Triangulation

توضیح الگوریتم :

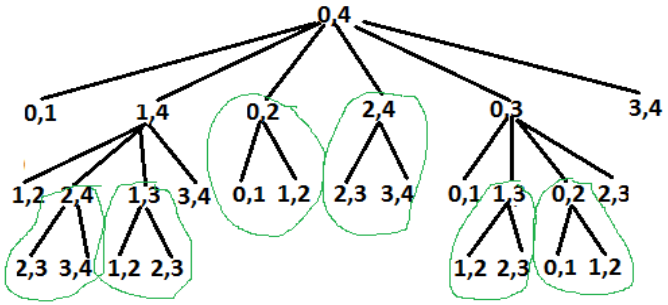
```
value = minimum(value, (triangulation(points, i, k) +  triangulation(points, k, j) + cost(points, i, k, j)))
```

قسمت اول :

این مسأله را میتوان بصورت بازگشتی نیز حل کرد که الگوریتم آن به شکال بالا میشود.

ایده حل آن به این شکل است که شکل مورد نظر را به سه بخش تقسیم میکنیم و الگوریتم را به طور بازگشتی روی هر یک از این سه قسمت اجرا میکنیم.

ما ابتدا مسأله را حل شده فرض میکنیم و یکی از قطر ها را انتخاب میکنیم و باید زیرمسأله های بوجود آمده نیز بهینه باشند.



Recursion Tree for recursive implementation. Overlapping subproblems are encircled.

میبینیم که بسیار از زیرمسأله ها باهم تداخل دارند، پس میتوان از برنامه نویسی پویا استفاده کرد.

```
value = array[i][k] + array[k][j] + cost(points, i, j, k)
```

برای این کار از یک ماتریس [n][n] استفاده میکنیم و خانه ی [0][n] ماتریس جواب مسأله یعنی کمترین مقدار طول قطر ها است.

قسمت دوم :

تا اینجای کار کمترین مقدار طول قطر ها به دست آمد .

اما ما باید قطر هارا را رسم کنیم ، برای اینکار باید یک ماتریس P درنظر بگیریم و در هر مرحله مقدار k را وارد این ماتریس کنیم.

برای مثال : میخواهیم قطرهای یک n ضلعی را پیدا کنیم :
برای شروع میبینیم که خانه ی [n-1][0] ماتریس p چند است ، با فرض اینکه k باشد ما بصورت بازگشتی 0 تا n-1 را از مقدار k میشکنیم بدین صورت که عبارت بالا به [n-1][k]+[k][0] شکسته میشود.
و ما در هرمرحله چک میکنیم که آیا این دو مقدار تشکیل یک قطر میدهند یا خیر.

```
def findDiameter(i, j, k, points, matrixp):
    if j - i < 2:
        return

    if isNotSide(points, points[i], points[j]):
        diameter.append(i)
        diameter.append(j)

    findDiameter(i, k, matrixp[i][k], points, matrixp)
    findDiameter(k, j, matrixp[k][j], points, matrixp)
    return diameter
```

توضیح کد :

1) فایل model.Models :

در این فایل مدل point وجود دارد که نشان دهنده ی یک نقطه است و دارای مقدار x,y میباشد.

2) فایل src.triangulation :

مهمترین فایل برنامه که حاوی logic اصلی برنامه نیز میباشد :

خط 82 که متد main وجود دارد و شروع برنامه است که میتوان از دیتا های دیفالت یا ورودی استفاده کرد.

خط 54 که متد triangulation وجود دارد و پیاده سازی همان الگوریتمی است که در بالا توضیح دادم.

خط 41 که متد findDiamete وجود دارد و بصورت بازگشتی همانطور که در قسمت دوم توضیح الگوریتم توضیح دادم مختصات قطر هارا پیدا میکند

خط 35 متد distance که فاصله ی بین دو نقطه را حساب میکند

خط 20 متد cost که هزینه ی هر قطر را حساب میکند

```
def cost(points, i, j, k):
    p1 = points[i]
    p2 = points[j]
    p3 = points[k]
    totalcost = 0
    if isNotSide(points, p1, p2):
        totalcost = totalcost + distance(p1, p2)
    if isNotSide(points, p2, p3):
        totalcost = totalcost + distance(p2, p3)
    if isNotSide(points, p3, p1):
        totalcost = totalcost + distance(p3, p1)

    return totalcost / 2
```

نحوه ی کار کرد این متد به این صورت است که ابتدا چک میکند دو نقطه ی موردنظر ما رأس های مجاور نباشند سپس فاصله ی دو نقطه را حساب میکند .

از آنجایی که در این متد هر قطر دوبار حساب میشود پس باید مقدار نهایی را نصف کنیم.

خط 9 متد isNotSide که دو نقطه میگیرد و چک میکند که دو نقطه مجاور نباشند .

```
def isNotSide(points, p1, p2):
    for i in range(0, len(points) - 1):
        if (p1 == points[i] and p2 == points[i + 1]) or (p1 == points[i + 1] and p2 == points[i]):
            return False

    if (p1 == points[len(points) - 1] and p2 == points[0]) or (p1 == points[0] and p2 == points[len(points) - 1]):
        return False

    return True
```

3) فایل Ui.Ui :

در این فایل با استفاده از Tiknter گرافیک برنامه زده شده.

که من در این مرحله برای اینکه گرافیک خوبی داشته باشیم همه ی داده هارا 15 برابر کردم .

سپس با استفاده از مختصات رأس ها و داده هایی که از متد findDiamete بدست آمده شکل مورد نظر رسم میشود

مثال :

ورودی :

```
pointList = [Point(40, 30), Point(30, 40), Point(20, 40), Point(10, 30),
             Point(10, 20), Point(20, 10), Point(30, 10), Point(40, 20)]
```

خروجی کنسول :

```
--------------------------------------------------------------------------------
Enter 0 for default data and 1 for for your own data : 0
1-Polygon, 2-Octagonal(1), 3-Octagonal(2), 4-Rectangle : 3
Cost is : 121.06549570167537
```

خروجی گرافیکی :