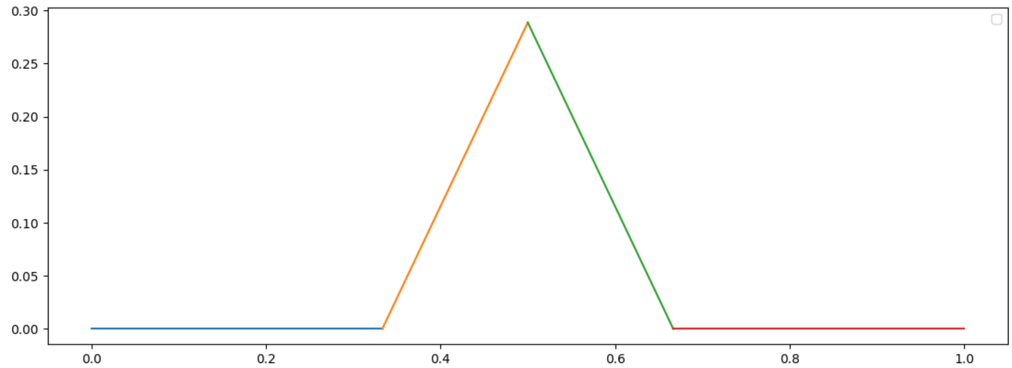
گزارش کار تمرین اول شبیه‌سازی رایانه‌ای در فیزیک

علی اکرامیان – 99100563

تمرین 2.1 (مجموعه‌ی کوخ):

من برای تولید این مجموعه، دو تابع تعریف کردم. یکی تابع op(obj) که یک آبجکت را می‌گیرد و ابتدا آن را 1/3 کرده و ذخیره می‌کند. سپس این شیئ یک‌سوم شده را 60 درجه می‌چرخاند و به طول 1/3 انتقال می‌دهد و بعد از آن همان شيي یک‌سوم شده را 1/3 به عقب می‌گرداند سپس آن را منفی 60 درجه می‌چرخاند و بعد از آن 2/3 واحد انتقال می‌دهد. بعد از آن نیز شیئ 1/3 شده را 2/3 انتقال می‌دهد. این 4 کار همان کاری‌ست که برای ساخت فراکتا نیاز داریم. حال کافی‌ست یک شیئ به آن بدهیم تا کار کند! تابع دوم نیز تابع snow(n) هست که یک تابع بازگشتی‌ست که n که مرتبه‌ی فراکتال را نشان می‌دهد. و این تابع مرتبه‌ی قبلی رو به همین تابه می‌دهد که این تابع نیز مرتبه‌ي n-1 را به تابع op پاس می‌دهد که تابع op نیز آن 4 کار را روی آن انجام داده و نتیجه را با کتابخانه‌ی matplotlib رسم می‌کند.
مثلا اگر به آن یک خط که از نقطه‌ی (0,0) به (0,1) رسم شده به آن بدهیم، نتیجه‌ی زیر را می‌دهد:



کد در صفحه‌ی بعد آمده است.

چند توضیح دیگر:

1. ماتریس R ماتریس دوران 60 درجه است.
2. برای چرخاندن نیز از ضرب دو ماتریس دوران و نقال استفاده شده که ترانهاده کردن‌ها نیز برای چرخش منفی 60 و درست بودن پرینت آخر کار است.
3. برای تبدیل سوم ابتدا شیئ را عقب می‌کشم و سپس می‌چرخانم تا لازم نباشد ماتریس دوران جدید بنویسم.
4. این کد مشکلی که دارد این است که هر بار که اجرا می‌شود، مرتبه‌ي فراکتال بالا می‌رود و برای مراتب بالاتر درست کار نمی‌کند و نتوانستم درستش کنم :(

simulation > Snowflake.py > ...

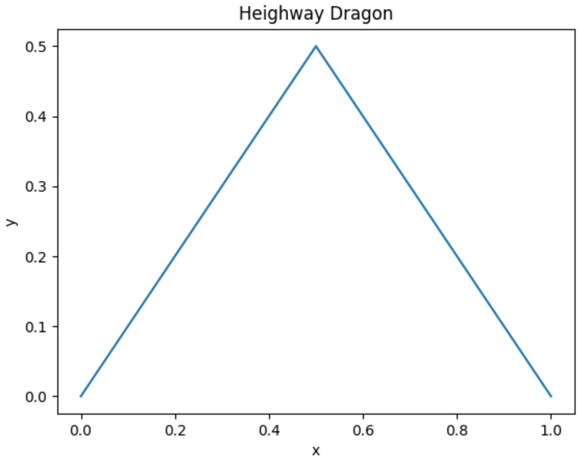
```python
import numpy as np
import matplotlib.pyplot as plt
line = np.array([[0, 0], [1, 0]])
angle = np.pi / 3
R = np.array([[np.cos(angle), -np.sin(angle)], [np.sin(angle), np.cos(angle)]])
def op(obj):
    obj1 = obj / 3
    rotated_obj1 = np.dot(R, obj1.T).T
    translation_vector1 = np.array([1/3, 0])
    obj2 = rotated_obj1 + translation_vector1
    pre_translation_vector = -np.array([1/3, 0])
    pre_translated_obj = obj1 + pre_translation_vector
    rotated_obj = np.dot(R.T, pre_translated_obj.T).T
    translation_vector = np.array([2/3, 0])
    obj3 = rotated_obj + translation_vector
    translation_vector2 = np.array([2/3, 0])
    obj4 = obj1 + translation_vector2
    ans = np.array([obj1,obj2,obj3,obj4])
    print(ans,"ans")
    return ans
def snow(n):
    if n<1:
        return line
    else:
        a = op(snow(n-1))
        return a
ans = snow(1)
obj1 = ans[0]
obj2 = ans[1]
obj3 = ans[2]
obj4 = ans[3]
plt.plot(obj1[:,0],obj1[:,1])
plt.plot(obj2[:,0],obj2[:,1])
plt.plot(obj3[:,0],obj3[:,1])
plt.plot(obj4[:,0],obj4[:,1])
plt.legend()
plt.show()
```

تمرین 2.2 (اژدهای هی‌وی):

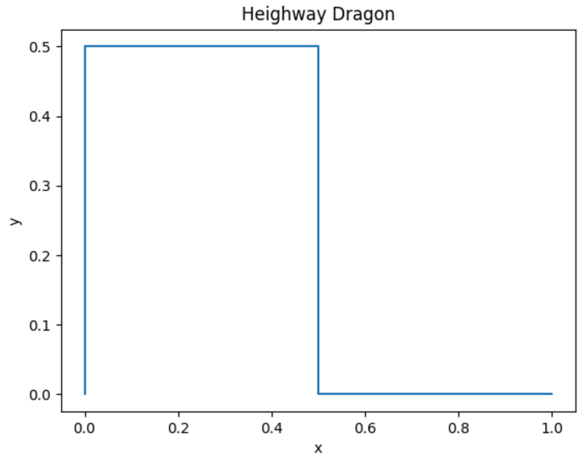
من برای تولید این فراکتال یک تابع بازگشتی Dragon(n) تعریف کرده‌ام که n تعداد بار تکرار (مرتبه) این اژدها است. این تابع دو بخش دارد. یکی پس از گرفتن خط (در مرحله‌ي اول) آن را می‌چرخاند 45 درجه (با ماتریس دوران R) و سپس طول آن را درست می‌کنیم تا اندازه‌ي قطر مربع شود (تقسیم بر رادیکال 2 می‌کنیم) سپس همین کار را برای نصفه‌ی راست خط نیز انجام می‌دهیم (به ترانهاده‌ی ماتریس دوران نیاز داریم و یک انتقال تا در جای درست قرار گیرد). خروجی تابع نیز تمام مختصات نقاط است (pts).
برای کشیدن نیز یک آرایه گذاشته‌ام تا تمام ایکس‌ها و وای‌ها را بگیرد (که نقاط شکستگی متناظر هم‌اند) و آنها را رسم می‌کند.
حال اگر این تابع به صورت بازگشتی تکرار شود، مرتبا این نقاط را اپدیت می‌کند و شکل نهایی اژدها را به ما می‌دهد. در زیر کد را مشاهده می‌کنید:

```
simulation > Dragon.py > ...
import matplotlib.pyplot as plt
import numpy as np
angle=np.pi/4
R=np.array([[np.cos(angle),-np.sin(angle)],
            [np.sin(angle),np.cos(angle)]])
R=R/np.sqrt(2)
def Dragon(n):
    if n<1:
        return [[0,0],[1,0]]
    else:
        pts0=Dragon(n-1)
        pts=[]
        for x,y in pts0:
            [x1,y1]=R @ [x,y]
            pts.append([x1,y1])
        for x,y in reversed(pts0):
            [x2,y2]=[1,0] - R.T @ [x,y]
            pts.append([x2,y2])
        return pts
points = Dragon(2)
x = [p[0] for p in points]
y = [p[1] for p in points]
print(x)
print(y)
plt.plot(x,y)
ft=fontdict={'family':'sans-serif'}
plt.title("Heighway Dragon",ft)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

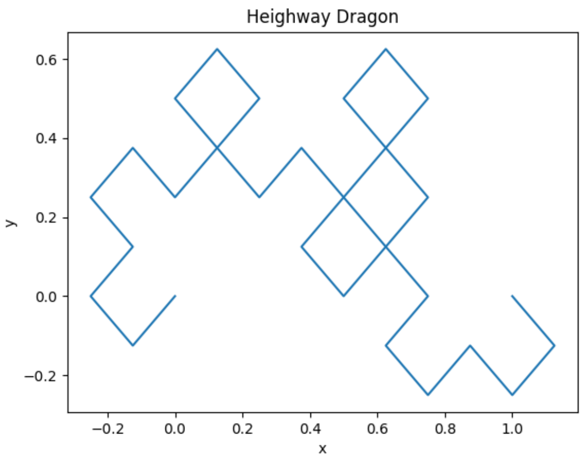
حال شکل مراحل مختلف را با هم می‌بینیم از اجرای این کد:
ابتدا از خطی که توسط دو نقطه‌ی [0,0],[1,0] رسم می‌شود شروع می‌کنیم و داخل تابع می‌اندازیم:



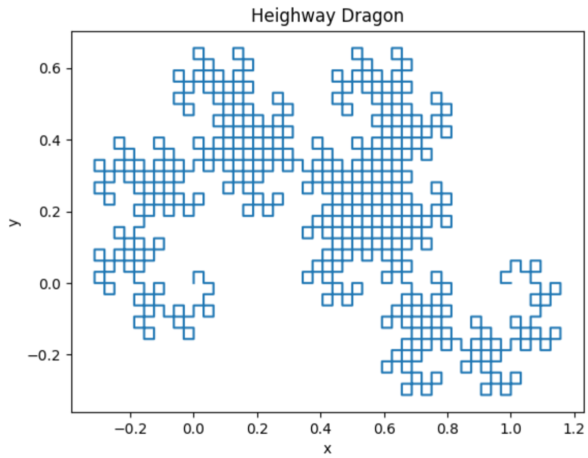
مرحله‌ی 1



مرحله‌ی 2

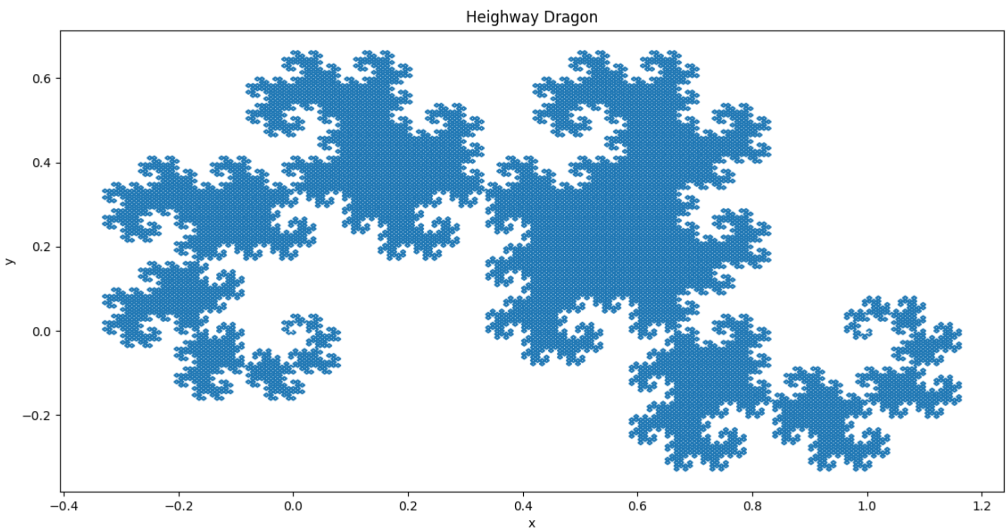


مرحله‌ی 5



مرحله‌ی 10

و مرحله‌ی 15 به صورت زیر است:



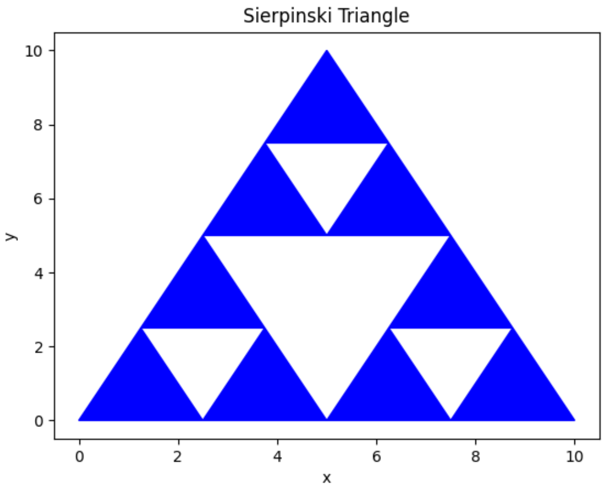
تمرین 2.3 (مثلث سرپینسکی):

در این تمرین من یک تابع بازگشتی triangle تعریف کرده‌ام که مرتبه‌ی صفرمش یک مثلث از مختصات (0,0) به طول ضلع L می‌سازد و داخل آن را به رنگ آبی می‌کند. در مراحل بعدی، که بیشتر از 0 است، این تابع سه بار خودش در مرحله‌ی پیش را صدا می‌زند و با انتقال و نصف کردن طول ضلع خودش در مرحله‌ی قبل می‌توان این این مثلث سرپیسنکی مرحله‌ی n را رسم کرد. چون تابع سه بار خود قبلی اش را صدا می‌زند پس $f(n) = 3 f(n-1)$ پس مرتبه‌ی این الگوریتم از نوع $O(3^n)$ است که تا مرحله‌ی 6 ام من پیش می‌روم و زمان منطقی ای میبرد ولی در مراتب بالاتر به دلیل بالا رفتن حجم محاسبات (هر سری کل قبلی‌ها را باید حساب کند) دیگر زمان منطقی‌ای نمی‌برد و خیلی طول می‌کشد (مثلا برای 10 خیلی صبر کردم!). می‌توان این کد را با ریختن نقاط راسی مثلث‌های قبلی در یک آرایه ((triangle(n-1)) سرعت اجرا را بالابرد که هر سری نخواهد همه‌اش را دوباره حساب کند ولی چون رزولیشن صفحه‌ی من از مرحله‌ی 6 به بعد عملا فرقی با هم نداشتند پس این بهینه‌سازی برای من که سودی ندارد پس انجامش ندادم!
کد زیر این مثلث را به ازای مختصات راس شروع (0,0) و طول ضلع 10 و مرتبه‌ی 6 حساب می‌کند:

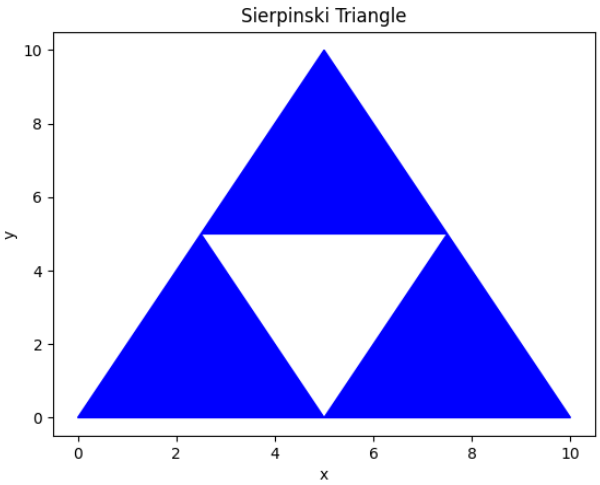
simulation > Sierpinski.py > ...

```python
import matplotlib.pyplot as plt
import numpy as np

def triangle(n,x,y,L):
    if n>0:
        triangle(n-1,x,y,L/2)
        triangle(n-1,x+L/2, y,L/2)
        triangle(n-1,x+L/4,y +L/2,L/2)
    else:
        Xs=np.array([x,x+L,x+L/2])
        Ys=np.array([y,y,y+L])
        plt.subplot().fill(Xs,Ys,c='b')

triangle(6, 0, 0, 10)

ft=fontdict={'family':'sans-serif'}
plt.title("Sierpinski Triangle",ft)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

شکل‌ها را نیز می‌بینیم در ادامه:

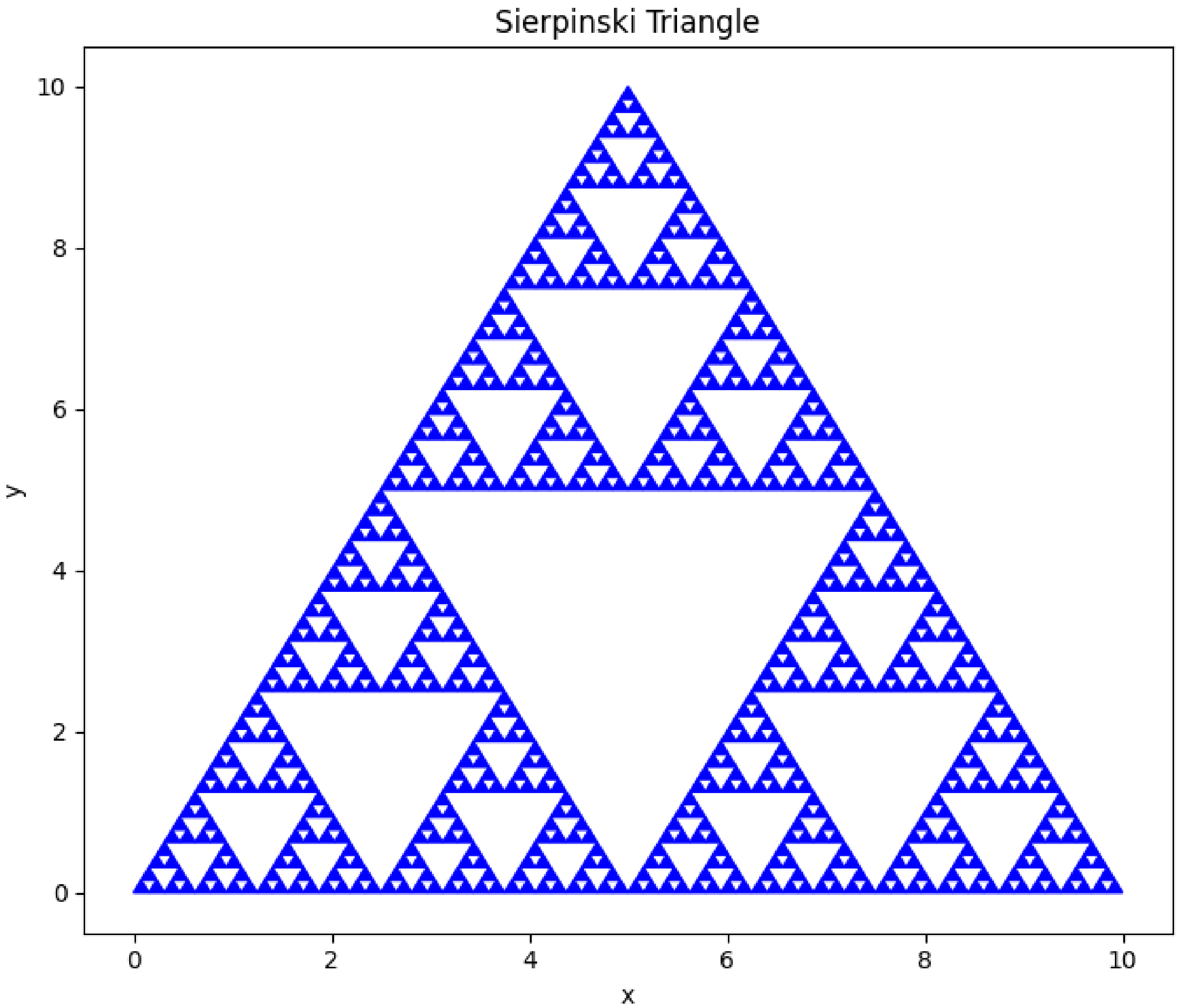


مرحله‌ی 2



مرحله‌ی 1

و در مرحله‌ي 6 ام نیز به صورت زیر می‌شود:



تمرین 2.4 (مثلث سرپینسکی با مثلث خیام-پاسکال):

من یک تابع به نام pascal(n) تعریف کردم که n+1 سط از مثلث پاسکال را تولید می‌کند این تابع می‌تواند اعداد قبل و بعد بالای یک عدد را جمع کند (num[i-1][j-1]+num[i-1][j]) و سپس آن را به یک آرایه بیافزاید تا ذخیره شود و برای رسم ما ازین آرایه‌ی دو بعدی که هر درایه‌ی این آرایه، یک سطر ماست (خودش آرایه‌ست) استفاده می‌کنیم.
حال یک تابع دیگر داریم که من چون هم ایده‌اش پیشرفته بود هم توابعش این را از اینترنت گرفتم و نوشتم و این تابع طرز کار جالبی دارد که توضیح می‌دهم.
این تابع برای درست الاین (align) کردن طول حداکثر کارکتر عددی ما در مثلث را می‌گیرد و دو فاصله قبل و بعد آن میزند تا اعداد در هم فرو نروند در چاپ! حال به هر سطر به اندازه‌ای که از طول بزرگ‌ترین سطر کوچک‌تر است مقدار دلخواهی اسپیس اضافه می‌کند (من این مقدار را در ترمینال خودم ست کردم پس ممکن است به تمیزی تصویری که من در نهایت می‌دهم نباشد وقتی شما اجرا می‌کنید!)
حال می‌توان یک مثلث متساوی‌الضلاع داشت که مثلث خیام‌پاسکال در آن است!
من راستش ایده‌ی اساین کردن یک رنگ به عددهای زوج/فرد و رسم آن را نداشتم پس تا همین‌جا کدها و عکس را تحویل می‌دهم :(

کد زده‌شده:

simulation > pascal.py > ...

```python
def pascal(n):
    num=[[1]]
    for i in range(1,n):
        row=[1]
        for j in range(1,i):
            row.append(num[i-1][j-1]+num[i-1][j])
        row.append(1)
        num.append(row)
    return num

def draw(num):
    maxlen=len(str(num[-1][len(num)//2]))          # Note
    for i in range(len(num)):
        arr=[str(num).center(maxlen) for num in num[i]]
        row=' '.join(arr)
        print('   '*(len(num)-i-1)+row)
draw(pascal(20))

# Note : I got the idea of aligning and centering the pascal triangle
#        from Internet.
```

مثلث کشیده‌شده تا سطر 20 ام:

```
● ali@Ali:~/Documents/VS Code$ /usr/bin/python3 "/home/ali/Documents/VS Code/simulation/pascal.py"
                                                         1
                                                      1     1
                                                   1     2     1
                                                1     3     3     1
                                             1     4     6     4     1
                                          1     5    10    10     5     1
                                       1     6    15    20    15     6     1
                                    1     7    21    35    35    21     7     1
                                 1     8    28    56    70    56    28     8     1
                              1     9    36    84   126   126    84    36     9     1
                           1    10    45   120   210   252   210   120    45    10     1
                        1    11    55   165   330   462   462   330   165    55    11     1
                     1    12    66   220   495   792   924   792   495   220    66    12     1
                  1    13    78   286   715  1287  1716  1716  1287   715   286    78    13     1
               1    14    91   364  1001  2002  3003  3432  3003  2002  1001   364    91    14     1
            1    15   105   455  1365  3003  5005  6435  6435  5005  3003  1365   455   105    15     1
         1    16   120   560  1820  4368  8008 11440 12870 11440  8008  4368  1820   560   120    16     1
      1    17   136   680  2380  6188 12376 19448 24310 24310 19448 12376  6188  2380   680   136    17     1
   1    18   153   816  3060  8568 18564 31824 43758 48620 43758 31824 18564  8568  3060   816   153    18     1
1    19   171   969  3876 11628 27132 50388 75582 92378 92378 75582 50388 27132 11628  3876   969   171    19     1
```