

LAPORAN TUGAS KECIL 2
MEMBANGUN KURVA BEZIER DENGAN ALGORITMA TITIK
TENGAH BERBASIS *DIVIDE AND CONQUER*



Disusun oleh:
13522093 - Matthew Vladimir Hutabarat
13522098 - Suthasoma Mahardika Munthe

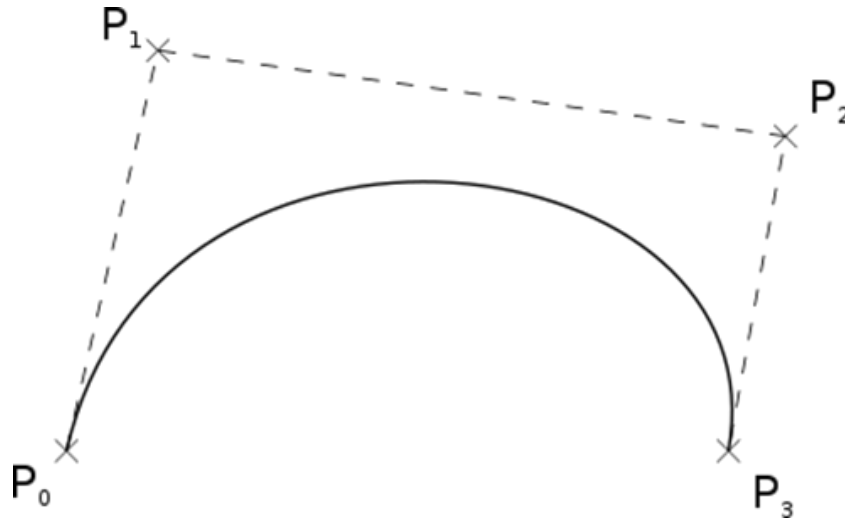
IF2211 - Strategi Algoritma

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

DAFTAR ISI

DAFTAR ISI	2
BAB 1 DESKRIPSI MASALAH	3
BAB 2 TEORI DASAR	6
2.1. Algoritma Brute Force	6
2.2. Algoritma Divide and Conquer	6
BAB 3 ANALISIS DAN IMPLEMENTASI PROGRAM	7
3.1. Analisis dan Implementasi algoritma brute force	7
3.2. Analisis dan Implementasi algoritma Divide and Conquer	8
BAB 4 SOURCE CODE	11
BAB 5 EKSPERIMEN	15
5.1. Input dan Output Algoritma Brute Force dan Divide and Conquer	15
5.2. Analisis Kompleksitas Algoritma Divide and Conquer dengan Brute Force	16
5.3. Analisis Perbedaan Algoritma Divide and Conquer dengan Brute Force	17
BAB 6 PENUTUP	18
6.1. Kesimpulan	18
6.2. Saran	18
6.3. Komentar dan Refleksi	18
6.4. Tabel Checkpoint	18
DAFTAR PUSTAKA	19
LAMPIRAN	20

BAB 1 DESKRIPSI MASALAH



Gambar 1. Kurva Bézier Kubik

(Sumber: https://id.wikipedia.org/wiki/Kurva_B%C3%A9zier)

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 P_0 dan P_1 , maka posisinya dapat dinyatakan dengan persamaan parametrik berikut.

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

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 P_0 ke P_1 . 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 kuadrat** terhadap titik P_0 dan P_2 . Berikut adalah uraian persamaannya.

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

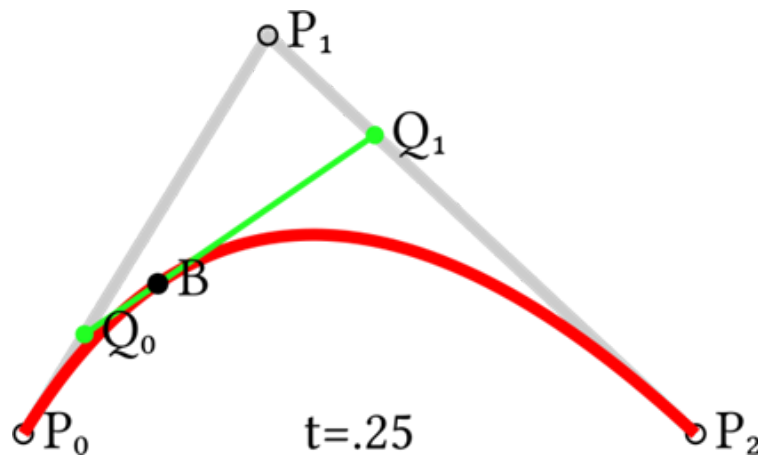
$$Q_1 = B(t) = (1 - t)P_1 + tP_2, t \in [0, 1]$$

$$Q_2 = B(t) = (1 - t)P_2 + tP_3, t \in [0, 1]$$

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

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

Berikut adalah ilustrasi dari kasus diatas.



Gambar 2. Pembentukan Kurva Bézier Kuadrat.

(Sumber: <https://simonhalliday.com/2017/02/15/quadratic-bezier-curve-demo/>)

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.

$$S_0 = B(t) = (1 - t)^3P_0 + 3(1 - t)^2tP_1 + 3(1 - t)t^2P_2 + t^3P_3, t \in [0, 1]$$

$$T_0 = B(t) = (1 - t)^4P_0 + 4(1 - t)^3tP_1 + 6(1 - t)^2t^2P_2 + 4(1 - t)t^3P_3 + t^4P_4, t \in [0, 1]$$

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 Anda diminta untuk mengimplementasikan pembuatan kurva Bézier dengan algoritma titik tengah berbasis **divide and conquer**.

Ilustrasi kasus

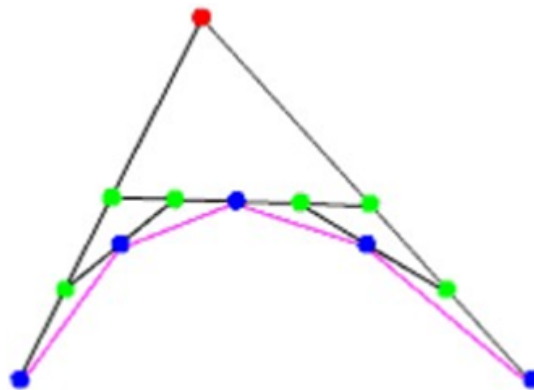
Idenya cukup sederhana, relatif mirip dengan pembahasan sebelumnya, dan dilakukan secara iteratif. Misalkan terdapat tiga buah titik, P_0 , P_1 , dan P_2 , dengan titik P_1 menjadi titik kontrol antara, maka:

- Buatlah sebuah titik baru Q_0 yang berada di tengah garis yang menghubungkan P_0 dan P_1 , serta titik Q_1 yang berada di tengah garis yang menghubungkan P_1 dan P_2 .
- Hubungkan Q_0 dan Q_1 sehingga terbentuk sebuah garis baru.
- Buatlah sebuah titik baru R_0 yang berada di tengah Q_0 dan Q_1 .
- Buatlah sebuah garis yang menghubungkan $P_0 - R_0 - P_2$.

Melalui proses di atas, telah dilakukan 1 buah iterasi dan diperoleh sebuah “kurva” yang belum cukup mulus dengan aproksimasi 3 buah titik. Untuk membuat sebuah kurva yang lebih baik, perlu dilakukan iterasi lanjutan. Berikut adalah prosedurnya.

- Buatlah beberapa titik baru, yaitu S_0 yang berada di tengah P_0 dan Q_0 , S_1 yang berada di tengah Q_0 dan R_0 , S_2 yang berada di tengah R_0 dan Q_1 , dan S_3 yang berada di tengah Q_1 dan P_2 .
- Hubungkan S_0 dengan S_1 dan S_2 dengan S_3 sehingga terbentuk garis baru.
- Buatlah dua buah titik baru, yaitu T_0 yang berada di tengah S_0 dan S_1 , serta T_1 yang berada di tengah S_2 dan S_3 .
- Buatlah sebuah garis yang menghubungkan $P_0 - T_0 - R_0 - T_1 - P_2$.

Melalui iterasi kedua akan tampak semakin mendekati sebuah kurva, dengan aproksimasi 5 buah titik. Anda dapat membuat visualisasi atau gambaran secara mandiri terkait hal ini sehingga dapat diamati dan diterka dengan jelas bahwa semakin banyak iterasi yang dilakukan, maka akan membentuk sebuah kurva yang tidak lain adalah kurva Bézier.



Gambar 3. Hasil pembentukan Kurva Bézier Kuadratik dengan *divide and conquer* setelah iterasi ke-2.

BAB 2

TEORI DASAR

2.1. Algoritma *Brute Force*

Algoritma Brute Force adalah algoritma pemecahan masalah yang dilakukan secara langsung (*straight forward*) dengan menguji semua kemungkinan solusi. Umumnya, Algoritma ini akan menguji setiap kemungkinan secara berurutan sampai menemukan solusi yang optimal.

Algoritma ini cukup sederhana, namun seringkali membutuhkan waktu dan sumber daya komputasi yang besar untuk menyelesaikan permasalahan terutama ketika menangani permasalahan dengan ruang pencarian yang besar. Dibalik kelemahannya, algoritma ini memiliki keunggulan dimana dapat menjawab hampir semua persoalan yang ada dan terkadang persoalan yang ada hanya dapat diselesaikan dengan brute force.

Beberapa contoh persoalan populer yang dapat diselesaikan oleh *Brute Force* ialah:

1. Pemecahan Sandi (Cryptography)
2. Penyelesaian Sudoku
3. Permasalahan N-Puzzle
4. Permasalahan Penyusunan Kado (Subset Sum)

2.2. Algoritma *Divide and Conquer*

Algoritma Divide and Conquer adalah algoritma yang memecahkan persoalan dengan membagi persoalan menjadi beberapa bagian yang lebih kecil lalu menyelesaikan setiap bagian secara terpisah dan kemudian menggabungkan solusi-solusi tersebut untuk mendapatkan solusi akhir.

Umumnya, langkah-langkah umum dalam memecahkan algoritma Divide and Conquer adalah:

1. Divide (Pembagian): Masalah yang besar dibagi menjadi sub-sub masalah yang lebih kecil dan serupa.
2. Conquer (Penyelesaian): setiap sub-masalah diselesaikan secara rekursif.
3. Combine (Penggabungan): Solusi dari setiap sub-masalah digabungkan untuk mendapatkan solusi akhir dari masalah asli

Beberapa contoh persoalan populer yang dapat diselesaikan oleh *Divide and Conquer* ialah:

1. Pengurutan (Merge Sort, Quick Sort)
2. Game Board Solving (Tic-Tac-Toe, Sudoku)
3. Bezier Curve
4. Binary Search

BAB 3

ANALISIS DAN IMPLEMENTASI PROGRAM

3.1. Analisis dan Implementasi algoritma *brute force*

3.1.1. Analisis *brute force*

Proses pembentukan kurva Bezier menggunakan algoritma *brute force* dirancang dengan menghitung posisi setiap titik untuk nilai t yang berada pada interval 0 hingga 1 secara inklusif. Penulis melakukan generalisasi sehingga algoritma *brute force* dapat melakukan proses perhitungan kurva Bezier dengan n titik kontrol.

Generalisasi digunakan untuk membandingkan hasil kurva yang diperoleh dari proses algoritma *divide and conquer* yang telah digeneralisasikan juga untuk memproses n titik kontrol.

Kurva yang dihasilkan dari n titik kontrol akan menghasilkan 1 titik pada kurva untuk setiap nilai t . Misalkan ada n titik kontrol, titik pada kurva akan dihasilkan dengan menghubungkan setiap titik secara bertahap sehingga menyisakan satu titik saja. Kumpulan n titik, misalkan P_0, P_1, P_2 , hingga P_n , pada proses kalkulasi pertama akan menghasilkan Q_0, Q_1 , hingga Q_{n-1} . Pada proses kedua akan dihasilkan R_0, R_1, R_2 , hingga R_{n-2} . Algoritma *brute force* yang digunakan akan melakukan rekursi perhitungan titik hingga tersisa 2 titik yang harus dihitung posisi pada garis yang menghubungkan kedua titik tersebut dan mengembalikan posisi tersebut sebagai titik pada kurva dengan nilai rasio t .

3.1.2. Implementasi *brute force*

Sesuai dengan analisis yang telah dilakukan di atas, algoritma *brute force* yang penulis gunakan akan memanfaatkan proses rekursi. Jumlah titik yang akan dihasilkan dari algoritma ini akan disesuaikan dengan jumlah titik yang akan dihasilkan algoritma *divide and conquer* di bawah. Jumlah titik adalah sebanyak $2^n + 1$, dengan n adalah jumlah iterasi.

Berikut ini adalah algoritma *brute force* yang penulis gunakan.

1. Hitung banyak titik yang akan dihasilkan sesuai nilai iterasi n yang diterima, yaitu $2^n + 1$ sebut saya banyak titik. Kemudian lakukan inisialisasi perhitungan posisi titik pada kurva dengan nilai t sebanyak banyak titik. Simpan nilai $ratio = 1/2^n$.
2. Inisialisasikan rasio i dengan nilai 0.
3. Jika $i \leq$ banyak titik, lakukan kalkulasi titik dengan banyak titik kontrol sebanyak m dengan nilai $t = i * ratio$.
 - a. Jika $m = 2$ (P_0 dan P_1), maka langsung cari titik yang menghubungkan 2 titik (Q) tersebut dengan rasio nilai t dan dikembalikan.

$$Q = (1 - t)P_0 + tP_1$$
 - b. Jika $m > 2$, misalkan (P_1, P_2, P_3 , hingga P_n), maka lakukan perhitungan antara P_1-P_2, P_2-P_3 , hingga $P_{n-1}-P_n$.

$$Q_1 = (1 - t)P_1 + tP_2$$

$$Q_2 = (1 - t)P_2 + tP_3$$

$$Q_3 = (1 - t)P_3 + tP_4$$

$$\dots$$

$$Q_{n-1} = (1 - t)P_{n-1} + tP_n$$

Kemudian kumpulan titik Q ini diproses ulang melalui proses 3.

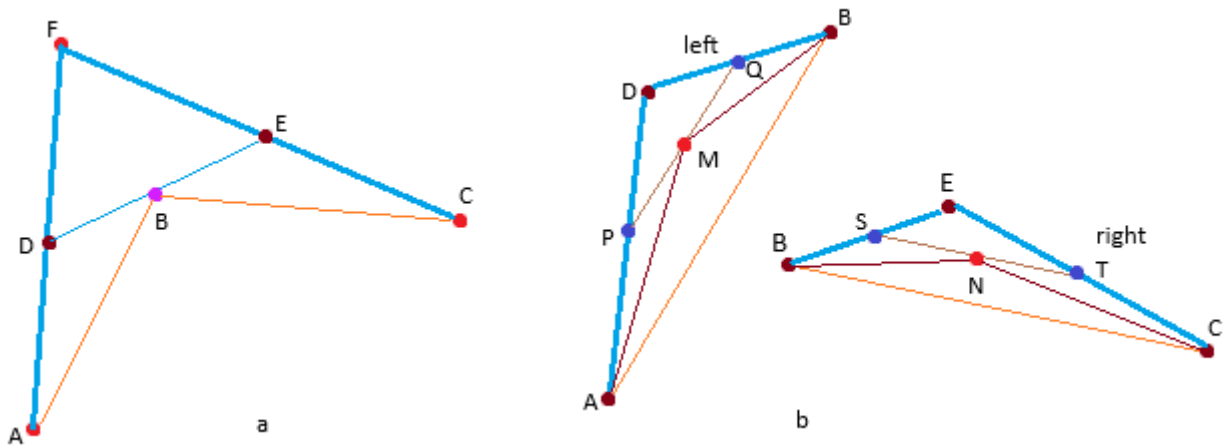
4. Nilai i ditambah sebanyak 1 dan hasil titik dari proses 3 disimpan. Lakukan proses 3 sampai nilai i sudah mencapai banyak titik yang diinginkan.
5. Terminasi program yang menghasilkan titik-titik yang berada pada kurva Bezier menggunakan algoritma *brute force* ini.

3.2. Analisis dan Implementasi algoritma *Divide and Conquer*

3.2.1. Analisis *divide and conquer*

Algoritma *divide and conquer* digunakan untuk melakukan perhitungan titik kurva Bezier pada sebuah n titik kontrol. Sesuai dengan deskripsi masalah yang telah dijelaskan di atas, proses perhitungan titik dilakukan bertahap sesuai iterasi.

Penulis melakukan generalisasi sehingga algoritma yang penulis gunakan dapat menentukan titik pada kurva Bezier dengan n titik kontrol. Pada kasus 3 titik kontrol di atas (kurva kuadratik), setiap iterasi selanjutnya memerlukan 3 titik kontrol baru yang dihasilkan dari iterasi sebelumnya untuk menghasilkan titik baru.



Gambar 3.2.1 (a) *Divide and conquer* iterasi 1 (b) *Divide and conquer* iterasi 2

Pada proses iterasi kedua, titik yang sebelumnya dihasilkan pada proses iterasi pertama, yaitu A , D , dan B untuk bagian kiri dan B , E , dan C untuk bagian kanan (lihat Gambar 3.2.1 a), digunakan untuk menghasilkan titik baru yaitu titik M dan N (lihat Gambar 3.2.1 b). Proses *divide and conquer* terlihat pada Gambar 3.2.1 b, yaitu setelah proses iterasi pertama akan dihasilkan beberapa titik baru pada bagian kiri (*left*) dan bagian kanan (*right*). Secara rekursif akan dilakukan proses perhitungan terhadap bagian *left* dan bagian *right* sesuai jumlah iterasi yang ditentukan.

Generalisasi yang diterapkan pada n titik kontrol untuk iterasi selanjutnya juga memerlukan titik acuan sebanyak n buah titik yang disimpan pada bagian *left* dan bagian *right*. Namun, untuk menghasilkan satu titik yang berada pada kurva dengan n titik kontrol, diperlukan proses perhitungan titik tengah sebanyak $n(n-1)/2$ kali. Terlihat

pada proses 3 titik kontrol pada Gambar 3.2.1 diperlukan 3 kali proses perhitungan titik tengah.

3.2.2. Implementasi *divide and conquer*

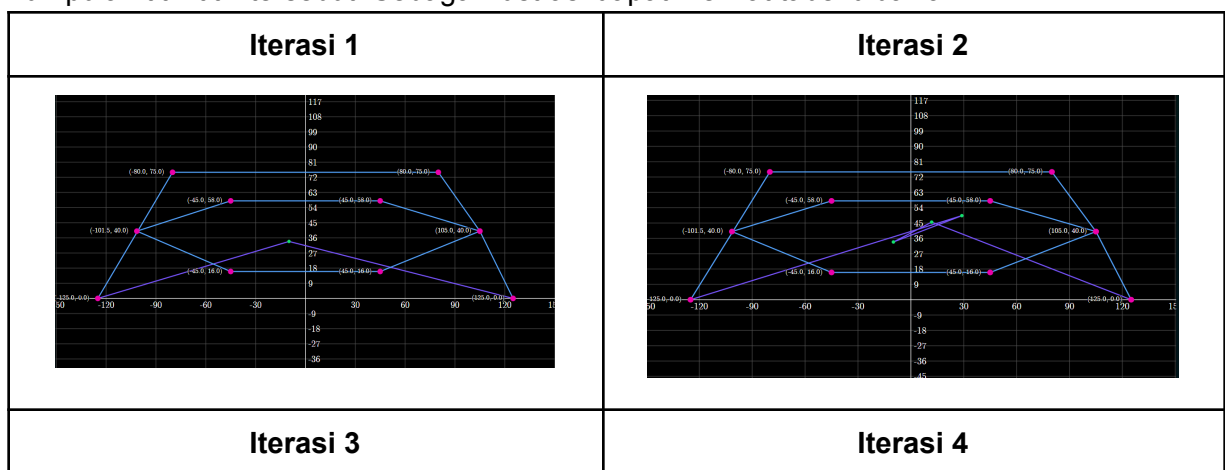
Setelah analisis yang dilakukan ide *divide and conquer* akan membagi hasil setiap iterasi menjadi bagian *left*, *mid*, dan *right*. Bagian *mid* adalah titik yang berada pada kurva yang diperoleh pada proses iterasi tersebut. Bagian *left* dan *right* adalah bagian yang akan digunakan untuk menentukan titik baru pada kurva sehingga grafik kurva menjadi lebih akurat dan lebih kontinu.

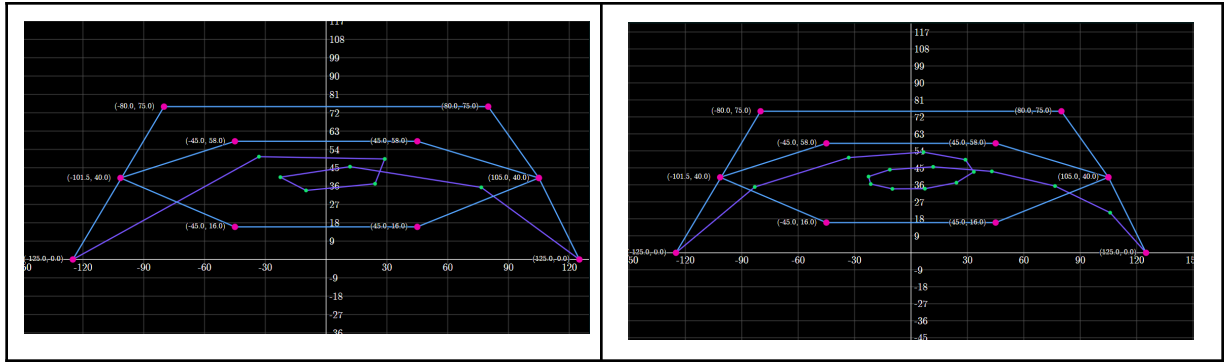
Berikut ini adalah proses algoritma *divide and conquer* untuk menentukan kurva Bezier.

1. Titik-titik kontrol sebanyak n buah dan jumlah iterasi ditetapkan.
2. Lakukan proses perhitungan titik tengah (menjadi satu titik, yaitu *mid*) dari n titik kontrol jika tahap iterasi belum melebihi iterasi yang ditentukan. Bagian *left* akan diisi titik awal yang dihasilkan dari setiap tahap hingga menjadi satu titik. Contohnya, pada Gambar 3.2.1, titik A , D , dan B akan disimpan pada bagian *left*. Pada bagian *right* setiap titik akhir yang dihasilkan dari setiap tahap hingga mencapai satu titik akan disimpan pada bagian ini. Contohnya, pada Gambar 3.2.1 bagian *right* akan berisi titik B , E , dan C . Jumlah titik pada bagian *left* dan *right* selalu berjumlah n .
3. Lakukan tahap 2 untuk bagian *left* dan *right*.
4. Kemudian hasil dari bagian *left* dan *right* disambungkan dengan *mid* yang telah disimpan dengan urutan *left-mid-right* dan disimpan sebagai *points*.
5. Proses di atas tidak mengikutsertakan titik awal yaitu P_1 dan titik akhir P_n . Oleh karena itu, sambungkan titik awal dan titik akhir menjadi P_1 -*points*- P_n .
6. Hasil sambungan titik-titik pada proses 5 adalah kurva Bezier yang dihasilkan.

3.2.3. Ilustrasi algoritma *divide and conquer*

Seiring bertambahnya jumlah iterasi maka jumlah titik titik pembentuk kurva akan semakin banyak dan semakin kelihatan bentuk kurva Bezier yang dibentuk oleh kumpulan titik titik tersebut. Sebagai ilustrasi dapat melihat tabel dibawah ini.





BAB 4

SOURCE CODE

Source code divide and conquer

```
package bezier

func (bp BezierPoints) FindCurve(maxIter int) BezierPoints {
    result := FindPoints(bp, 0, maxIter)
    result.InsertFirst(bp.List[0])
    result.InsertLast(bp.List[len(bp.List)-1])

    return result
}

func FindPoints(bp BezierPoints, iter, maxIter int) BezierPoints {
    if iter >= maxIter {
        return BezierPoints{}
    }
    left, mid, right := FindMidPoint(bp)
    leftPoints := FindPoints(left, iter+1, maxIter)
    rightPoints := FindPoints(right, iter+1, maxIter)

    mid.InsertAfter(rightPoints)
    leftPoints.InsertAfter(mid)
    return leftPoints
}

func FindMidPoint(bp BezierPoints) (BezierPoints, BezierPoints, BezierPoints) {
    tmpBP := []BezierPoints{}

    left := BezierPoints{}
    left.InsertLast(bp.List[0])

    iter := bp.Neff - 1

    right := BezierPoints{}
    right.InsertFirst(bp.List[iter])
}
```

```

    for i := 0; i < iter; i++ {
        if i == 0 {
            tmpBP = append(tmpBP, BezierPoints{})
            traversal := bp.Neff - 1
            for j := 0; j < traversal; j++ {
                tmpBP[0].InsertLast(MidPoint(bp.List[j], bp.List[j+1]))
            }
        } else {
            tmpBP = append(tmpBP, BezierPoints{})
            traversal := tmpBP[i-1].Neff - 1
            for j := 0; j < traversal; j++ {
                tmpBP[i].InsertLast(MidPoint(tmpBP[i-1].List[j],
tmpBP[i-1].List[j+1]))
            }
        }
        left.InsertLast(tmpBP[i].List[0])
        right.InsertFirst(tmpBP[i].List[iter-i-1])
    }
    return left, tmpBP[iter-1], right
}

```

Source code brute force

```

package bezier

func GetRatioPoint4(points BezierPoints, ratio float64) Point {
    if points.Neff == 2 {
        tmpPoint := Point{(1-ratio)*points.List[0].X + ratio*points.List[1].X,
(1-ratio)*points.List[0].Y + ratio*points.List[1].Y}
        return tmpPoint
    }

    newpoints := BezierPoints{}
    for i := 1; i < points.Neff; i++ {
        tmpPoint := Point{(1-ratio)*points.List[i-1].X +
ratio*points.List[i].X, (1-ratio)*points.List[i-1].Y + ratio*points.List[i].Y}
        newpoints.InsertLast(tmpPoint)
    }
}

```

```

    }
    return GetRatioPoint4(newpoints, ratio)
}

func (bp BezierPoints) DrawCurveBruteForce(n_point int) BezierPoints {
    points := BezierPoints{}

    add := 1 / float64(n_point-1)

    for i := 0; i < n_point; i++ {
        points.InsertLast(GetRatioPoint4(bp, float64(i)*add))
    }

    return points
}

```

Source code utility (point)

```

package bezier

type Point struct {
    X float64 `json:"x"`
    Y float64 `json:"y"`
}

func MidPoint(p1, p2 Point) Point {
    mid_X := (p1.X + p2.X) / 2
    mid_Y := (p1.Y + p2.Y) / 2
    return Point{mid_X, mid_Y}
}

type BezierPoints struct {
    List []Point
    Neff int
}

func (bp *BezierPoints) InsertBefore(other BezierPoints) {
    bp.List = append(other.List, bp.List...)
}

```

```
    bp.Neff = bp.Neff + other.Neff
}

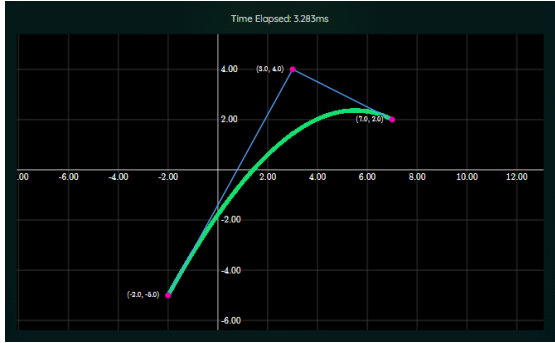
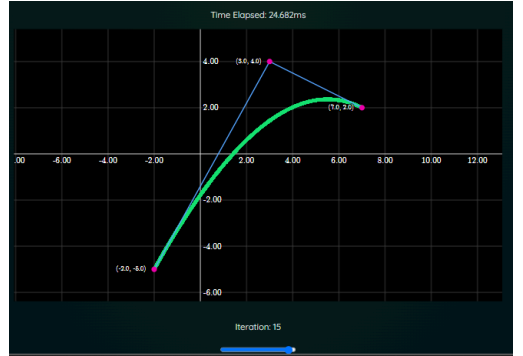
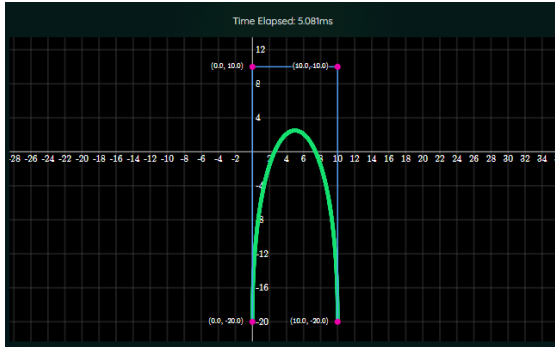
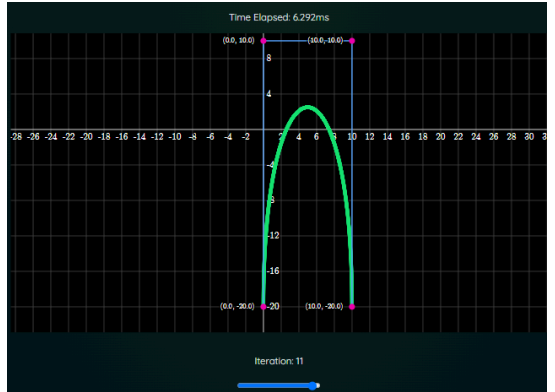

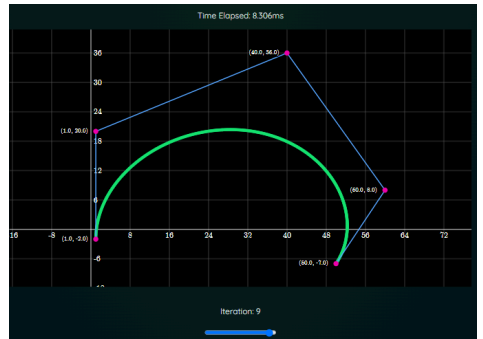
func (bp *BezierPoints) InsertAfter(other BezierPoints) {
    bp.List = append(bp.List, other.List...)
    bp.Neff = bp.Neff + other.Neff
}

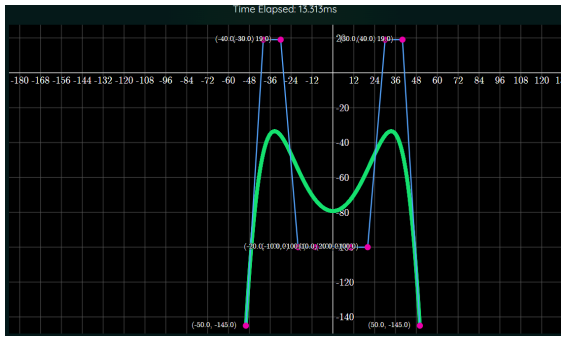
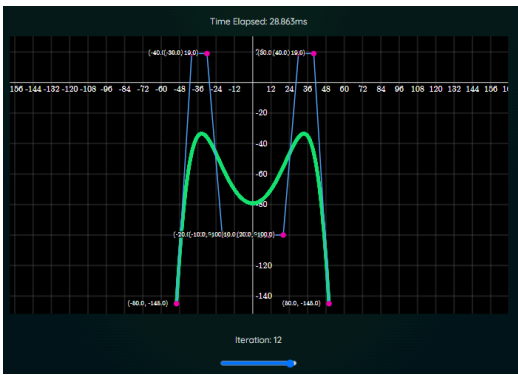
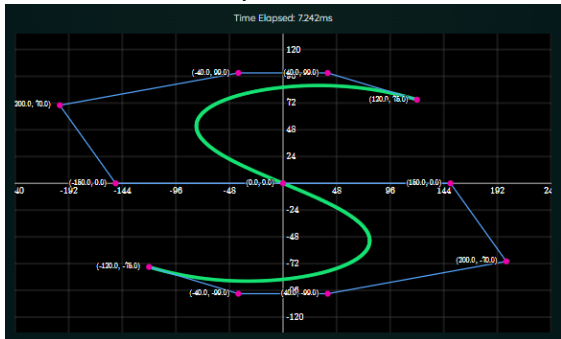

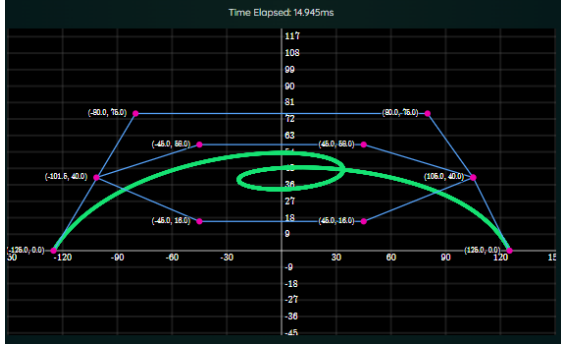
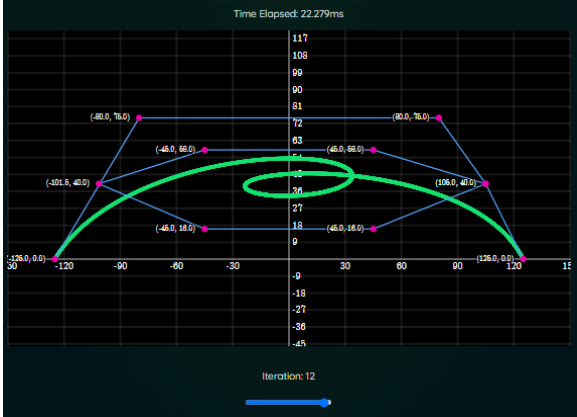
func (bp *BezierPoints) InsertLast(other_point ...Point) {
    bp.List = append(bp.List, other_point...)
    bp.Neff = bp.Neff + len(other_point)
}

func (bp *BezierPoints) InsertFirst(other_point ...Point) {
    bp.List = append(other_point, bp.List...)
    bp.Neff = bp.Neff + len(other_point)
}
```

BAB 5 EKSPERIMEN

5.1. Input dan Output Algoritma *Brute Force* dan Divide and Conquer

Input	Brute Force	Divide and Conquer
Points : (-2,-5), (3,4), (7,2), Iteration : 15	Time Elapsed: 3,283ms 	Time Elapsed: 24,682ms 
Points : (0,-20), (0,10), (10,10), (10,-20) Iteration: 11	Time Elapsed: 5,081ms 	Time Elapsed: 6,292ms 
Points: (1,-2), (1,20), (40,36), (60,8), (50,-7), Iteration: 9	Time Elapsed: 7,793ms 	Time Elapsed: 8,306ms 
Points: (-50,-145), (-40,19),	Time Elapsed: 13,313ms	Time Elapsed: 28,863ms

<p>(-30,19), (-20,-100), (-10,-100), (10,-100), (20,-100), (30,19), (40,19), (50,-145),</p> <p>Iteration: 12</p>		
<p>Points: (120,75), (40,99), (-40,99), (-200,70), (-150,0), (0,0), (150,0), (200,-70), (40,-99), (-40,-99), (-120,-75),</p> <p>Iteration: 9</p>	<p>Time Elapsed: 7,242ms</p> 	<p>Time Elapsed: 7,884ms</p> 
<p>Points: (-125,0), (-80,75), (80,75), (105,40), (45,16), (-45,16), (-101,5,40), (-45,58), (45,58), (105,40), (125,0),</p> <p>Iteration : 12</p>	<p>Time Elapsed: 14,945ms</p> 	<p>Time Elapsed: 22,279ms</p> 

5.2. Analisis Kompleksitas Algoritma Divide and Conquer dengan Brute Force

5.2.1. Kompleksitas algoritma divide and conquer

$$T(n, 1) = \frac{n(n-1)}{2}$$

$$\begin{aligned}
T(n, k) &= \frac{n(n-1)}{2} + 2T(n, k-1) \\
T(n, k) &= \frac{n(n-1)}{2} + 2\left(\frac{n(n-1)}{2} + 2T(n, k-2)\right) \\
T(n, k) &= (1 + 2 + 4 + \dots + 2^{k-1}) \frac{n(n-1)}{2} \\
T(n, k) &= (2^k - 1) \frac{n(n-1)}{2} = O(2^k n^2),
\end{aligned}$$

dengan mempertimbangkan adanya proses *insertion* titik-titik pada bagian awal list maka untuk setiap iterasi akan ada proses tambahan yaitu insertion dengan kompleksitas 2^{k-1} sehingga fungsi saat ini adalah:

$$\begin{aligned}
T(n, k) &= \frac{n(n-1)}{2} 2^{k-1} + 2T(n, k-1) \\
T(n, k) &= \frac{n(n-1)}{2} 2^{k-1} + 2\left(\frac{n(n-1)}{2} 2^{k-1} + 2T(n, k-2)\right) \\
T(n, k) &= (1 + 2 + 4 + \dots + 2^{k-1}) \frac{n(n-1)}{2} 2^{k-1} \\
T(n, k) &= (2^{k-1} - 1) \frac{n(n-1)}{2} 2^{k-1} \\
T(n, k) &= (2^{k-1} - 1) \frac{n(n-1)}{2} 2^{k-1} \\
T(n, k) &= (2^{2k-2} - 2^{k-1}) \frac{n(n-1)}{2} = O(2^{2k} n^2) = O(4^k n^2)
\end{aligned}$$

Hal inilah yang menyebabkan pada bagian eksperimen, waktu eksekusi menggunakan algoritma *brute force* lebih singkat dibandingkan waktu eksekusi menggunakan algoritma *divide and conquer*.

Jika penulis memilih proses *insertion* yang lebih baik maka tentunya secara konsep $T(n, k)$ menggunakan algoritma *divide and conquer* menjadi lebih baik daripada algoritma *brute force*.

5.2.2 Kompleksitas algoritma *brute force*

$$T(n, k) = (2^k + 1) \frac{n(n-1)}{2} = O(2^k n^2)$$

dengan:

n = jumlah titik kontrol

k = jumlah iterasi

5.3. Analisis Perbedaan Algoritma *Divide and Conquer* dengan *Brute Force*

Berdasarkan hasil eksperimen pada bagian 5.1. didapatkan bahwa hasil algoritma *Divide and Conquer* relatif lebih lambat ketimbang memakai algoritma *Brute Force*. Hal ini dapat dibuktikan dengan kompleksitas *brute force* $T(n, k) = O(2^k n^2)$ dengan kompleksitas *divide and conquer* $T(n, k) = O(4^k n^2)$. Dimana untuk setiap titik kontrol dan jumlah iterasi yang sama akan menyebabkan konstanta kompleksitas *divide and conquer* selalu lebih besar daripada konstanta kompleksitas *brute force*, hal ini membuat waktu kompilasi algoritma *divide and conquer* lebih lama ketimbang algoritma *brute force*.

BAB 6 PENUTUP

6.1. Kesimpulan

Dalam membentuk kurva Bezier dapat menggunakan algoritma *Divide and Conquer* atau *Brute Force*. Berdasarkan hasil perhitungan kompleksitas dan hasil eksperimen didapatkan bahwa menggunakan algoritma *Brute Force* dalam membentuk kurva Bezier lebih optimal ketimbang memakai algoritma *Divide and Conquer*.

6.2. Saran

Dalam membuat visualisasinya, untuk titik yang banyak dengan jarak yang besar akan membuat grafik terlihat kecil dan tidak jelas karena overlap dengan label label sumbu x atau y. Menurut kami, membuat visualisasi yang lebih besar seperti seluruh layar akan membuat visualnya terlihat lebih jelas dan enak dipandang.

6.3. Komentar dan Refleksi

Kami senang mendapatkan tugas kecil membuat kurva Bezier. Di tugas kali ini, kami belajar lebih dalam tentang algoritma *Divide and Conquer* dan *Brute Force*. Dalam visualisasinya kami belajar lebih dalam tentang framework React beserta library tambahan bernama Mafs dalam menampilkan grafik kartesius.

6.4. Tabel Checkpoint

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

DAFTAR PUSTAKA

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Tucil2-2024.pdf>

https://id.wikipedia.org/wiki/Kurva_B%C3%A9zier

[https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-\(2024\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian2.pdf)

[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/2021-2022/Algoritma-Brute-Force-(2022)-Bag1.pdf)

LAMPIRAN

LINK REPOSITORY

Link repository GitHub : https://github.com/sotul04/Tucil2_13522093_13522098

PEMBAGIAN TUGAS

NIM	Nama	Tugas
13522093	Matthew Vladimir Hutabarat	Visualizer, laporan
13522098	Suthasoma Mahardika Munthe	Algoritma, laporan