif. Berdasarkan uji coba, diperoleh bahwa pembentukan aproksimasi kurva dengan divide and conquer cenderung lebih cepat dari pembentukan secara *brute force* yang memanfaatkan rumus kurva bezier. Namun, bila dilihat dari kompleksitas waktunya ternyata kompleksitas kedua algoritma tersebut tidak jauh berbeda, hal ini mungkin dapat menjelaskan alasan pada beberapa data uji algoritma *brute force* lebih cepat dari *divide and conquer*.

5.2. Saran

Terdapat beberapa saran untuk pengembangan kedepannya yang dapat membuat program menjadi lebih baik, seperti menambahkan fitur GUI untuk melakukan input data, visualisasi untuk titik pada ruang 3 dimensi, dan lain-lain.

DAFTAR REFERENSI

Algoritma Divide and Conquer. (n.d.). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian1.pdf

LAMPIRAN

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.	✓	

Repositori github dapat diakses melalui pranala berikut:

https://github.com/IrfanSidiq/Tucil2_13522007