Laporan
Tugas Kecil 2 IF2211 Strategi Algoritma
Semester II tahun 2023/2024

Membangun Kurva Bézier dengan Algoritma Titik Tengah berbasis Divide and Conquer

Dosen Pengampu:
Ir. Rila Mandala, M.Eng., Ph.D.
Monterico Adrian, S.T., M.T.

Disusun Oleh:
Ahmad Rafi Maliki (13522137)

Teknik Informatika
Sekolah Teknik Informatika dan Elektro
Institut Teknologi Bandung

Daftar Isi

Bab I : Landasan Teori

1.1. Algoritma *Divide and Conquer*

Algoritma *Divide and Conquer* memiliki arti yang sesuai dengan namanya yaitu membagi dan menaklukan. Pada algoritma ini, sebuah permasalahan/*problem* dibagi-bagi menjadi *sub-problem* yang lebih kecil sehingga tiap *sub-problem* tersebut lebih mudah untuk diselesaikan. Setelah membagi permasalahan menjadi lebih kecil dan menyelesaikannya, hasil dari tiap penyelesaian *sub-problem* tersebut disatukan kembali untuk memperoleh hasil utuh dari permasalahan awal.

Banyak sekali algoritma berbasis *divide and conquer* yang sering digunakan dalam dunia pemrograman, antara lain *quick sort, merge sort,* dan*, binary searc*h yang biasanya diimplementasikan secara rekursif untuk membagi permasalahan menjadi masalah yang lebih kecil sehingga lebih mudah untuk diselesaikan.

1.2. Algoritma *Brute Force*

Sesuai dengan namanya, algoritma ini mengandalkan *kekuatan* untuk menyelesaikan suatu permasalahan. Hampir seluruh permasalahan di lingkup komputasional mampu diselesaikan dengan algoritma ini. Namun, menggunakan algoritma ini tidak datang dengan harga yang murah. Algoritma ini cenderung membutuhkan waktu yang lama dan *computational power* yang tinggi karena algoritma ini akan mencari seluruh kemungkinan solusi dari permasalahan dan menentukan solusi mana yang paling optimal.

Walaupun algoritma ini cukup mahal, banyak sekali *problem* yang satu-satunya cara untuk menyelesaikannya adalah menggunakan algoritma ini, seperti *Traveling Salesman Problem* atau masalah sesederhana mencari suatu elemen pada *array* yang acak.

1.3. Kurva Bezier

Kurva bezier adalah sebuah kurva parametric yang digunakan untuk menggambar garis lengkung yang mulus pada suatu bidang. Titik pada kurva bezier dihasilkan melalui hasil kalkulasi algoritma titik tengah dari titik-titik kontrolnya. Sebuah kurva bezier dikatakan berderajat n apabila memiliki n+1 titik kontrol. Aplikasi kurva bezier dalam dunia komputasi antara lain pada *CAD Software, 3D modeling,* dan *typefaces.*

Bab II : Implementasi Algoritma

2.1. Pemodelan Program

Sebelum memahami implementasi algoritma *divide and conquer* dan algoritma *brute force* pada program penyusunan kurva bezier, penting untuk memahami dulu pemodelan dari variabel penting yang digunakan pada program.

2.1.1. *Class* Point

```python
class Point:

    # Constructor
    def __init__(self, x: float, y: float):
        self.x = float(x)
        self.y = float(y)
```

Gambar 2.1.1.1
class Point

Pada implementasi program ini, objek titik pada grafik dimodelkan dalam kelas Point yang memiliki nilai x dan nilai y bertipe data float.

2.1.2. Variabel *Global*

```python
# User input handler
num_control = input_num_control()
points = input_points(num_control)
iterations = input_iterations()

# Initializations
builder = points.copy() # A list of points on the Bézier curve with its relative controlers
control_points = points.copy() # A list of Bézier curve's controler points
extreme_point = Point.get_extreme_point(points)

# Initialize nth iteration points with 0th iteration points
nth_iteration_points_dnc = [[control_points[0], control_points[num_control-1]]]
nth_iteration_points_bf = [[control_points[0], control_points[num_control-1]]]
```

Gambar 2.1.2.1
global variables

Program ini memiliki tiga variabel *input* yang dapat diberikan oleh user, variabel <num_control> menyatakan banyaknya titik kontrol yang diberikan user. Sebuah kurva bezier memiliki derajat yang bernilai satu kurangnya dari banyak titik kontrol. Variabel <points> digunakan untuk menyimpan nilai dari titik kontrol namun seiring berjalannya program nilai variabel akan berubah sesuai keperluan. Terakhir variabel <iteration> menyatakan banyaknya iterasi yang diinginkan.

Selain dari ketiga variabel *input* terdapat lima variabel lainnya seperti yang didefinisikan pada gambar 2.1.2.1 yang nilainya akan berubah secara dinamis seiring berjalannya perhitungan titik kurva bezier.

2.2. Implementasi Algoritma *Divide and Conquer*

2.2.1. Metode Rekursif

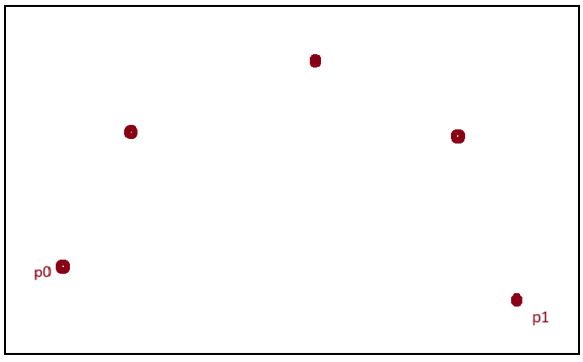
Dalam pembentukan kurva bezier menggunakan algoritma *divide and conquer* akan dilakukan pencarian titik tengah yang posisinya dipengaruhi oleh semua titik kontrol.

Ide dari algoritma *divide and conquer* secara rekursif adalah membagi list kumpulan titik kontrol yang dimiliki secara rekursif sampai terbentuk sub list/sub kurva lebih kecil yang berukuran sebesar jumlah titik kontrol mula-mula. Pada iterasi nol dari rekursi ini, list titik kontrol yang dimiliki sudah berukuran terkecil sehingga tidak perlu dibagi lagi.

Setelah list dibagi menjadi beberapa sublist, barulah algoritma pencarian akan mencari titik tengah dari tiap sublist yang posisinya dipengaruhi oleh tiap-tiap titik pada sublist tersebut.

Setelah semua titik tengah dari tiap sublist ditemukan, kumpulan dari titik tengah tersebut akan dihubungkan diantara titik kontrol awal dan akhir membentuk aproksimasi kurva bezier, semakin banyak iterasi yang dilakukan akan semakin banyak titik aproksimasi yang dibentuk mengakibatkan aproksimasi kurva akan semakin akurat.

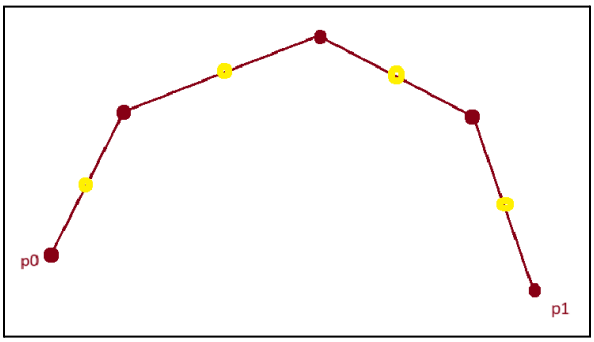
Untuk memperjelas cara pencarian titik tengah berdasarkan penjelasan di atas, silakan amati beberapa ilustrasi berikut.



Gambar 2.2.1.1

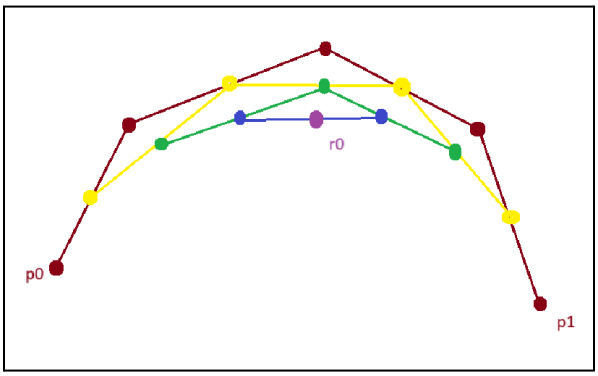
Gambar di atas, merupakan kondisi mula-mula suatu kurva bezier yang memiliki lima titik kontrol (berderajat empat) dimana p0 merupakan titik awal dan p1 merupakan titik akhir, Menurut algoritma, kurva yang terbentuk oleh titik kontrol mula-mula sudah merupakan sub kurva terkecil sehingga list titik tidak

perlu dipecah lagi. Algoritma kemudian akan menemukan titik tengah dari setiap pasangan titik yang bersebelahan sehingga terbentuk sub kurva yang jumlah titiknya berkurang satu.



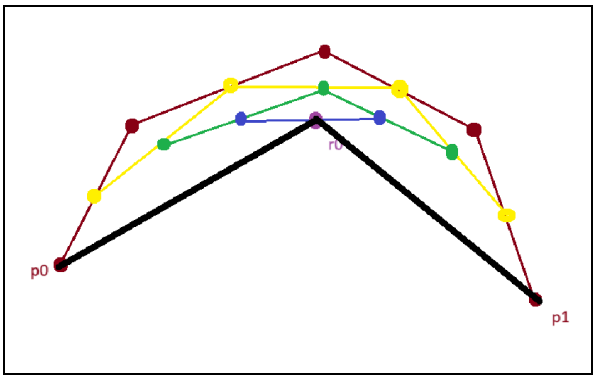
Gambar 2.2.1.2

Setelah titik baru (titik kuning pada gambar) yang merupakan titik tengah dari setiap pasangan titik merah yang bersebelahan ditemukan, hal yang sama akan dilakukan pada list titik kuning dan list setelahnya secara rekursi sampai list baru yang dihasilkan hanya memiliki panjang satu titik.



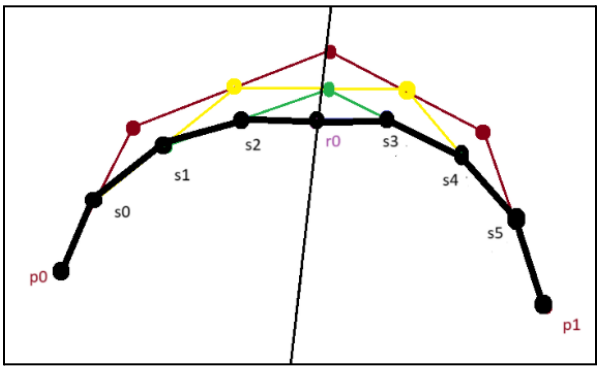
Gambar 2.2.1.3

Setelah ditemukan titik tengah terakhir yang hanya sendiri yaitu titik ungu (r0), algoritma akan selesai melakukan rekursi dan mengembalikan nilai list [p0, r0, p1]. Pasangan titik ini lah merupakan hasil dari iterasi pertama algoritma ini yang mana titik tersebut merupakan titik pembentuk kurva bezier.



Gambar 2.2.1.4

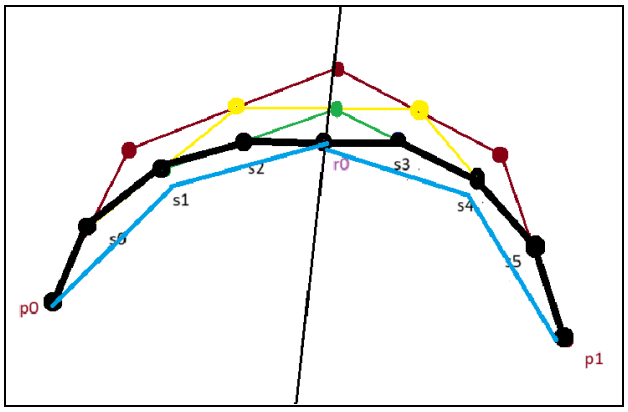
Garis hitam yang terbentuk dengan menghubungkan titik [p0, r0, p1] merupakan garis yang membentuk kurva bezier yang dihasilkan.



Gambar 2.2.1.5

Dapat diamati bawah setelah iterasi selesai, akan dihasilkan dua sub kurva yang berukuran sama dengan sub kurva mula-mula. Sub kurva tersebut adalah [p0, s0, s1, s2, r0] kita sebut <sub_left> dan [r0, s3, s4, s5, p1] kita sebut <sub_right>, pembagian menjadi left dan right adalah pembagian kurva yang memiliki sembilan titik menjadi sub kurva dengan lima titik (yaitu panjang titik kontrol awal) seperti yang algoritma *divide and conquer* ini lakukan,

Untuk iterasi berikutnya, akan dilakukan perhitungan titik tengah dari masing-masing sub kurva baru yang terbentuk pada iterasi sebelumnya. Contohnya pada <sub_left>, di iterasi sebelumnya dihasilkan titik yang menghubungkan p0 dengan r0 yang merupakan titik ujung dari kurva <sub_left>. Di iterasi berikutnya akan dilakukan algoritma pencarian titik tengah lagi pada kurva <sub_left> sehingga terdapat titik tengah yang menghubungkan p0 dan r0.



Gambar 2.2.1.6

Pada iterasi berikutnya kurang lebih akan terbentuk kurva seperti kurva warna biru pada gambar sehingga kurva yang awalnya terbentuk dari dua garis, sekarang terbentuk dari empat garis, jumlah garis terus akan bertambah jadi dua kali setiap iterasinya.

```
for i in range(1, iterations+1):

    ''' Bezier calculations with recursion, prone to RecursionError '''

    try:
        builder = calc_bezier_dnc(builder, num_control)
        points = get_points(builder, num_control)
        nth_iteration_points_dnc.append(points)

        succeses_iterations += 1

    except RecursionError:
        print(f"\nTerjadi RecursionError pada iterasi ke-{i}. Terlalu banyak kontroler point ({len(builder)} elemen).")
        break
```

Gambar 2.2.1.7

Berikut merupakan potongan *source code* implementasi algoritma menggunakan bahasa Python. Potongan kode tersebut merupakan kode yang terletak di program main yang bertugas untuk melakukan looping perhitungan sebanyak iterasi yang diinginkan, hasil iterasi akan disimpan di list *nth_iteration_points_dnc*.

Variabel builder akan menyimpan seluruh titik pada sub kurva (pada gambar 2.2.1.5 adalah seluruh titik hitam [p0, s0, s1, s2, r0, s3, s4, s5, p1]) hasil perhitungan fungsi *calc_bezier_dnc()*, lalu fungsi *get_points()* akan mengolah variabel builder sehingga dihasilkan hanya titik yang terletak pada kurva akhir (pada gambar 2.2.1.4 adalah titik [p0, r0, p1]). Iterasi berikutnya akan menggunakan data dari variabel *builder* iterasi sebelumnya sehingga menghasilkan variabel *points* yang baru.

```python
def calc_bezier_dnc(points: List[Point], num_control: int) -> List[Point]:

    mid_points = []
    result = []

    if len(points) > num_control:

        left = points[0:num_control]
        right = points[num_control-1:len(points)]

        result = calc_bezier_dnc(left, num_control)[:-1] + calc_bezier_dnc(right, num_control)

    else:

        if len(points) == 1:
            result = points

        else:
            for i in range(num_control-1):

                mid = points[i].midpoint(points[i+1])
                mid_points.append(mid)

            inner = calc_bezier_dnc(mid_points, num_control-1)
            result = [points[0]] + inner + [points[num_control-1]]

    return result
```

Gambar 2.2.1.8

Berikut merupakan *source code* dari fungsi *calc_bezier_dnc()*. Fungsi ini berperan untuk mencari titik-titik tengah dari tiap sub kurva.

Bisa diperhatikan bahwa pada blok program {if len(points) > num_control:} akan mengecek apakah kurva lebih besar dari jumlah titik kontrol atau tidak, dan jika iya maka akan dibagi menjadi sub kurva kiri dan sub kurva kanan, kemudian akan diselesaikan masalah masing-masing dan hasilnya akan disatukan kembali sesuai prinsip *divide and conquer*.

Jika sub kurva sudah sama dengan atau lebih kecil dari jumlah titik kontrol, blok program {else:} akan terpanggil. Pada blok program ini dilakukan rekursi pengecilan sub kurva secara rekursi juga sampai ditemukan titik tengah seperti yang diilustrasikan pada gambar 2.2.1.3, dimana kurva merah yang memiliki lima titik, kemudian dibentuk kurva kuning dengan empat titik, sampai dihasilkan satu titik akhir yaitu titik ungu. Setelah ditemukan titik terakhir, blok program {if len(points) == 1:} yang merupakan basis rekursi akan terpanggil, sehingga program akan *return* dan mengembalikan seluruh titik yang terbentuk (contoh pada gambar 2.2.1.5 yaitu titik [p0, s0, s1, s2, r0, s3, s4, s5, p1]).

```python
def get_points(points: List[Point], num_control: int) -> List[Point]:

    temp = []
    for i in range(len(points)):
        if i % (num_control-1) == 0:
            temp.append(points[i])

    return temp
```

Gambar 2.2.2.9

Berikut merupakan *source code* untuk fungsi *get_points()* yang mana akan hanya mengembalikan titik-titik yang terletak pada indeks yang habis dibagi oleh jumlah titik kontrol dikurang satu. Kerja fungsi ini diamati pada perbedaan dari gambar 2.2.1.5 ke 2.2.1.4.

2.2.2. Metode Sekuensial

Perbedaaan metode sekuensial dengan rekurens hanya terletak di tahap pembagian kurva menjadi sub kurva kecil berukuran jumlah titik kontrol awal. Jika pada metode rekurens pembagian kurva dilakukan dengan cara rekursif pada fungsi *calc_bezier_dnc(),* pada metode ini pembagian kurva dilakukan secara sekuensial pada main program sehingga list point yang diterima oleh *calc_bezier_dnc()* sudah berukuran sejumlah titik kontrol awal dan tak perlu dibagi lagi menjadi *left* dan *right.*

Adaptasi sekuensial ini diperlukan karena adanya batasan bahasa pemrograman dimana atas batas rekursi maksimal yang bisa ditangani oleh bahasa pemrograman tertentu. Dengan cara sekuensial akan lebih sulit untuk melewati batas tersebut karena rekursi hanya dilakukan pada sub kurva yang sudah berukuran paling kecil, ketimbang satu rekursi besar yang melibatkan semua titik.

```python
for i in range(1, iterations+1):

    ''' Bezier calculations with iteration, able to handle larger iterations '''

    temp_builder = [builder[0]]

    for j in range(0, len(builder)-1, num_control-1):

        partial_builder = calc_bezier_dnc(builder[j:j+num_control], num_control)
        temp_builder.extend(partial_builder[1:])

    builder = temp_builder
    points = get_points(builder, num_control)
    nth_iteration_points_dnc.append(points)

    succeses_iterations += 1
```

Gambar 2.2.2.1

Berikut merupakan *source code* yang terletak pada program main yang memiliki peran sama seperti kode pada gambar 2.2.1.1 yaitu untuk melakukan looping perhitungan tiap iterasi. Namun, pada kode ini terdapat *nested loop* yang bertugas untuk melakukan pemanggilan fungsi secara sekuensial dengan *list slicing* untuk menghindari batas rekursi maksimal. Hasil dari pemanggilan dengan tiap *list slicing* kemudian di konkatenasi menjadi satu sesuai prinsip *divide and conquer.*

Metode sekuensial ini adalah metode yang akhirnya diterapkan pada program Tugas Kecil ini.

2.3. Implementasi Algoritma *Brute Force*

Pada metode *brute force* akan dilakukan interpolasi dengan bantuan polinomial Bernstein terhadap titik-titik kontrol sehingga terbentuk persamaan kurva bezier yang diinginkan.

$$C(t) = \sum_{i=0}^{n} b_{i,n}(t) P_i \, , t \in [0, 1]$$

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

$$\binom{n}{i} = \frac{n!}{(n-i)!i!}$$

Berikut merupakan formula yang digunakan untuk perhitungan titik pada kurva bezier. Nilai pada kurva bezier <*C(t)*> diperoleh dari hasil sumasi perkalian nilai polinomial Bernstein <$b_{i,n}$ *(t)*> dengan titik kontrol <P_i> untuk setiap titik kontrol i.

```
for i in range(1, iterations+1):
    nth_iteration_points_bf.append(calc_bezier_bf(control_points, num_control, i))
```

Gambar 2.3.1

Berikut merupakan *source code* implementasi algoritma *brute force* yang terletak pada program main. Terdapat *loop* yang bertugas untuk memanggil fungsi *calc_bezier_bf()* sebanyak iterasi yang diperlukan.

Namun, pada metode *brute force* tidak ada konsep iterasi karena dengan menggunakan fungsi kita bisa langsung mendapatkan nilai <x, y> yang merupakan titik pada kurva bezier, perlu penyesuain sehingga kita bisa memperoleh kurva bezier yang menyerupai hasil tiap iterasi metode *divide and conquer* dengan cara memanipulasi *increment* nilai t dari 0 sampai 1.

```python
def calc_bezier_bf(points: List[Point], num_control: int, iter: int) -> List[Point]:

    result = []
    num_points = count_points(iter)

    for t in range(num_points):

        x = 0
        y = 0

        for i in range(num_control):
            x += points[i].x * bernstein_polynomial(i, num_control-1, t/(num_points-1))
            y += points[i].y * bernstein_polynomial(i, num_control-1, t/(num_points-1))

        result.append(Point(x, y))

    return result
```

Gambar 2.3.2

Berikut merupakan *source code* dari fungsi *calc_bezier_bf()*. Fungsi ini akan melakukan proses sumasi hasil perkalian titik kontrol dengan polinomial Bernstein untuk seluruh titik kontrol. Nilai t yang diperlukan pada polinomial Bernstein diperoleh dengan cara membagi t yang merupakan variabel iterator dari nol sampai banyak titik yang diinginkan dengan banyak titik yang diinginkan dikurangi satu.

```python
def bernstein_polynomial(i: int, n: int, t: float) -> float:

    return math.comb(n, i) * (t**i) * ((1-t)**(n-i))
```

Gambar 2.3.3

Berikut merupakan *source code* dari fungsi *bernstein_polynomial()*. Fungsi ini akan mengembalikan nilai dari koefisien polinomial Bernstein.

```python
def count_points(iter: int) -> int:

    if iter == 0:
        return 2
    else:
        return 2*count_points(iter-1) - 1
```

Gambar 2.3.3

Berikut merupakan *source code* dari fungsi *count_points()*. Fungsi ini merupakan fungsi rekursif yang akan mengembalikan banyak titik pada kurva bezier yang diperoleh dari metode *divide and conquer* pada iterasi *iter*.

2.4. Implementasi Bonus

2.4.1. Kurva Bezier Derajat N

Mengacu pada ilustrasi gambar 2.2.1.3 dan *source code* gambar 2.2.1.8, melakukan kalkulasi titik-titik kurva bezier berderajat N dapat dilakukan. Pada ilustrasi disajikan kurva berderajat empat dengan lima titik kontrol, pada blok program {else:}, program akan secara rekursif melakukan kalkulasi mencari titik tengah untuk sub-kurva yang berderajat lebih kecil sampai ditemukan satu titik akhir. Kurva merah yang berderajat empat, berubah menjadi kuning yang berderajat tiga, lalu jadi hijau yang berderajat dua, lalu biru berderajat satu hingga ungu berderajat nol. Hasil dari peng-kalkulasian secara rekursif ini adalah titik-titik paling dalam yaitu titik-titik hitam pada gambar 2.2.1.5.

Setelah diperoleh titik-titik hitam yang saya sebut <builder>, fungsi *get_points()* pada gambar 2.2.2.9 akan meng-ekstrak titik-titik yang akan menjadi pembentuk kurva bezier yaitu yang berada di indeks yang nilai nya kongruen dengan nol dalam modulo derajat kurva.

2.4.2. Visualisasi Pembentukan Kurva

Mengacu pada *source code* gambar 2.2.1.7 dan 2.3.1, hasil kalkulasi dari setiap iterasi disimpan pada variabel *nth_iteration_points_dnc* untuk hasil dari algoritma *divide and conquer* dan *nth_iteration_points_bf* untuk hasil dari algoritma *brute force.*

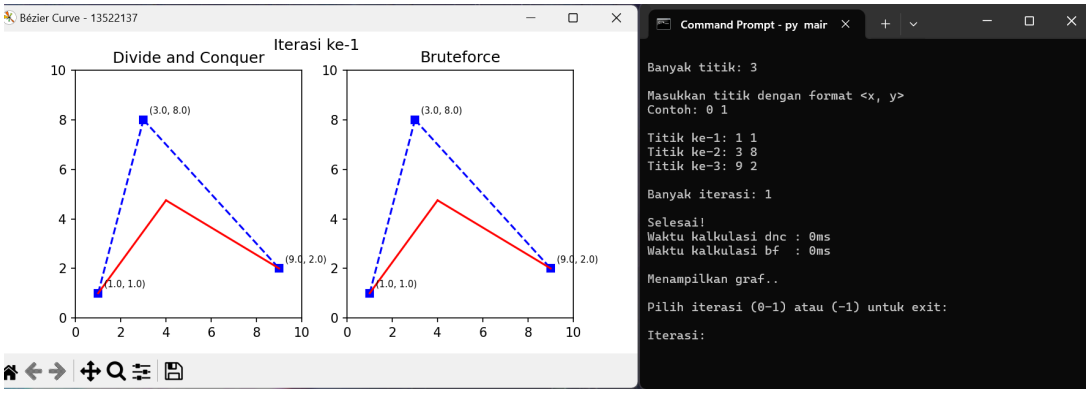
```
# Render result with delay for each iterations
animate_points(fig, ax1, ax2, extreme_point, nth_iteration_points_dnc, nth_iteration_points_bf, control_points, delay)

print(f"\nPilih iterasi (0-{succeses_iterations}) atau (-1) untuk exit: \n")
while True:

    show = input_show_iterations(succeses_iterations)

    if (show == -1):
        print("\nEXIT!\n")
        sys.exit()
    else:
        draw_graph(fig, ax1, ax2, extreme_point, nth_iteration_points_dnc[show], nth_iteration_points_bf[show], control_points, show)
```

Gambar 2.4.2

Kemudian pada program main terdapat pemanggilan fungsi *animate_points()* yang mana akan menampilkan kurva yang terbentuk dari kedua algoritma mulai dari iterasi nol sampai iterasi ke-N dengan *delay* di antara setiap iterasinya. Fitur tambahan lainnya terdapat pada blok program {while True:} dimana *user* dapat memilih iterasi mana yang ingin ditampilkan.

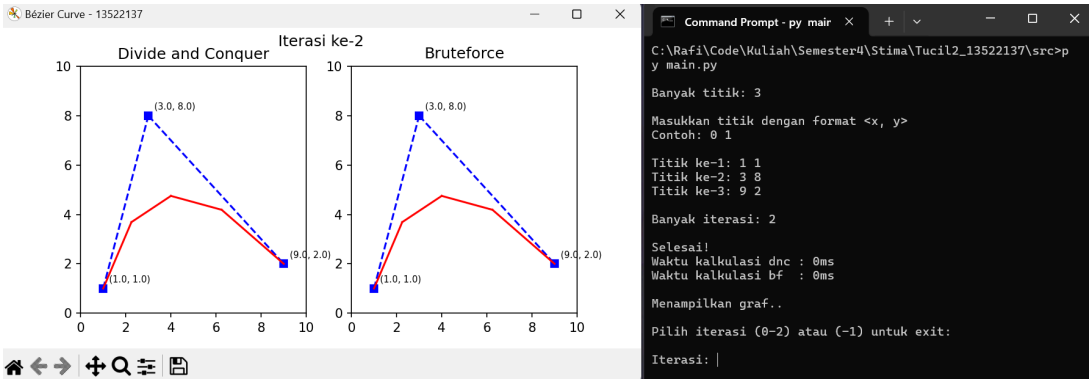
Bab III : Pengujian

3.1. Test Case 1



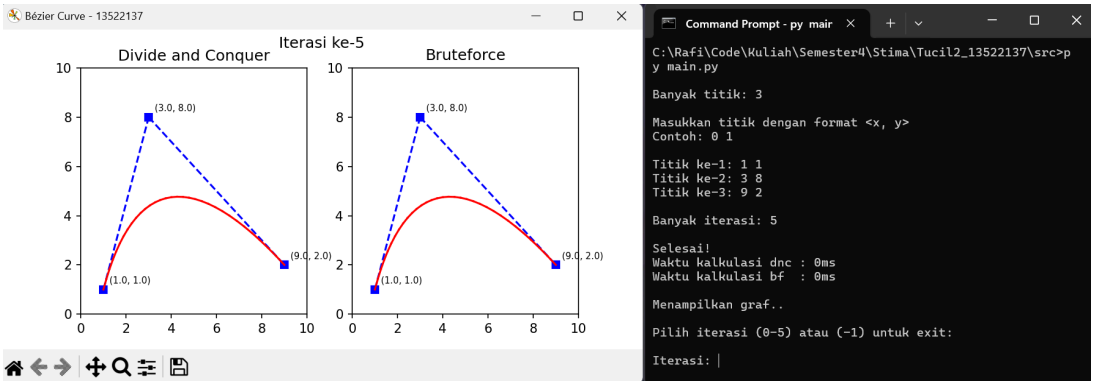
Gambar 3.1.1

3.2. Test Case 2



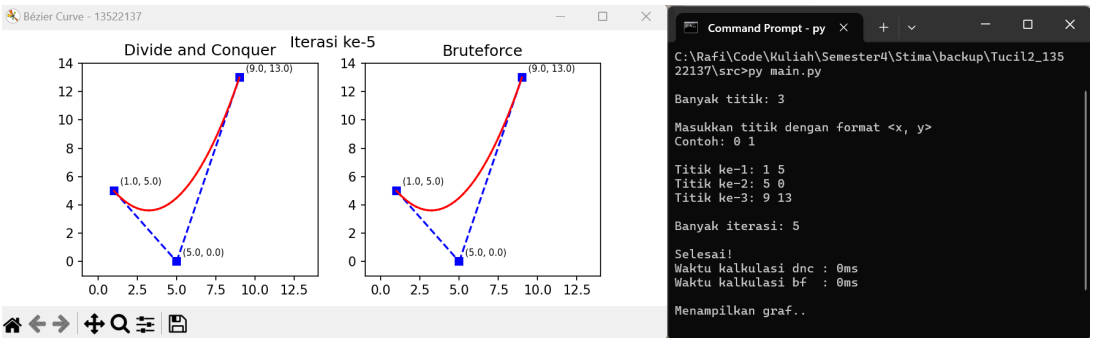
Gambar 3.2.1

3.3. Test Case 3



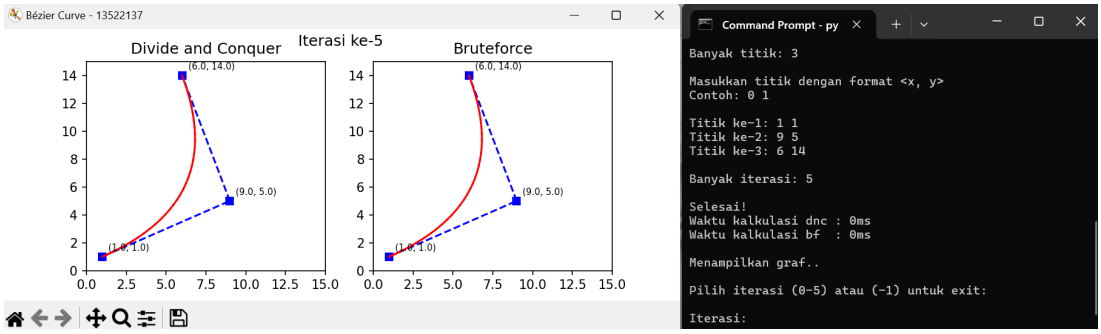
Gambar 3.3.1

3.4. Test Case 4



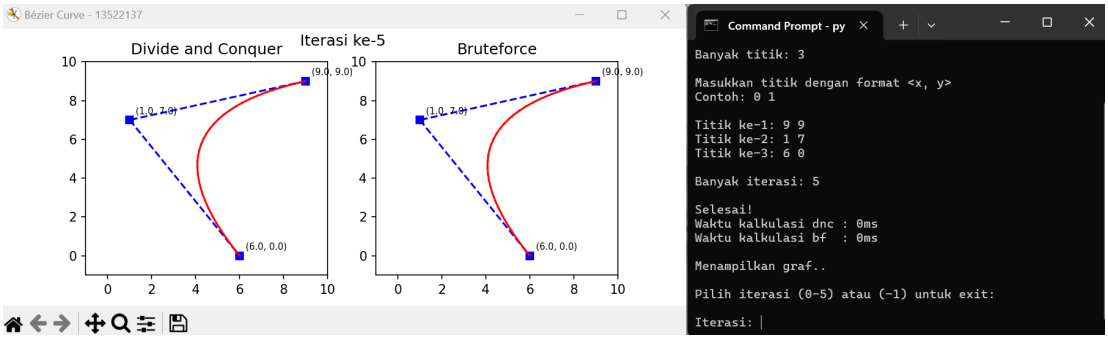
Gambar 3.4.1

3.5. Test Case 5



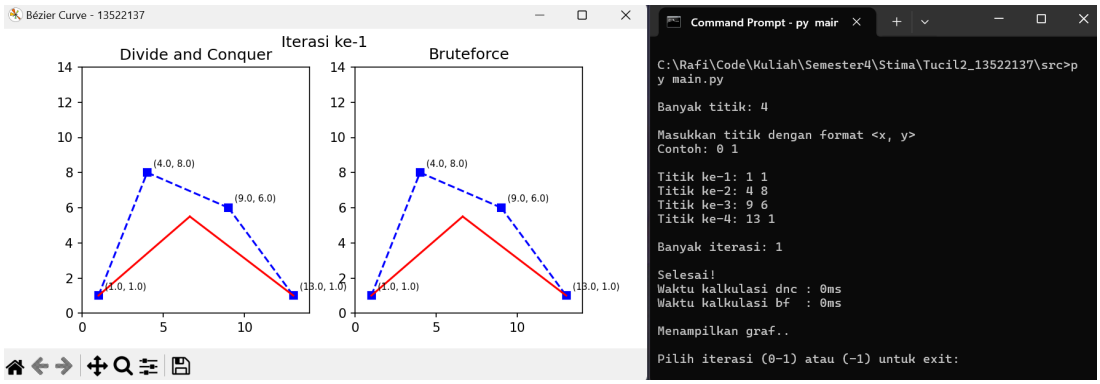
Gambar 3.5.1

3.6. Test Case 6



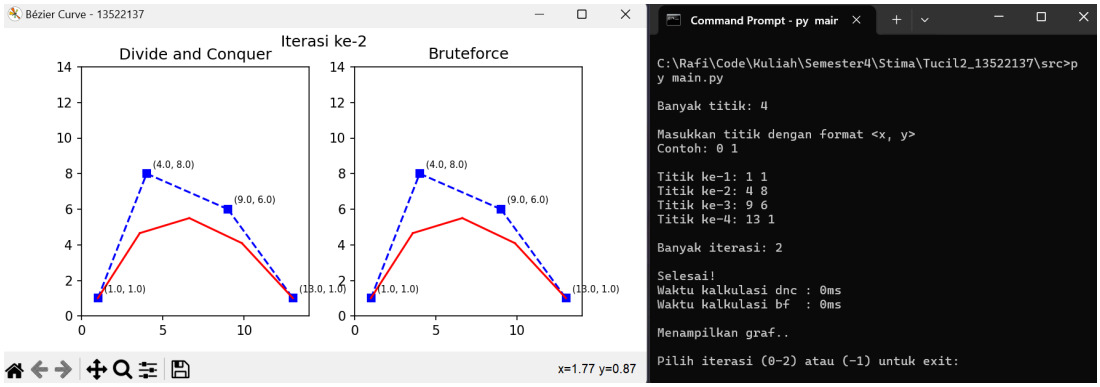
Gambar 3.6.1

3.7. Test Case 7



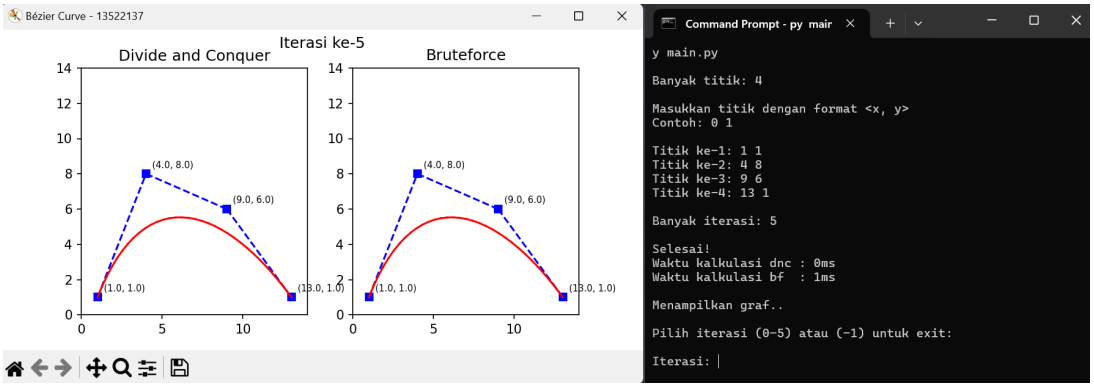
Gambar 3.7.1

3.8. Test Case 8



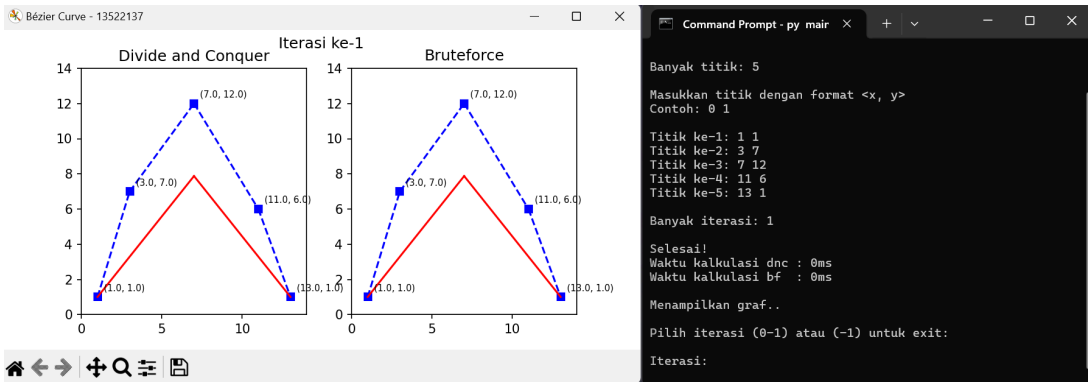
Gambar 3.8.1

3.9. Test Case 9



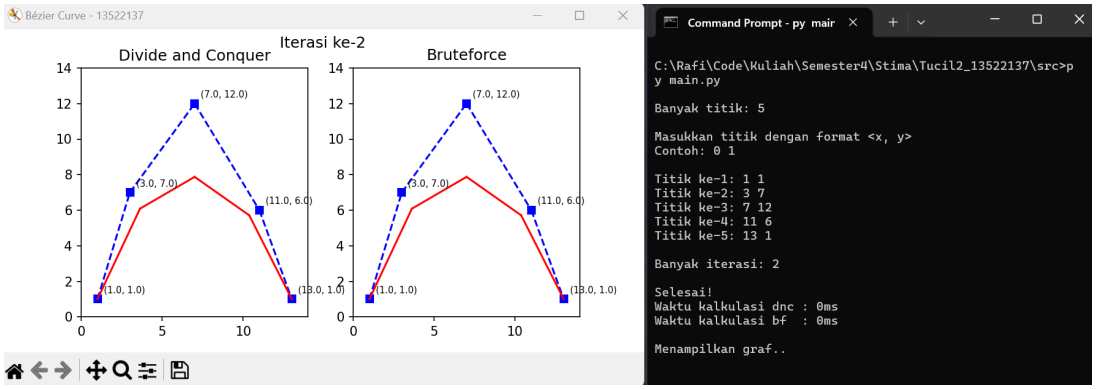
Gambar 3.9.1

3.10. Test Case 10



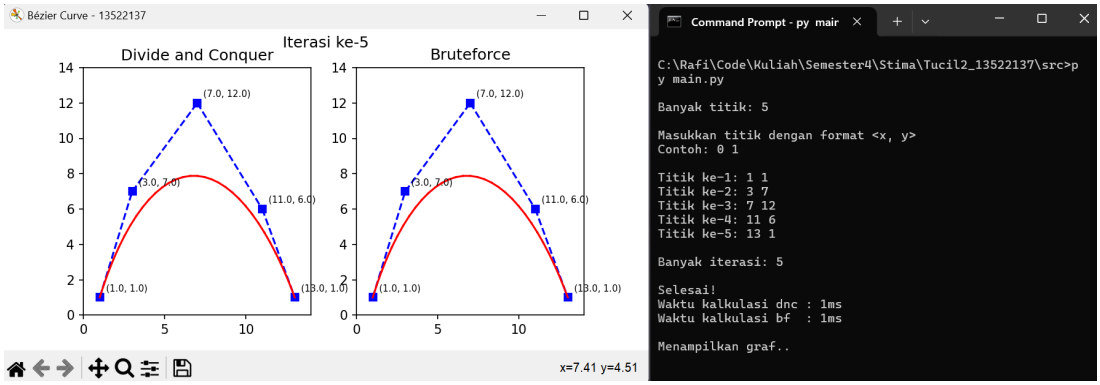
Gambar 3.10.1

3.11. Test Case 11



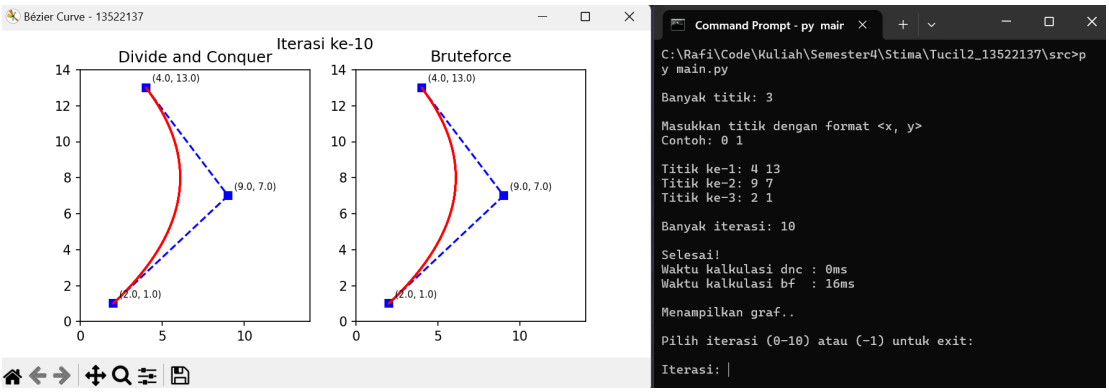
Gambar 3.11.1

3.12. Test Case 12



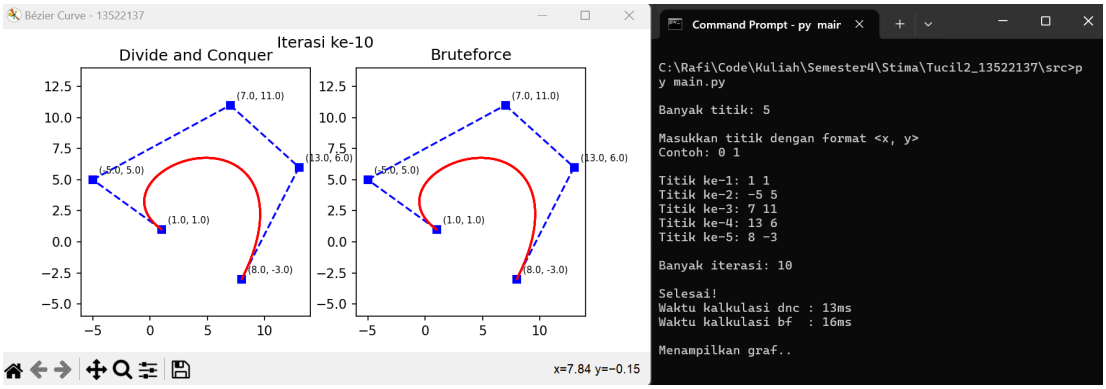
Gambar 3.12.1

3.13. Test Case 13



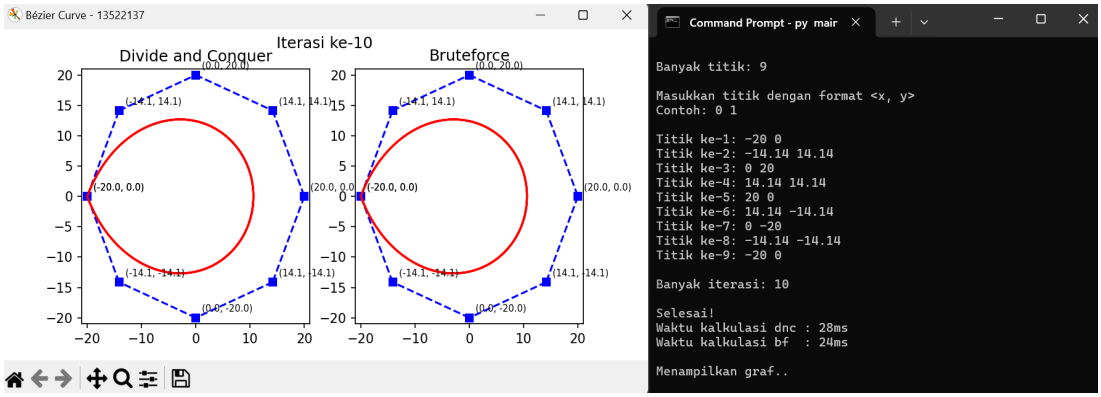
Gambar 3.13.1

3.14. Test Case 14



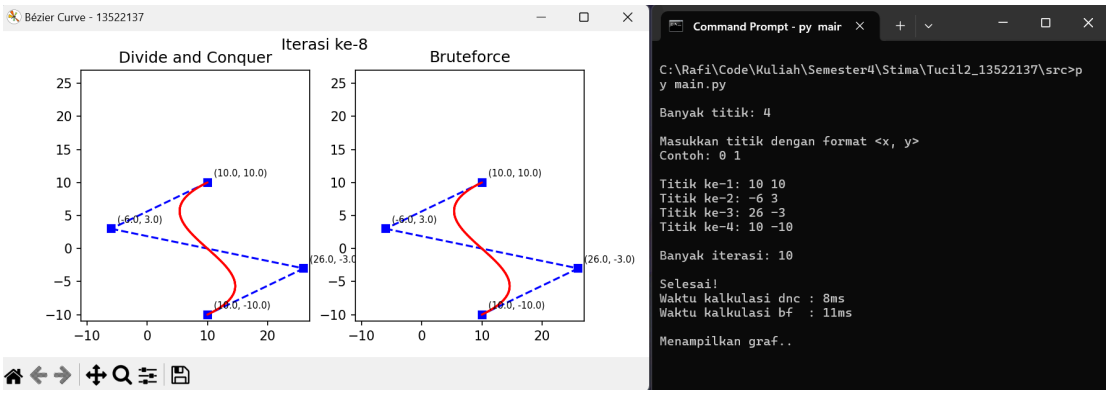
Gambar 3.14.1

3.15. Test Case 15



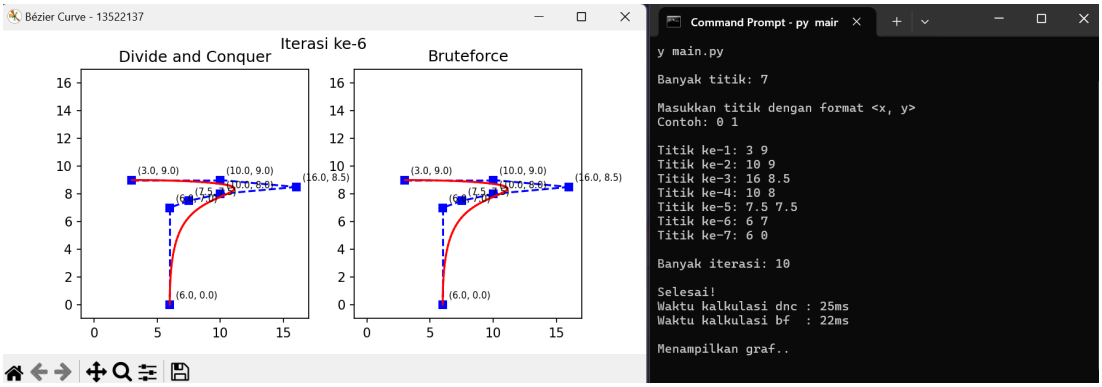
Gambar 3.15.1

3.16. Test Case 16



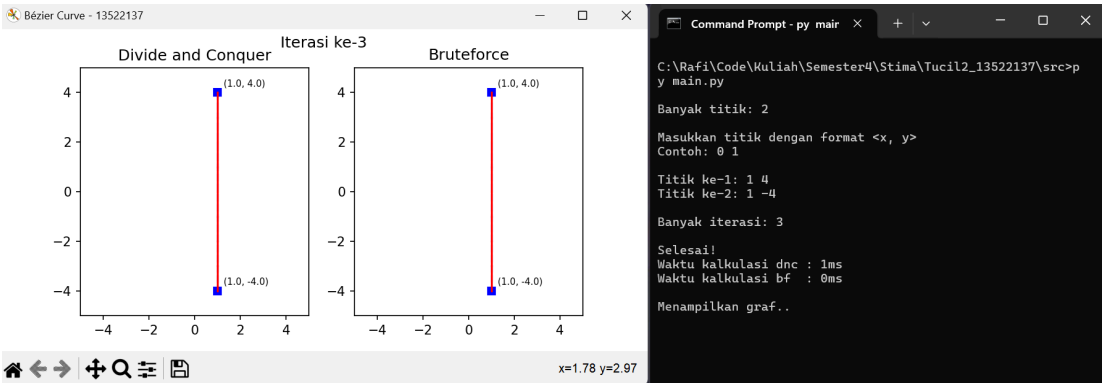
Gambar 3.16.1

3.17. Test Case 17



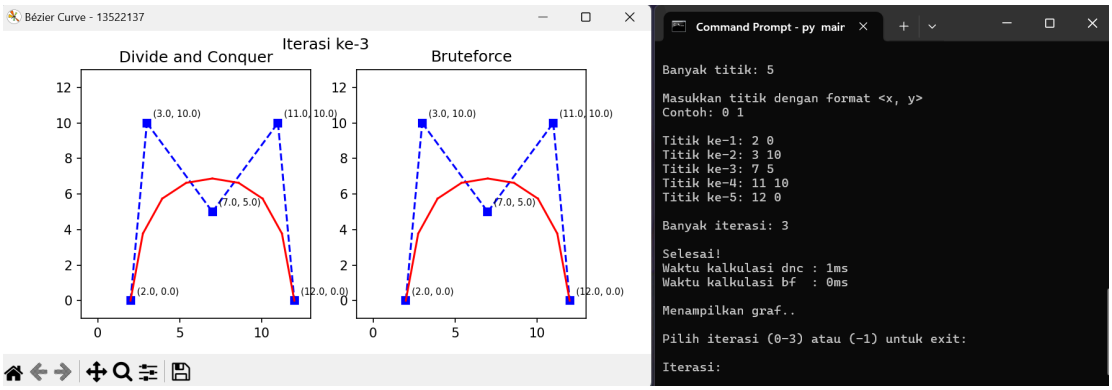
Gambar 3.17.1

3.18. Test Case 18



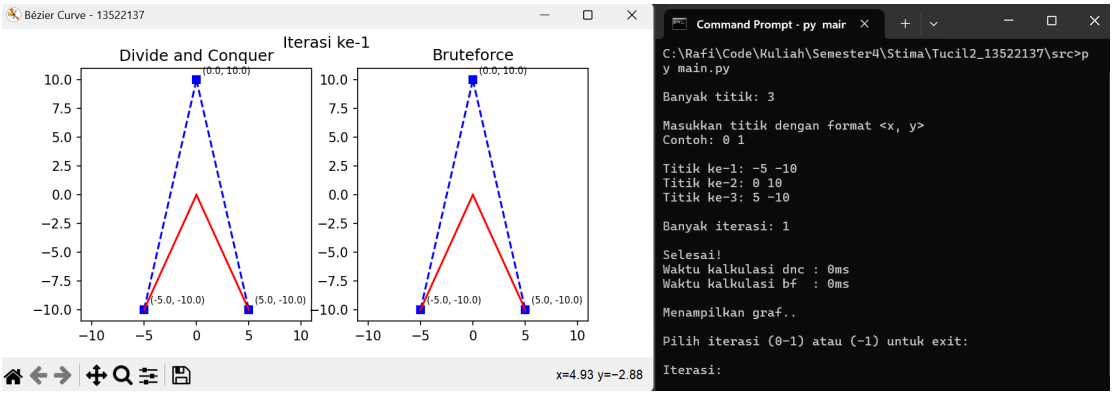
Gambar 3.18.1

3.19. Test Case 19



Gambar 3.19.1

3.20. Test Case 20



Gambar 3.20.1

Bab IV : Analisis Hasil Pengujian

4.1. Analisis perbandingan kurva kedua algoritma

Berdasarkan seluruh hasil uji coba yang disajikan mulai dari gambar 3.1.1 hingga gambar 3.17.1 algoritma *Divide and Conquer* sukses menghasilkan grafik yang serupa dengan kurva yang dihasilkan menggunakan algoritma *Brute Force* di setiap iterasinya untuk kurva berderajat N.

4.2. Analisis waktu eksekusi

Mengacu pada hasil uji coba dari gambar 3.1.1 hingga 3.912.1 tidak tampak perbedaan di waktu eksekusi untuk kurva berderajat kecil ($N \leq 4$) dan iterasi rendah ($i \leq 5$), yaitu keduanya membutuhkan waktu eksekusi 0ms atau 1ms.

Namun, uji coba pada gambar 3.13.1 dan 3.14.1 dimana kurva memiliki derajat kecil namun dilakukan iterasi sampai iterasi ke-10, algoritma *divide and conquer* memiliki waktu eksekusi yang lebih rendah ketimbang algoritma *brute force*. Hal ini menunjukan bahwa jika untuk iterasi tinggi, algoritma *divide and conquer* memiliki keunggulan.

Hal berbeda pun terjadi pada uji coba gambar 3.15.1 dan 3.17.1 dimana diuji coba untuk kurva dengan derajat tinggi. Pada kurva derajat tinggi di iterasi yang sama, kalkulasi menggunakan algoritma *brute force* membutuhkan waktu eksekusi yang lebih rendah ketimbang menggunakan *divide and conquer.* Hal ini menunjukan bahwa metode *brute force* lebih cocok untuk kurva yang berderajat tinggi.

4.3. Kompleksitas algoritma *Divide and Conquer*

Berdasarkan analisis waktu eksekusi pada bagian 4.3, algoritma *divide and conquer* memiliki keunggulan untuk kalkulasi kurva bezier derajat rendah namun iterasi yang tinggi.

Pada suatu kurva bezier, jumlah titik yang dihasilkan tidak dipengaruhi oleh derajat namun hanya dipengaruhi oleh jumlah iterasi, pada iterasi pertama (I_1) akan dihasilkan tiga titik kemudian pada iterasi ke i (I_n) akan dihasilkan titik sebanyak $2*I_{n-1}-1$ secara rekursif atau jumlah titik $I(n) = 2^n + 1$ jika mengubah persamaan menjadi non rekursif.

Algoritma *divide and conquer* pada iterasi I_n akan melakukan rekursi sebanyak (<jumlah titik pada iterasi tersebut> div <derajat kurva>) untuk membagi sub kurva menjadi seukuran derajat kurva tersebut di iterasi ke nol. Kemudian di tiap sub kurva tersebut akan dilakukan rekursi lagi sebanyak derajat kurva (N) kali.

Dengan memperhitungkan jumlah rekursi kedua angka di atas, pada iterasi ke (n), kurva berderajat (N) memiliki kompleksitas sebagai berikut:

Jumlah rekursi untuk membagi kurva menjadi sub kurva seukuran banyak titik kontrol awal (N) pada iterasi ke-n = $\frac{2^n + 1}{N}$

Jumlah rekursi di tiap sub kurva (konstan untuk iterasi berapapun) = $\sum_{i=1}^{N} i = \frac{N + N^2}{2}$

Jumlah rekursi total pada iterasi ke-n = $\frac{2^n + 1}{N} x \frac{N + N^2}{2} = 2^{n-1} + 2^{n-1}N + \frac{1}{2} + \frac{N}{2}$

Jumlah rekursi dari iterasi ke-1 sampai ke-n = $\sum_{i=1}^{n} i = 2^{i-1} + 2^{i-1}N + \frac{1}{2} + \frac{N}{2}$

diperoleh, $\sum_{i=1}^{n} i = 2^i N = (2^{n+1} - 2)\ N$

Maka algoritma ini memiliki kompleksitas

O(2^{n}N)

Berdasarkan hasil kalkulasi di atas, algoritma *divide and conquer* yang digunakan pada program ini kompleksitasnya akan bertumbuh secara eksponensial basis dua menurut banyaknya iterasi dan akan bertumbuh secara linier menurut banyaknya titik kontrol.

4.4. Kompleksitas algoritma *Brute Force*

Kelebihan yang dimiliki oleh algoritma *brute force* adalah mampu untuk menghasilkan grafik yang setara terhadap grafik yang dihasilkan oleh *divide and conquer* pada iterasi ke-n tanpa perlu melakukan kalkulasi iterasi sebelumnya. Algoritma ini mampu langsung menghasilkan titik-titik kurva bezier pada iterasi ke berapapun karena titik-titiknya dihitung menggunakan sebuah persamaan hasil interpolasi titik-titik kontrol.

Dengan melakukan analisis *source code* algoritma ini pada gambar 2.3.1 dan 2.3.1, maka diperoleh perhitungan kompleksitas sebagai berikut

Banyak titik yang perlu dikalkulasi pada iterasi ke-n = $2^n + 1$

Banyak loop yang diperlukan untuk menghitung tiap titik = N (banyak titik kontrol)

diperoleh, banyak loop = $N(2^n + 1)$

Maka algoritma ini memiliki kompleksitas

$O(2^nN)$

Berdasarkan hasil kalkulasi di atas, algoritma *brute force* yang digunakan pada program ini kompleksitasnya akan bertumbuh secara eksponensial basis dua menurut banyaknya iterasi dan akan bertumbuh secara linier menurut banyaknya titik kontrol.

4.5. Analisis Perbandingan Kompleksitas

Berdasarkan perhitungan kompleksitas pada bagian 4.4 dan 4.4, kedua algoritma memiliki algoritma yang sama. Namun, hasil uji coba yang disajikan pada Bab 3 memberikan hasil waktu perhitungan yang berbeda.

Hal demikian terjadi karena kompleksitas algoritma *divide and conquer* merupakan banyaknya kalkulasi yang diperlukan dari iterasi ke-1 hingga ke-n. Namun, kompleksitas yang dimiliki oleh algoritma *brute force* hanya banyaknya kalkulasi yang diperlukan pada iterasi ke-n karena memang seperti itu cara kerja algoritma tersebut, bisa langsung mencapai hasil akhir.

Tetapi, pada implementasi program ini walaupun algoritma *brute force*, tetap dilakukan kalkulasi tiap iterasi dari iterasi ke-1 sampai ke-n untuk keperluan perbandingan. Sehingga, jika iterasi nya tinggi tentu saja akumulasi waktu eksekusi algoritma *divide and conquer* akan lebih rendah ketimbang algoritma *brute force*. Hal tersebut sesuai dengan pernyataan pada bagian 4.2 yang diperoleh dari mengamati hasil percobaan.

Kesimpulannya, jika kita hanya peduli terhadap hasil akhir yaitu kurva yang dibentuk pada iterasi ke-n, implementasi dari algoritma ini menggunakan algoritma *divide and conquer* ataupun *brute force* memiliki kompleksitas yang sama saja.

Daftar Pustaka

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf

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

Shiach, Dr. Jon. "Bezier Curves." 2015.

Lampiran

Lampiran 1
Checklist

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✓	
2. Program dapat melakukan visualisasi kurva Bezier.	✓	
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	✓	

Lampiran 2
Pranala ke repository

https://github.com/rafimaliki/Tucil2_13522137