به نام خدا

پروژه اوّل شبیه‌سازی کامپیوتری

مهدی معصوم پور 98243055

محمدحسین ارسلان 98243005

دید کلی پس از انجام پروژه

به طور کلی هدف پروژه نشان دادن تاثیر افزایش دفعات تکرار آزمایش است که بنابر قضیه اعداد بزرگ منجر به صفر شدن اختلاف گزارش‌های شبیه‌سازی و گزارش و محاسبات تحلیلی می‌گردد.

رویکردمان در گزارش متنی این پروژه توضیح عملکرد توابع مختلف و برداشت‌هایمان از نتیج و تصاویر نمودار و تحلیل باتوجه به درخواست‌های مکتوب در صورت متنی پروژه نخست درس است.

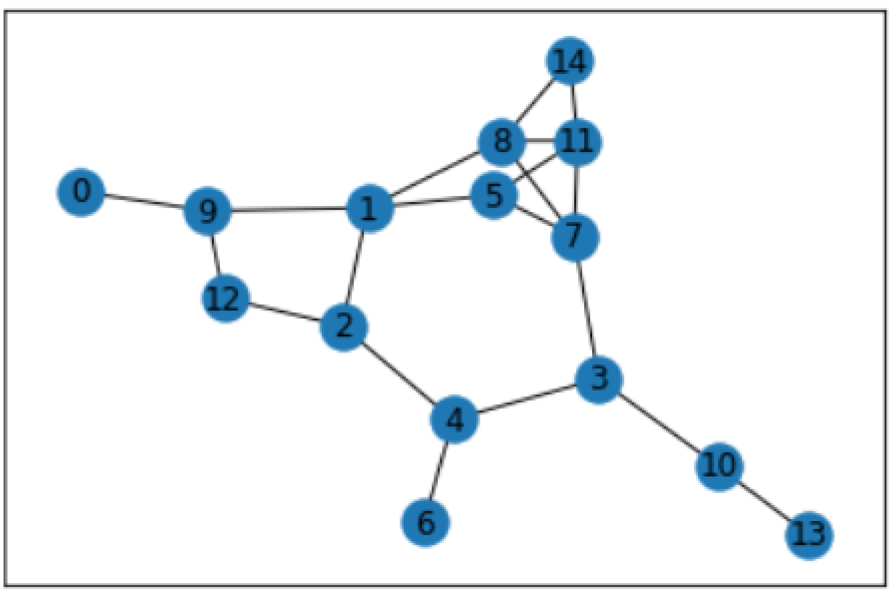
خواسته اول

```python
def generate_random_graph(n , m):
    all_edges_possible = int(n*(n-1)/2)
    g = nx.Graph()
    if m > all_edges_possible:
        print("number of edges is not possible!")
        return g
    for i in range(n):
        g.add_node(i)
    all_edges = list(range(n**2))
    while True:
        if g.number_of_edges() == m:
            return g
        x = ra.sample(all_edges,1)
        all_edges.remove(x[0])
        i = x[0]//(n)
        j = x[0]%(n)
        if i == j:
            continue
        g.add_edge(i,j)
```

در ابتدا تابعی به منظور تولید گرافی مشابه با مدل دوم ارائه شده در بخش توضیحات پروژه تدوین و به پروژه اضافه کرده‌ایم که گرافی را به صورت تصادفی و با دانستن تعداد یال‌ها و تعداد رئوس تشکیل داده و به‌عنوان خروجی به کاربر باز می‌گرداند. در آرایه‌ای به اندازه n^2 به نوعی تمام یال‌های ممکن در یک گراف nراسی را داریم، سپس در یک حلقه دائما برقرار تا زمانی که تمام mیال درخواستی کاربر در گراف نهادینه شد ادامه می‌دهیم و در هر گام یک عدد از آرایه‌ای که توضیح دادیم به صورت تصادفی گزین کرده و با دستورالعمل‌های برگرفته از درس ساختارهای داده جایگاه آن عدد را در یک آرایه دو بعدی که پیدا می‌کنیم به طوری‌که یک یال میان راس i و j وجود دارد درصورتی که در آن آرایه دوبعدی فرضی در خانه (i,j) عدد یک وجود داشته باشد ولی مجدد تاکید می‌شود که در آرایه‌ای که ما به گزینش عدد تصادفی می‌پردازیم ما صرفا یک بعد از n^2 عدد داریم. مثلا 83 در آرایه اول اگر انتخاب شد یعنی ما باید میان راس 8 و 3 یک یال برقرار کنیم. آن عدد تصادفی را از لیست اولیه حذف می‌کنیم تا گراف ساده بماند (اعدادی مثل 44 و غیره هم بیانگر وجود یک یال به خود یا همان طوقه هستند که آن ها همدر لیست یال ها قرار نمی‌گیرند.)

سپس به صورت تصادفی به کمک تابع بالا یک گراف تهیه شده است به شکل زیر که در ادامه یکسری گزارش‌ها را مطابق گراف مرسوم به گزارش الصاق گردیده است.

```
g = generate_random_graph(15,20)
nx.draw_networkx(g)
```



سپس با کمک گرفتن از توابعی که گراف در این محیط به ما ارائه کرده است مجموعه درجه رئوس و البته مجموع درجات رئوس گراف تولید شده را نمایش داده ایم. دقت شود که مجموع درجات رئوس دوبرابر تعداد یالهای گراف است چرا که هر یال را یکبار به ازای گره اول یال و یکبار به ازای گره دوم یال در محاسبات خود دخالت داده‌ایم.

لازم به ذکر است که این بخش به منظور آشنایی با توابع مورد استفاده در بخش های بعدی در پروژه قرار گرفته است ولی در خواسته پایانی از این تابع بهره‌مند خواهیم شد.

```python
def avg_edge_net(iterate ,  n ,  p):
    summ = 0
    bigNum = list()
    avg_of_degrees = list()
    variance_of_degrees = list()
    for i in range (iterate):
        g = nx.erdos_renyi_graph(n,p)
        degrees = list(d for n, d in g.degree())
        sum_of_this_iter = sum(degrees)
        avg_of_degrees.append(sum_of_this_iter/n)
        variance_of_degrees.append(np.var(degrees))
        summ = summ + (sum_of_this_iter/2)
        bigNum.append(summ/(i+1))
    return bigNum,avg_of_degrees,variance_of_degrees
```

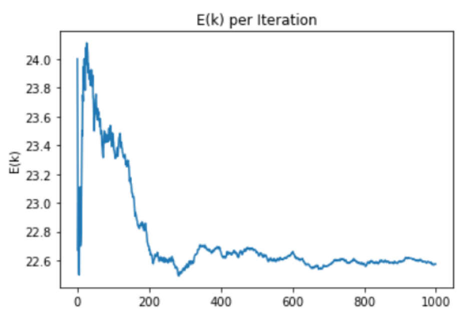
در ادامه قصد دخیل کردن قضیه اعداد بزرگ را در پروژه داریم، به همین منظور تابعی طراحی کرده‌ایم که به تعداد دفعات ارائه شده توسط کاربر میان رئوس گرافی با n گره، با احتمال p یال قرار می‌دهد، در این تابع به ازای هر مرحله به کمک تابعی که از پکیج networkx به ما معرفی گردید یک گراف اردوش-رنی می‌سازیم و سپس خواسته‌های بخش اول پروژه را به ازای هر گراف در متغیرهایی معین ثبت می‌کنیم. متغیر bigNum در هر iteration میانگین تعداد یالهای گراف‌ها (که در آن مرحله تاiterate هستند) را دربر دارد که تعداد یالها را به کمک نصف کردن مجموع درجات گره‌های مندرج در گراف به دست آورده‌ایم، واریانس و میانگین درجات را نیز در متغیرهای دیگر به ثبت رسانده‌ایم.

در ادامه مقداردهی‌های متفاوتی را به کمک این تابع بررسی کرده‌ایم که بعد از قرارگیری تصاویر به تحلیل تفاوت‌ها می‌پردازیم.

```python
iterate = 1000
bigNum,avg_of_degrees,variance_of_degrees = avg_edge_net(iterate, 10 , 0.5)
plt.plot(bigNum)
print("average of degrees : {}".format(sum(avg_of_degrees)/iterate))
print("variance of degrees : {}".format(sum(variance_of_degrees)/iterate))
plt.title('E(k) per Iteration')
plt.xlabel('Iterations')
plt.ylabel('E(k)')
```

```
average of degrees : 4.51499999999999
variance of degrees : 1.8067200000000012

Text(0, 0.5, 'E(k)')
```



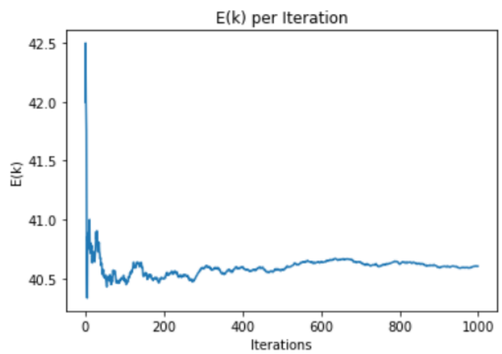
بار اول 300 مرتبه گرافی با 10 گره و احتمال قرارگیری یال 0.5 ساخته ایم و متغیر bigNum که متوسط تعداد یال‌های گرافها را مرحله به مرحله نشان می‌دهد رسم کرده ایم می‌بینیم که با گذر زمان متوسط تعداد یال‌ها به عددی بین 22.5 و 23 مایل است که با کمک فرمول ارائه شده در خواسته اول به کمک مقادیر ارائه شده در این مثال و حتی به صورت شهودی نیز می‌توان به تصدیق این مقدار مهر تایید داد. یک گراف 10 گره‌ای حداکثر 45 یال می‌تواند بپذیرد و اگر احتمال وجود یال میان دو گره را 0.5 تعیین کنیم، انتظار می‌رود تعداد یال ها در رنج 22.5 باشد البته با یک تلرانس که به اصطلاح فاصله اطمینان نیز نامیده می‌شود. به نوعی با امیدریاضی وجود یال که np است متناظر است.

در آزمایش دوم تعداد دفعات ساخت گراف را تغییر نمی‌دهیم و متغیر p را مقداری بیشتر تعیین می‌کنیم.

```
iterate = 1000
bigNum,avg_of_degrees,variance_of_degrees = avg_edge_net(iterate, 10 , 0.9)
plt.plot(bigNum)
print("average of degrees : {}".format(sum(avg_of_degrees)/iterate))
print("variance of degrees : {}".format(sum(variance_of_degrees)/iterate))
plt.title('E(k) per Iteration')
plt.xlabel('Iterations')
plt.ylabel('E(k)')
```

```
average of degrees : 8.1206
variance of degrees : 0.636880000000001

Text(0, 0.5, 'E(k)')
```

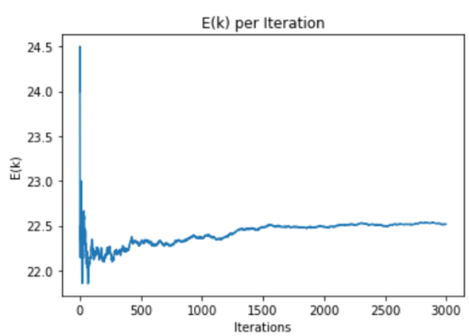


حال دفعات ساخت گراف را بیشتر کرده و می‌بینیم که واریانس کمتر می‌شود.

```
iterate = 3000
bigNum,avg_of_degrees,variance_of_degrees = avg_edge_net(iterate, 10 , 0.5)
plt.plot(bigNum)
print("average of degrees : {}".format(sum(avg_of_degrees)/iterate))
print("variance of degrees : {}".format(sum(variance_of_degrees)/iterate))
plt.title('E(k) per Iteration')
plt.xlabel('Iterations')
plt.ylabel('E(k)')
```

```
average of degrees : 4.503400000000007
variance of degrees : 1.7784800000000007

Text(0, 0.5, 'E(k)')
```



خواسته دوم

در این بخش ابتدا تابعی داریم که برای یک گراف توزیع درجه ها را محاسبه می‌کند و n_k/n را به درخواست صورت پروژه محاسبه می‌کند.

```
def degree_dist(g):
    degrees = list(d for n, d in g.degree())
    dd = Counter(degrees)
#     print(dd);
    for key in dd:
#         print(dd[key])
        dd[key] = dd[key]/n
    return dd
```

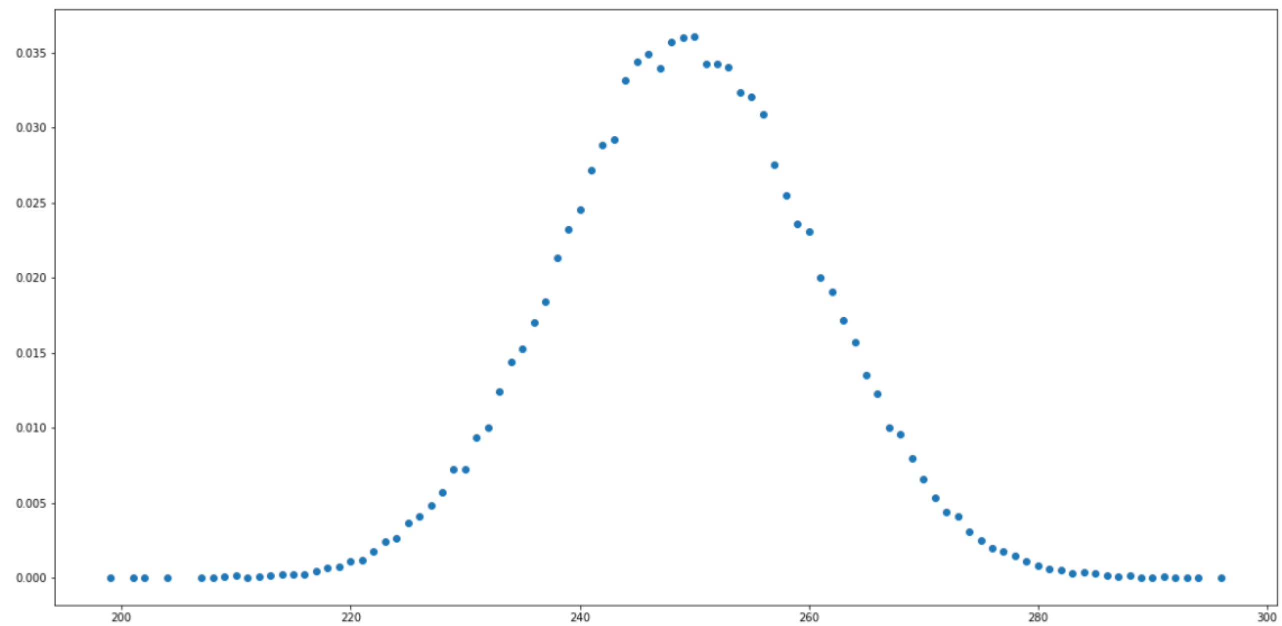
حال به هدف اصلی سوال پرداختیم که توزیع درجه ها در iterate تا گراف را بایکدیگر تجمیع کرده و نسبت می‌گیرد تا هربار توزیع درجه آن گراف‌هایی که تا مرحله i ساخته شده‌اند را داشته باشیم. نمایش دادن این متوسط را نیز در ادامه داریم و با تغییر مقادیر iterate و n بنابر قاعده اعداد بزرگ به تحلیل نتایج می‌پردازیم.

```
def degree_dist_LLN(iterate,n,p):
    counter = Counter();
    for i in range(iterate):
        g = nx.erdos_renyi_graph(n,p)
        counter = counter + degree_dist(g)
    for key in counter:
        counter[key] = counter[key]/iterate
    return counter
```

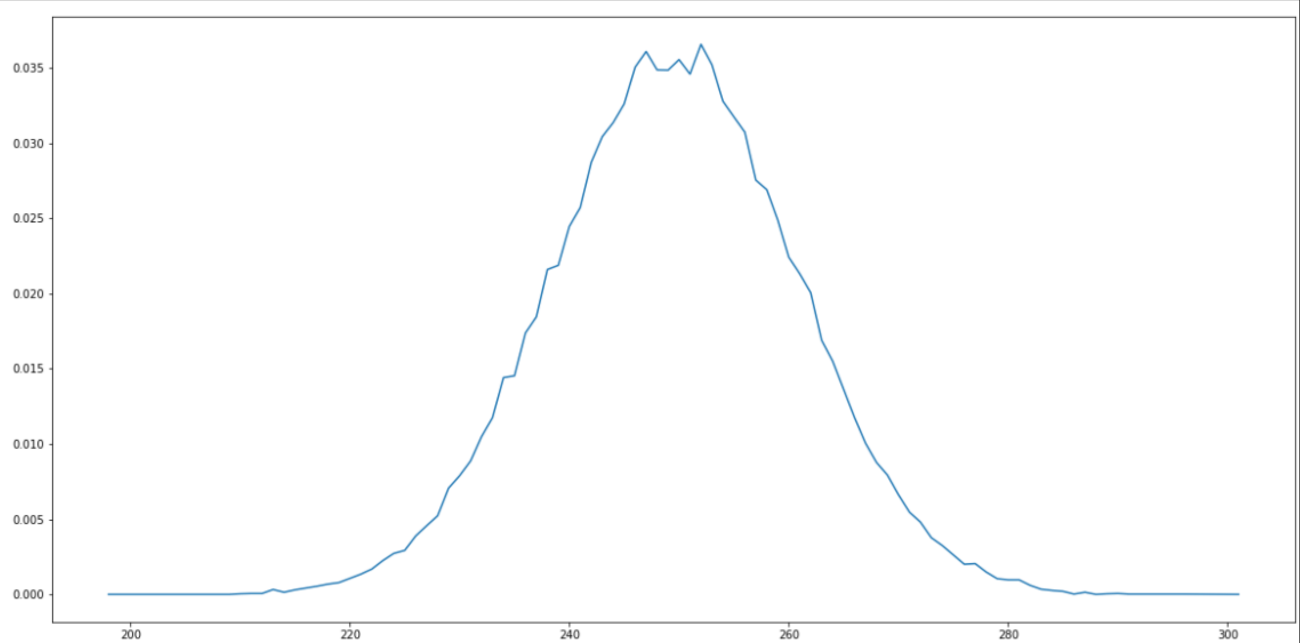
البته ایرادی که در اینجا مانع پیشروی و افزایش دقت و صحت اطلاعات مندرج در گزارش ما گردید ابهام در بحث تفاوت مطلب توزیع تصادفی گسسته و پیوسته بود و برداشت ما چنین است که درحالت گسسته که بصورت Scatter به نمایش گذاشته شده است توزیه پوآسن و درحالتی که پیوسته plot کرده‌ایم توزیع نرمال را داریم؛ بدین روی این بخش را پیش برده‌ایم.

```
iterate = 100
n = 500
c = degree_dist_LLN(iterate,n,0.5)
```

```
plt.figure(figsize=(20,10))
plt.scatter(c.keys(),c.values())
plt.show()
```




```
c = sorted(c.items())
X, Y = map(list, zip(*c))
plt.figure(figsize=(20,10))
plt.plot(X,Y)
plt.show()
```

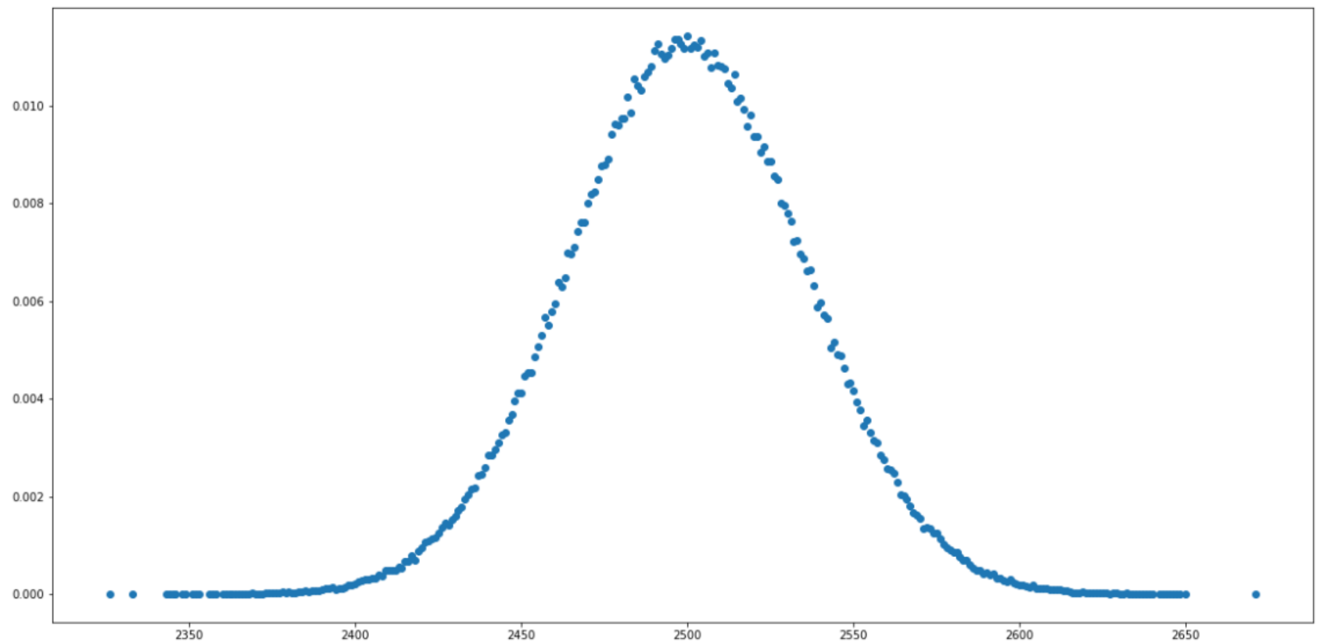


تصویر دوم توزیع پیوسته از همان مقادیر تصویر اول را به نمایش می‌گذارد.

در ادامه دریافتیم که با افزایش n نمودار نقطه‌ای نیز به توزیع نرمال مایل می‌شود.

```
iterate = 100
n = 5000
c = degree_dist_LLN(iterate,n,0.5)
```

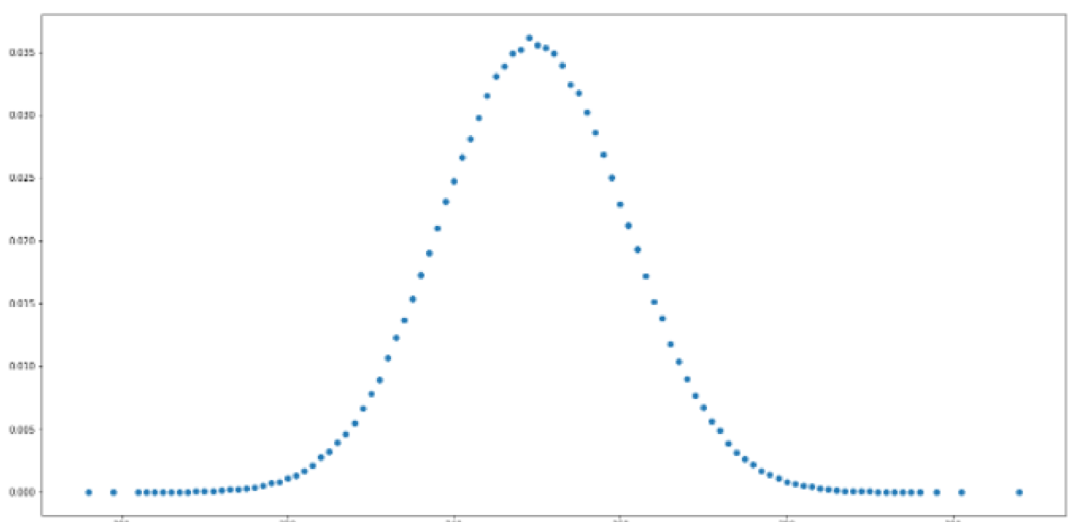
```
plt.figure(figsize=(20,10))
plt.scatter(c.keys(),c.values())
plt.show()
```



حال n را همان مقدار قبلی تعیین نموده و مقدار iteration را افزایش می‌دهیم.

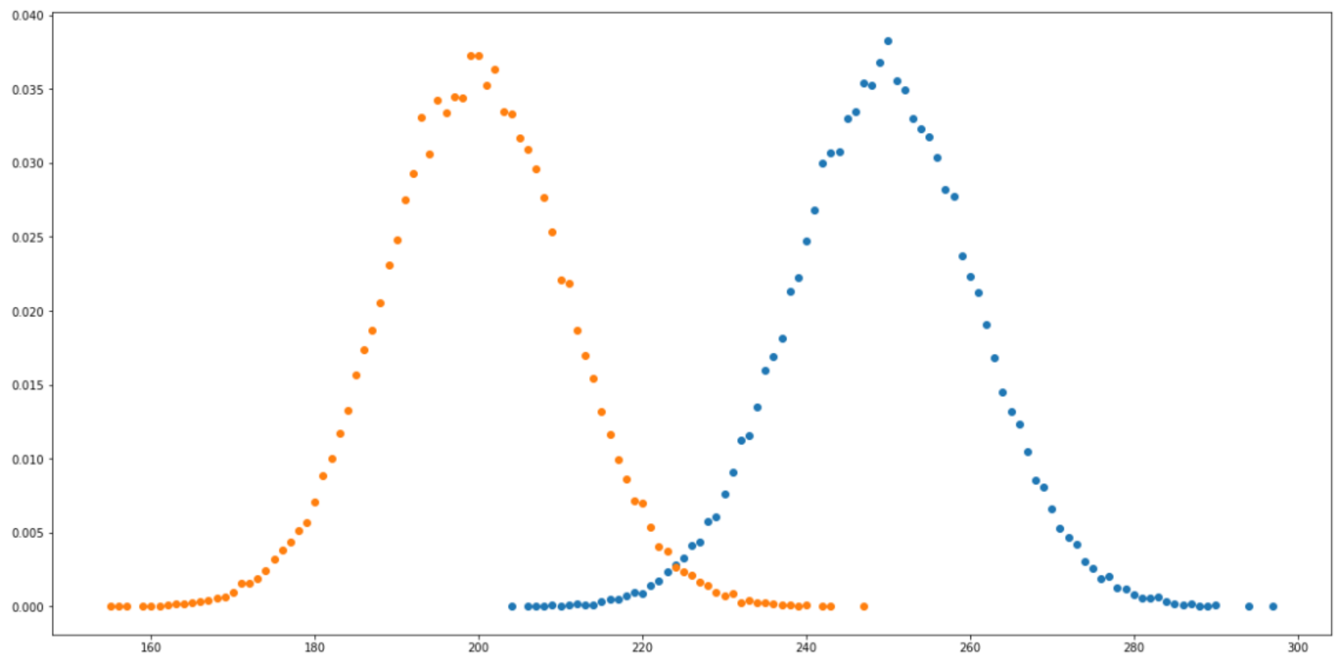
```
In [131]: iterate = 1000
          n = 500
          c = degree_dist_LLN(iterate,n,0.5)
```

```
In [132]: plt.figure(figsize=(20,10))
          plt.scatter(c.keys(),c.values())
          plt.show()
```

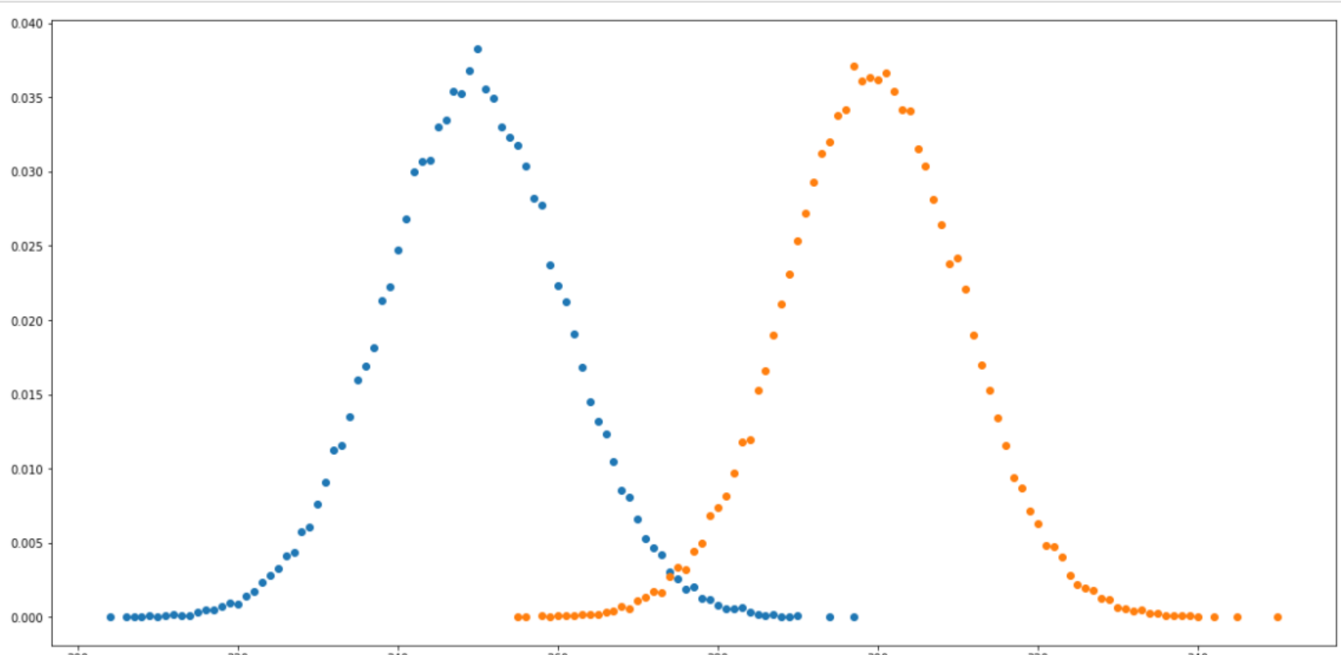


در ادامه این خواسته ما با تغییر دادن p و ثابت نگهداشتن سایر نقادیر نسبت به اولین مرحله(که در نمودارها با نقاط آبی رنگ حضور دارند)، بحث چولگی را به نمایش گذاشته ایم، وقتی احتمال کمتر را قرار می‌دهیم شاهد چولگی راست می‌شویم و وقتی احتمال را از حالت عادی که 0.5 بود بیشتر تعیین می‌کنیم به چولگی منفی می‌رسیم.

```
C = degree_dist_LLN(iterate,n,0.4)
plt.figure(figsize=(20,10))
plt.scatter(c.keys(),c.values())
plt.scatter(C.keys(),C.values())
plt.show()
```




```
C = degree_dist_LLN(iterate,n,0.6)
plt.figure(figsize=(20,10))
plt.scatter(c.keys(),c.values())
plt.scatter(C.keys(),C.values())
plt.show()
```



در قسمت بعد به کمک جستجو تقریبی برای k! براساس تقریب استرلینگ پیدا کردیم که بصورت زیر بود و پیاده سازی شده آن را در ادامه بهمراه دو تست آورده‌ایم.

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

```
k = 60
n_fact = np.sqrt(np.pi * 2 * k) * ((k/np.e)**k)
n_fact
```

```
8.3094383149767e+81
```

```
k = 5
n_fact = np.sqrt(np.pi * 2 * k) * ((k/np.e)**k)
n_fact
```

```
118.0191679575901
```

بدلیل حضور k! در مخرج محاسبات احتمال توزیع هاب‌ها کم می‌شود. توضیحی در نوتبوک هم قرار گرفته است که به تحلیل نمودار مرسوم در این قسمت می‌پردازد.

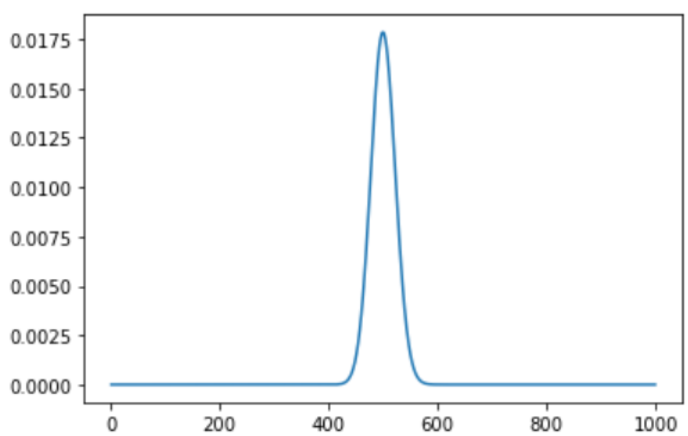
```
def pois(k , avg):
    return np.exp(-avg)/(np.sqrt(2*np.pi*k))*((avg*np.e)/k)**k
```

همانطور که در پایین دیده میشود به سرعت به صفر نزدیک میشود و هرچه مقادیر کا بیشتر میشود سریع تر به صفر نزدیک میشود و اصطلاحا دم نازک است.

```
k = np.arange(1000)
plt.plot(pois(k,500))
```

```
C:\Users\asus\AppData\Local\Temp/ipykernel_2244/2783104582.py:2: RuntimeWarning: divide by ze
  return np.exp(-avg)/(np.sqrt(2*np.pi*k))*((avg*np.e)/k)**k
```

```
[<matplotlib.lines.Line2D at 0x149e81e03d0>]
```



خواسته سوم

در این بخش رویه ما به اینصورت است که همانند بخش اول به ساخت یک گراف n گره‌ای می‌پردازیم و هرسری یک یال به آن می‌افزاییم. با این تفاوت که هربار طول بزرگترین خوشه را در دست خواهیم داشت و بررسی می‌کنیم که برابر با n هست یا خیر، اگر برابر با n شده باشد که یعنی گره‌های گراف تماما به هم متصل شده‌اند و گراف به اصطلاح همبند شده است حال ما این تعداد یال ها را برمی‌گردانیم و البته یک رسم نمودار هم مطابق با خواسته صورت پروژه رسم می‌کنیم که همانطور که در مثال اجرا شده مشاهده می‌کنید در زمانی که یال 405م وارد گراف می‌شود گراف همبند می‌شود.

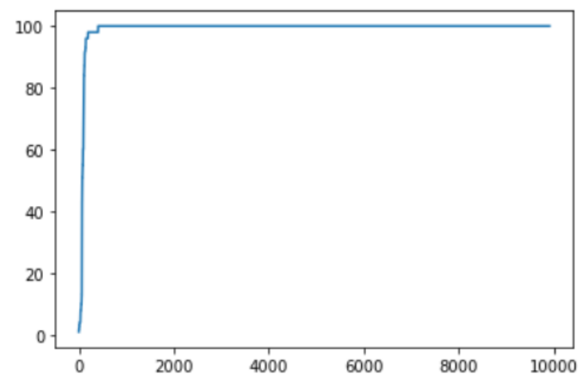
```
def generate_graph(n):
    all_edges_possible = int(n*(n-1)/2)
    g = nx.Graph()
    for i in range(n):
        g.add_node(i)
    li = list(range(int(n**2)))
    connected = 99999999999999
    p = list()
    while True:
        if g.number_of_edges() == all_edges_possible:
            return p , connected
        h = list(nx.connected_components(g))
        for i in range(len(h)):
              h[i] = len(h[i])

        p.append(np.max(h))
        if np.max(h) == n:
            connected = min (int(n**2) - len(li) , connected)

        x = ra.sample(li,1)
        li.remove(x[0])
        i = x[0]//(n)
        j = x[0]%(n)
        if i == j:
            continue
        g.add_edge(i,j)
```

```
lis , co = generate_graph(100)
plt.plot(lis)
print(co)
```

```
405
```

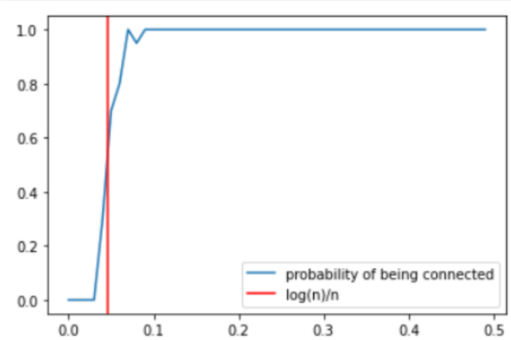


اطلاعاتی که دیدید یک بخش اضافه جهت درک بهتر این بخش بود که انجام دادیم، در ادامه خواسته اصلی را براورده کرده‌ایم و تابعی داریم که تعداد دفعات ساخت یک گراف n راسی را با احتمال ایجاد یال p دریافت می‌کند و سپس مقداری را که برمی‌گرداند، متوسط تعداد گراف‌های همبند (که به صورت تصادفی ساخته شده‌اند) می‌باشد. سپس به ازای احتمال‌های متفاوت این تابع را اجرا کرده و می‌بینیم از یک p که همان P_c طبق صورت پروژه است گراف ها همبند می‌شوند، حال پس از چند مرحله اجرا و پردازش این روش و میانگین گیری و رسم نمودار می‌بینیم که در نقطه‌ای که تقریبا نزدیک 0.04 است P_c قرار گرفته است، همچنین مقدار log(n)/n را هم چاپ کرده‌ایم و می‌بینیم که این دو مقدار تقریبا معادل هم شده اند.

```
def check_log(iterate ,n, p):
    number_of_connnected = 0
    for i in range(iterate):
        g = nx.erdos_renyi_graph(n,p)
        if nx.is_connected(g):
            number_of_connnected = number_of_connnected+1
    return number_of_connnected/iterate
```

```
li = dict()
n = 100
iterate = 20
for i in np.arange(0,0.5,0.01):
    li[i] = (check_log(iterate ,n, i))
```

```
plt.plot(li.keys(),li.values(),label='probability of being connected')
plt.axvline(np.log(n)/n,label='log(n)/n',color='red')
plt.legend()
plt.show()
print(np.log(n)/n)
```




```
0.04605170185988092
```

دومین خواسته دوم

$$\sum_{l=0}^{n}(-1)^{l}\binom{n}{l}\binom{m-np-lq+n-1}{n-1}$$

برای آوردن توجیه و اثبات برای رابطه بالا فرض میکنیم که n ظرف(پرانتز ها) داریم و از هر ظرف باید تعدادی توپ برداریم(توانی از X) و باید به این نکته نیز دقت کنیم که از هر ظرف حداکثر میتوان q-1 توپ برداشت. حال قضیه ما به یک سوال ساده اصل عدم شمول تبدیل میشود.در اینجا تعداد توپها m-np می باشد و تعداد ظرف ها نیز n عدد می باشد.

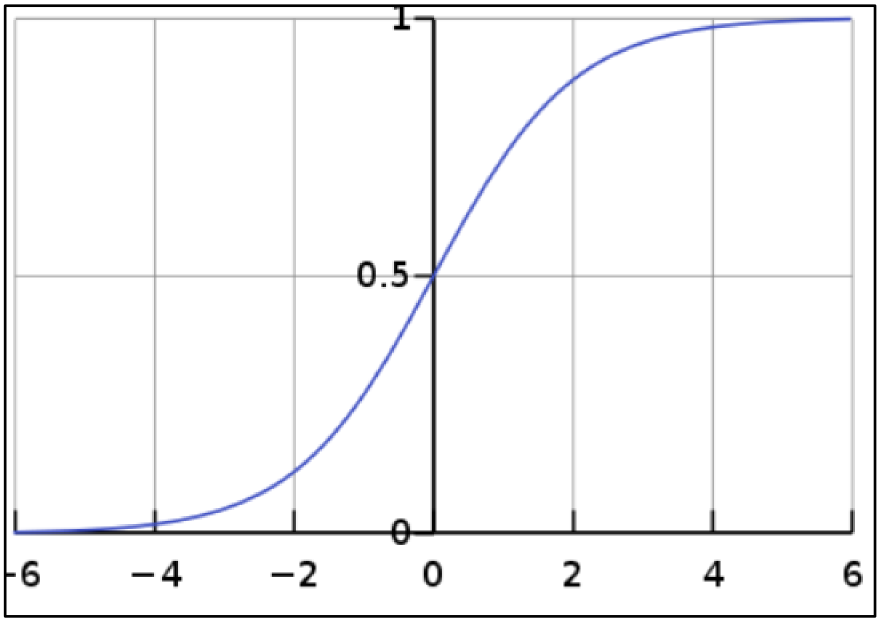
البته در صورت پروژه به جای l به اشتباه n نوشته شده بود.

برای قسمت آخر از ما خواسته شد که مدل fitness را با الگوریتمی که داده شده پیاده سازی کنیم که در پایین کد آن آورده شده.

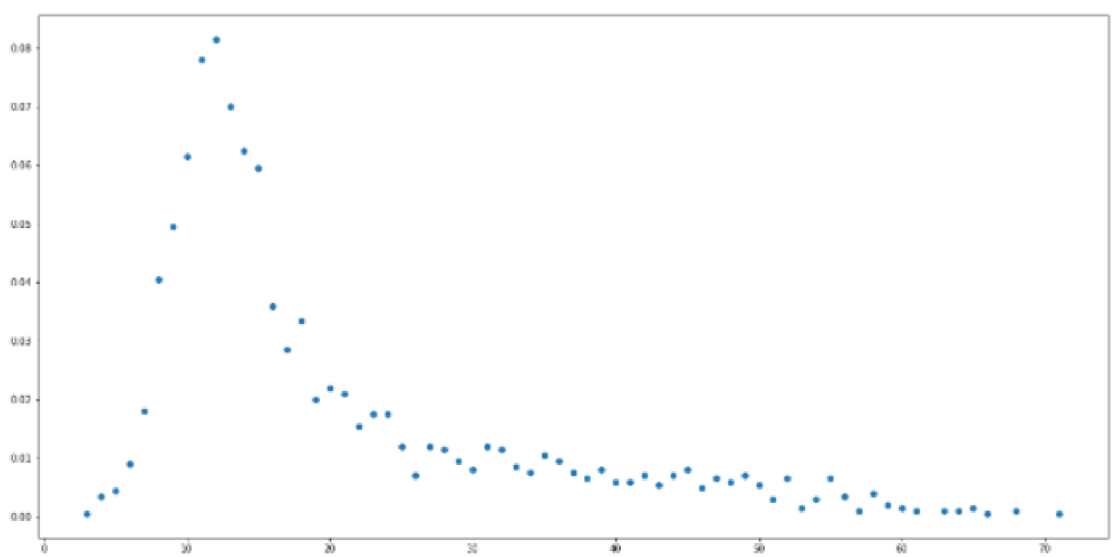
```
def fitness_model(g , n):
    beta = np.random.normal(size=n)
    for k in range(n**2):
        edges = list(g.edges)
        edgeIndex = ra.randint(0 , len(edges)-1)
        i = edges[edgeIndex][0]
        j = edges[edgeIndex][1]
        while g.degree(i) == n-1:
            edgeIndex = ra.randint(0 , len(edges)-1)
            i = edges[edgeIndex][0]
            j = edges[edgeIndex][1]
        m = i
        while i == m or (m in g.adj[i]):
            m = ra.randint(0 , n-1)
#           print(m)
        p = ra.uniform(0 , 1)
        if(p < pi(g.degree(m) , beta[m] , n)):
            g.remove_edge(i , j)
            g.add_edge(i,m)
    return g
```

برای احتمال داده شده به هر نود باید عدد داده شده را ($(k_m+1)\beta_m$)به عددی بین 0 و 1 تبدیل کنیم برای این کار این عدد را به n تقسیم کرده و حالا عدد ما فقط به بتا بستگی دارد و چون بتا از توزیع نرمال استاندارد آمده است مقدارش به احتمال 95 درصد بین -3و3 است پس اگر از این مقدار sigmoid بگیریم عددی بین 0 و 1 برمیگرداند و چون به احتمال بالایی بین -3 و 3 است مقدار برگشتی آن عددی تقریبا متفاوت برای این مقادیر میشود.

تابع sigmoid :




```
n = 100
m = 1000
c = degree_dist_LLN(20,n , m)
plt.figure(figsize=(20,10))
plt.scatter(c.keys(),c.values())
plt.show()
```



توزیع درجه در این مدل که در بالا آمده است دیگر از توزیع هایی که قبل از این گفته ایم تبعیت نمی‌کند و توزیعی اصطلاحا دم کلفت دارد و تعداد گره های با درجه بالا تر بیشتر وجود دارد و هاب ها نیز در این مدل دیده می‌شوند.