Xuan(James) Zhai - CS 5350 - Final Project (Part A)

Note: Part A is written in python, so the report for that part is written with Jupyter Notebook.

1: Making a graph in python

In [1]:
```
from os import close
import random
import time
import matplotlib.pyplot as plt


# Reference: https://www.pythonpool.com/adjacency-list-python/
adj_list = {}


def ClearList():                        # Delete an adj list
    adj_list.clear()


def InitList(v):
    for i in range(v):
        temp = []
        adj_list[i] = temp


def add_edge(node1, node2, weight):
    adj_list[node1].append([node2,weight])
    adj_list[node2].append([node1,weight])
```

In [6]:
```
def PrintOutput(isToaFile, filename):              # isToaFile = 0 means print to the
    if isToaFile:
        file = open(filename,"w")
        file.write(str(len(adj_list)) + "\n")
    else:
        print(str(len(adj_list)))

    initialindex = 1 + len(adj_list)
    for i in adj_list:
        if isToaFile:
            file.write(str(initialindex) + "\n")
        else:
            print(str(initialindex))
        initialindex = initialindex + len(adj_list[i])

    for i in adj_list:
        for j in adj_list[i]:
            if isToaFile:
                file.write(str(j[0]) + "\n")
            else:
                print(str(j[0]))

    if isToaFile:
        file.close()
```

1.1: Analysis

The adjacency list is a python dictionary (or a hash table). It has a group of keys which are the head node, those keys also refer to the vertices in a graph. For each key, there will be a list of pairs which are nodes and weight, and they are the nodes/vertices that the current one connected to.

The InitList function will add all the vertices to the graph, and the time complexity for that is Θ(n)

To add an edge to the graph, it will use the add_edge function. This function will push back the nodes that they connected to the list in the hash value.

The time compexity for adding an edge is Θ(1). That's because finding a key in a hash table is Θ(1). Add pushing back an item to a list is also Θ(1). Here the code will not check whether that edge is already connected, since if we add that the time compexity will be O(n). As a result, we won't add duplicate edge when we build the graph.

2: Making a Complete Graph

2.1: Source Code

In [7]:
```
def CompleteGraph(v):
    InitList(v)
    for i in range(v):
        for j in range(i,v):
            if(i != j):
                add_edge(i,j,1)
```

In [8]:
```
CompleteGraph(5)
PrintOutput(False,"file1.txt")
```

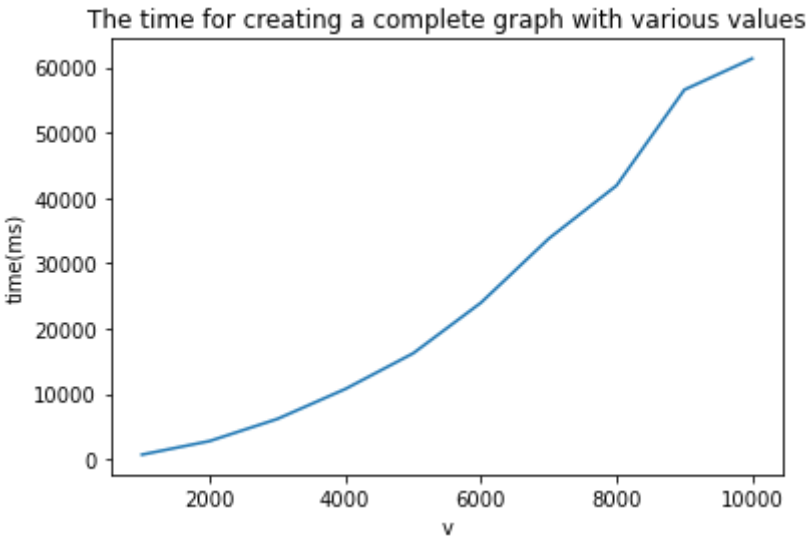
```
5
6
10
14
18
22
1
2
3
4
0
2
3
4
0
1
3
4
0
1
2
4
0
1
2
3
```

2.2: Run the code multiple times

In [10]:
```
time1 = []
for i in range(1000,11000,1000):
    print(i)
    start = time.time()
    CompleteGraph(i)
    end = time.time()
    duration = (end-start) * 1000
    time1.append(duration)
    ClearList()
print(time1)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
[706.0072422027588, 2789.975643157959, 6178.999662399292, 10756.002426147461, 16211.002826690674, 23958.988904953003, 33767.688035964966, 41896.99602127075, 56544.013977050781, 61315.98734855652]
```

In [11]:
```
xaxis = [1000,2000,3000,4000,5000,6000,7000,8000,9000,10000]
plt.plot(xaxis, time1)
plt.title("The time for creating a complete graph with various values")
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



v	Time(ms)
1000	706.01
2000	2789.98
3000	6189.00
4000	10756.00
5000	16211.00
6000	23958.99

v	Time(ms)
7000	33767.69
8000	41897.00
9000	56544.01
10000	61315.99

2.3: Analysis

From the plot and the table above, we can find out that the running time is close to the equation which

$$t' = (\frac{n'}{n})^2 * t$$

Therefore, it consolidates out assumption that creating a complete graph is Θ(v^2)

3: Making a Cycle

3.1: Source Code

In [12]:
```
def CycleGraph(v):
    InitList(v)
    for i in range(v-1):
        add_edge(i,i+1,1)
    add_edge(v-1,0,1)
```

In [13]:
```
CycleGraph(5)
PrintOutput(False,"file2.txt")
```

```
5
6
8
10
12
14
1
4
0
2
1
3
2
4
3
0
```

3.2: Run the code multiple times

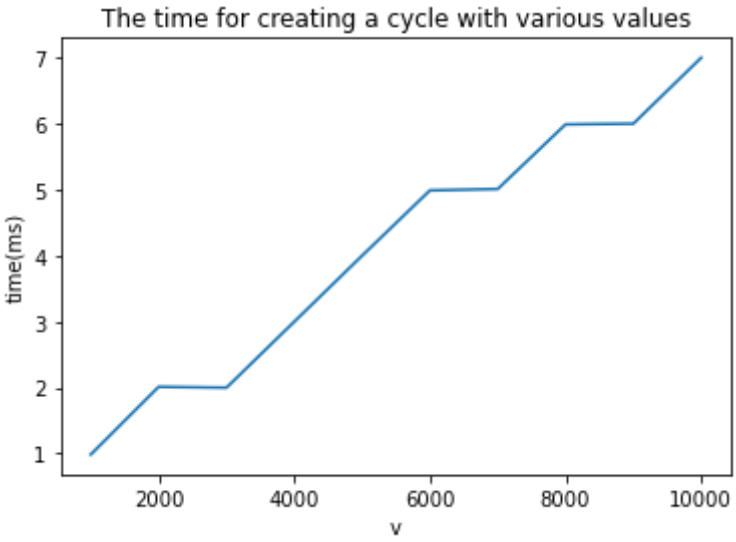
In [15]:
```
time2 = []
for i in range(1000,11000,1000):
    print(i)
    start = time.time()
    CycleGraph(i)
    end = time.time()
    duration = (end-start) * 1000
    time2.append(duration)
```

```
    ClearList()
print(time2)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
[0.9872913360595703, 2.012968063354492, 1.9998550415039062, 2.9997825622558594, 3.9999
48501586914, 4.989385604858398, 5.0106048583984375, 5.989789962768555, 6.0005187988281
25, 7.000207901000977]
```

In [16]:

```
plt.plot(xaxis, time2)
plt.title("The time for creating a cycle with various values")
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



v	Time(ms)
1000	0.99
2000	2.01
3000	2.00
4000	3.00
5000	4.00
6000	4.99
7000	5.01
8000	5.99
9000	6.00
10000	7.00

3.3: Analysis

From the plot and the table above, we can find out that although the running time is also an

increasing line, a constant growth in v will lead to a relatively constant growth in a proportional scale. Besides, the running time is close to the equation which

$$t' = (\frac{n'}{n}) * t$$

Therefore, it consolidates out assumption that creating a cycle is Θ(v)

4: Making a random graph with uniform distribution

4.1: Source Code

In [2]:
```
def RandomGraph(v,e):
    if e < 0 or e > v*(v-1)/2:
        print("Invalid number of edges")
        return

    InitList(v)
    pairlist = []
    numofedge = 0
    for i in range(v):
        for j in range(i,v):
            if(i != j):
                pairlist.append([i,j])          # Add all possible edges to a list

    while numofedge != e:
        newindex = random.randint(0,len(pairlist)-1)        # Randomly choose one edge
        add_edge(pairlist[newindex][0],pairlist[newindex][1],1)     # Add that to a gi

        temp = pairlist[newindex]
        pairlist[newindex] = pairlist[len(pairlist)-1]
        pairlist[len(pairlist)-1] = temp

        pairlist.pop()                      # Mark that visited edge
        numofedge = numofedge + 1
```

In [18]:
```
RandomGraph(5,3)
PrintOutput(False,"file1.txt")
```

```
5
6
8
8
9
10
3
4
4
0
2
0
```

Since the code will firstly find all possible edges and add it to a list, the Time compexity for filling that list is Θ(v^2). For the second part, the while loop will run e times which e is the number of edges. In the while loop, the Time complexity is Θ(1) since it will need to delete the added edge from the list. Here the code will swap the visited edge to the bottom and then delete it, so that the time complexity for deleting that edge is Θ(1)

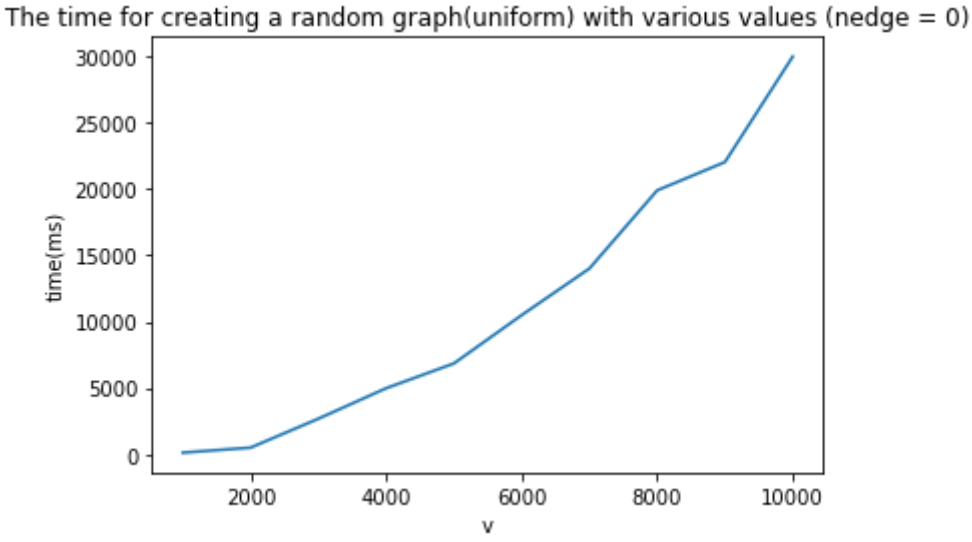
Therefore, the total time complexity is Θ(v^2+e). Here I choose to store all the possible edges to a list first because I want to make sure the randomly generated edge is not one that's already in the graph. So if e is 0, the time complexity for this algorithm is high (v^2). But if we want a randomly generated complete graph, then it won't need to worry about identifying whether that selection is in the graph of not.

4.2: Run the code multiple times (when number of edge is 0)

In [19]:
```
time3 = []
for i in range(1000,11000,1000):
    print(i)
    start = time.time()
    RandomGraph(i,0)
    end = time.time()
    duration = (end-start) * 1000
    time3.append(duration)
    ClearList()
print(time3)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
[128.01122665405273, 503.98802757263184, 2683.0146312713623, 4979.694128036499, 6846.9
99645233154, 10486.988544464111, 14003.012657165527, 19890.998363494873, 22025.0003337
8601, 29969.99979019165]
```

In [42]:
```
plt.plot(xaxis, time3)
plt.title("The time for creating a random graph(uniform) with various values (nedge =
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



v	Time(ms)
1000	128.01
2000	503.99

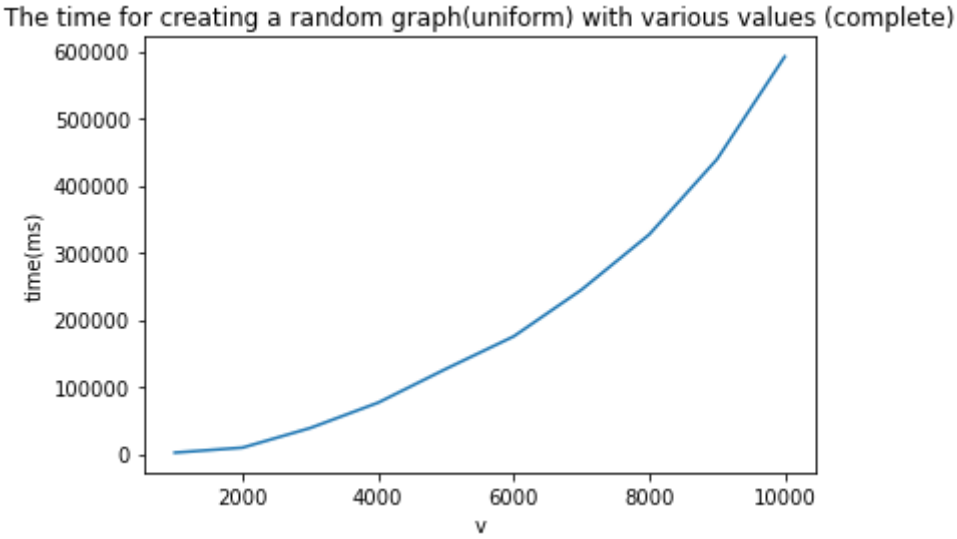
v	Time(ms)
3000	2683.01
4000	4979,69
5000	6847.00
6000	10486.99
7000	14003.01
8000	19891.00
9000	22025.00
10000	29970.00

4.3: Run the code multiple times (when it is a complete graph)

In [21]:
```
time4 = []
for i in range(1000,11000,1000):
    print(i)
    start = time.time()
    nedge = i*(i-1)/2
    RandomGraph(i,nedge)
    end = time.time()
    duration = (end-start) * 1000
    time4.append(duration)
    ClearList()
print(time4)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
[2173.999786376953, 9635.985136032104, 38891.00503921509, 76633.01110267639, 126963.56
439590454, 175354.00700569153, 244810.1842403412, 327187.00528144836, 439231.871604919
43, 592248.6681938171]
```

In [41]:
```
plt.plot(xaxis, time4)
plt.title("The time for creating a random graph(uniform) with various values (complete
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



v	Time(ms)
1000	2174.00
2000	9635.99
3000	38891.01
4000	76633.01
5000	126963.56
6000	175354.01
7000	244810.18
8000	327187.01
9000	439232.82
10000	592248.67

Analysis

From the two cases showed above, When the number of edge is 0, the plot is similar to the one in question, and data in the table also follows the equation.

$$t' = (\frac{n'}{n})^2 * t$$

When it's a complete graph, the shape of the curve is similar to the one above, but it's y-intercept is much higher.

Therefore, the time complexity is Θ(v^2+e)

5: Making a random graph with Skewed distribution

5.1: Source code

In [10]:

```
def RandomGraphS(v,e):                          # Random graph with Skewed distribution
    if e < 0 or e > v*(v-1)/2:
        print("Invalid number of edges")        # Check if the number of ver and edge are
        return

    InitList(v)
    totalpossibility = int((1+ (v-1)+(v-2)) * ((v-1)+(v-2)) /2)  # Find the total numb
```

```
    vertexpair = {}  # The key is the sum of two vertices, the value is a list of pai
    for i in range(v):                                  # Time complexity Theta(v^2)
        for j in range(i,v):
            if(i != j):
                if i+j in vertexpair.keys():
                    vertexpair[i+j].append([i,j])
                else:
                    vertexpair[i+j] = []
                    vertexpair[i+j].append([i,j])

    numofedge = 0

    while numofedge != e:
        newindex = random.randint(1,totalpossibility)       # Generate a random number
        loc = 0
        while newindex > 0:                                 # Identify which interval/k
            if loc in vertexpair and len(vertexpair[loc]) != 0:
                newindex -= (2*(v-1)-loc)
            loc += 1
        loc -= 1
        cindex = random.randint(0,len(vertexpair[loc])-1)       # Select a pair of ver
        add_edge(vertexpair[loc][cindex][0],vertexpair[loc][cindex][1],1)

        temp = vertexpair[loc][cindex]
        vertexpair[loc][cindex] = vertexpair[loc][len(vertexpair[loc])-1]
        vertexpair[loc][len(vertexpair[loc])-1] = temp
        vertexpair[loc].pop()                                   # Remove that chosen pa
        if len(vertexpair[loc]) == 0:
            totalpossibility -= (2*(v-1)-loc)     # If that key does not have a value,
        numofedge += 1
```

In [26]:

```
RandomGraphS(5,3)
PrintOutput(False,"file1.txt")
```

```
5
6
7
8
10
12
2
3
0
3
2
1
```

This algorithm uses the sum of two vertices(source and destination) for the distribution. So a pair of vertices with a smaller sum will be more likely to be chosen.

For example, for a graph with 5 vertices, the possible sum is from 1 (0,1) to 7 (3,4). The possibility of choosing 1 is 7/(1+2+3+4+5+6+7); the possibility of choosing 2 is 7/(1+...+7); the possibility of choosing 7 is 1/(1+...+7).

I firstly create a totalposibility, which is the denominator above. The last item is (v-1)+(v-2). So 1+2+3+...((v-1)+(v-2)) is (1+ (v-1)+(v-2)) *((v-1)+(v-2)) /2. Note in the formula above, the numerator is 2*(v-1)-n

Then, I create a python dictionary. The key is the sum of pair of vertices, while the value is a list of vertices. For example, vertexpair[3] will have [0,3] and [1,2]

After that, I use a while loop to go over each edge. In the while loop, the code will randomly choose a number from 1-denominator. Then it will use another while loop to find which key it correspond to. Next, it will randomly choose a pair from the list and add that edge to the graph. If so the list becomes empty, it will update the totalpossibility.

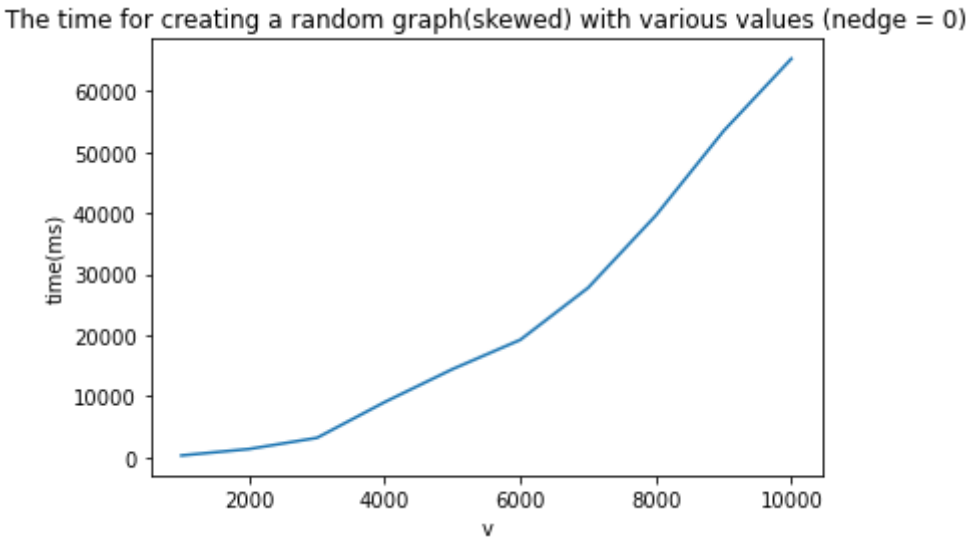
This algorithm has a time complexity Θ(v^2+v*e), or O(v^3) which is not a tight bound.

5.2: Run the code multiple times (when number of edge is 0)

In [27]:
```
time5 = []
for i in range(1000,11000,1000):
    print(i)
    start = time.time()
    RandomGraphS(i,0)
    end = time.time()
    duration = (end-start) * 1000
    time5.append(duration)
    ClearList()
print(time5)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
[318.00270080566406, 1377.9973983764648, 3202.9876708984375, 9061.015129089355, 14451.
99990272522, 19247.999668121338, 27757.00044631958, 39606.00018501282, 53427.028656005
86, 65257.46011734009]
```

In [32]:
```
plt.plot(xaxis, time5)
plt.title("The time for creating a random graph(skewed) with various values (nedge = 0
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



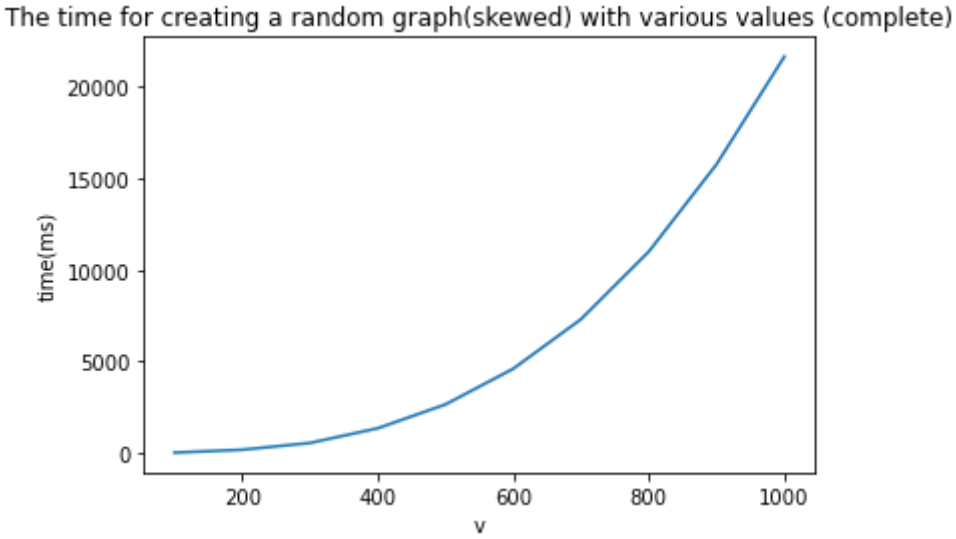
v	Time(ms)
1000	318.00
2000	1378.00
3000	3202.99
4000	9061.02
5000	14452.00
6000	19248.00
7000	27757.00
8000	39606.00
9000	53427.03
10000	65257.46

5.3: Run the code multiple times (when it is a complete graph)

In [29]:
```
time6 = []
for i in range(100,1100,100):
    print(i)
    start = time.time()
    nedge = i*(i-1)/2
    RandomGraphS(i,nedge)
    end = time.time()
    duration = (end-start) * 100
    time6.append(duration)
    ClearList()
print(time6)
```

```
100
200
300
400
500
600
700
800
900
1000
[27.80013084411621, 186.90013885498047, 554.4998168945312, 1357.706093788147, 2659.484124183655, 4608.774471282959, 7312.757349014282, 11004.433560371399, 15771.348762512207, 21647.63810634613]
```

In [33]:
```
xlaxis = [100,200,300,400,500,600,700,800,900,1000]
plt.plot(xlaxis, time6)
plt.title("The time for creating a random graph(skewed) with various values (complete)
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



v	Time(ms)
100	27.80
200	186.90
300	544.50
400	1357.71
500	2659.48
600	4608.77
700	7312.76
800	11004.43
900	15771.35
1000	21647.64

5.4: Analysis

Note: For running with complete graphs, since it's time complexity is too high, so I changed the dataset from 1000-10000 to 100-1000. That algorithm does accept 10000; I use smaller dataset is just for better analysis.

When the number edge is 0, it will just run the fist half of the code, and that time complexity should be Θ(v^2). From the result we can find out that, when we double "v", the time is multiply by something around 4. Therefore, it does follow our assumption.

When it's creating a complete graph, the time complexity should be Θ(v^2+v*e) since the second part will add "e" edges to the graph; adding each edge will needs to do loop in the dictionary to find the number (upper bound is v/2).

When we double the v, the time is multiple by around 8. Since it's a complete graph, $e = \frac{n*(n-1)}{2}$, so v^2+v*e is close to its upper bound v^3. For Θ(v^3), we know $ t2 = \frac{n2}{n1}^3t1$. Therefore, this result does corrspond to our assumption.

6: Making a random graph with Gaussian distribution

6.1: Source Code

In [12]:

```
def RandomGraphG(v,e):
    if e < 0 or e > v*(v-1)/2:
        print("Invalid number of edges")
        return

    InitList(v)
    totalpossibility = (1+(v-2))*(v-2) + (v-1)     # 1+2+3+...(v-3)+(v-2)+(v-3)+...+2+1

    vertexpair = {}
    for i in range(v):
        for j in range(i,v):
            if(i != j):
                if i+j in vertexpair.keys():
                    vertexpair[i+j].append([i,j])
                else:
                    vertexpair[i+j] = []
                    vertexpair[i+j].append([i,j])

    numofedge = 0

    while numofedge != e:
        newindex = random.randint(1,totalpossibility)
        loc = 0
        while newindex > 0:
            if loc in vertexpair and len(vertexpair[loc]) != 0:
                if loc < v:                 # If at the first half
                    newindex -= (loc)       # the nominator is equal to the sum
                else:       # If at the second half, it's eual to (v-1)*2-nominator
                    newindex -= ((v-1)*2-loc)
            loc += 1
        loc -= 1
        cindex = random.randint(0,len(vertexpair[loc])-1)
        add_edge(vertexpair[loc][cindex][0],vertexpair[loc][cindex][1],1)

        temp = vertexpair[loc][cindex]
        vertexpair[loc][cindex] = vertexpair[loc][len(vertexpair[loc])-1]
        vertexpair[loc][len(vertexpair[loc])-1] = temp
        vertexpair[loc].pop()
        if len(vertexpair[loc]) == 0:
            if loc < v:
                totalpossibility -= (loc)
            else:
                totalpossibility -= ((v-1)*2-loc)
        numofedge += 1
```

In [37]:

```
RandomGraphG(5,3)
PrintOutput(False,"file1.txt")
```

```
5
6
7
7
9
11
3
4
3
0
2
2
```

For this algorithm with Gaussian Distribution, the probability will not be in decreasing order.

For example, with 5 vertices, the maximum sum is 7(3,4), so the probability for sum 1 is 1/(1+2+3+4+3+2+1), for sum 2 it is 2/(1+2+3+4+3+2+1), for sum 7 it is 1/(1+2+3+4+3+2+1). Here the denominator is 1+2+...+(v-1)+(v-2)+...+2+1, which is (1+(v-2))*(v-2) + (v-1). For nominator, the first half of the sums will be equal to its index, the second half will be (v-1)*2- index.

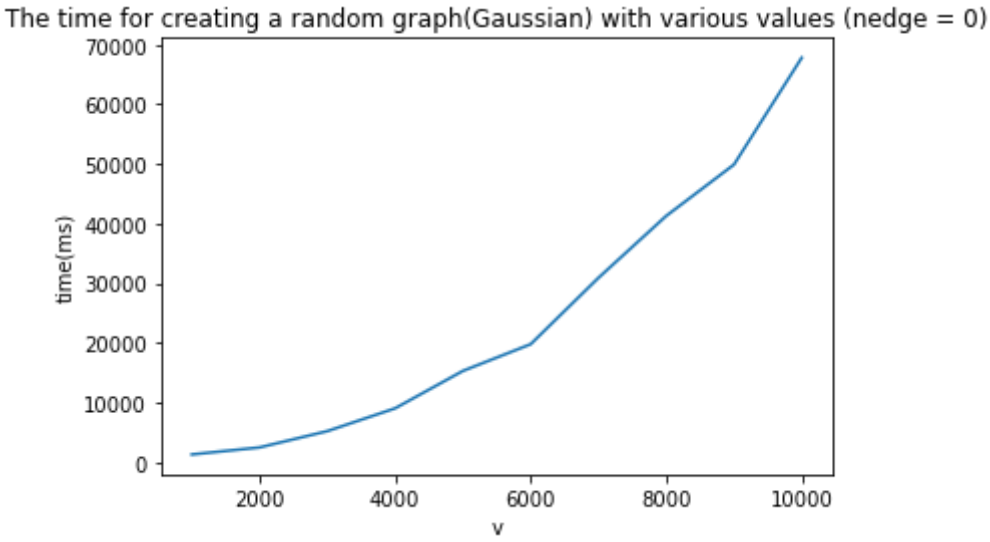
The rest part will be the same as the algorithm with Skewed Distribution.

6.2: Run the code multiple times (when number of edge is 0)

In [38]:
```
time7 = []
for i in range(1000,11000,1000):
    print(i)
    start = time.time()
    RandomGraphG(i,0)
    end = time.time()
    duration = (end-start) * 1000
    time7.append(duration)
    ClearList()
print(time7)
```

```
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
[1328.017234802246, 2506.9851875305176, 5240.015268325806, 9059.99755859375, 15326.986
55128479, 19803.040981292725, 30913.94329071045, 41285.990953445435, 49886.5704536438,
67787.64224052429]
```

In [43]:
```
plt.plot(xaxis, time7)
plt.title("The time for creating a random graph(Gaussian) with various values (nedge =
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



v	Time(ms)
1000	1328.02

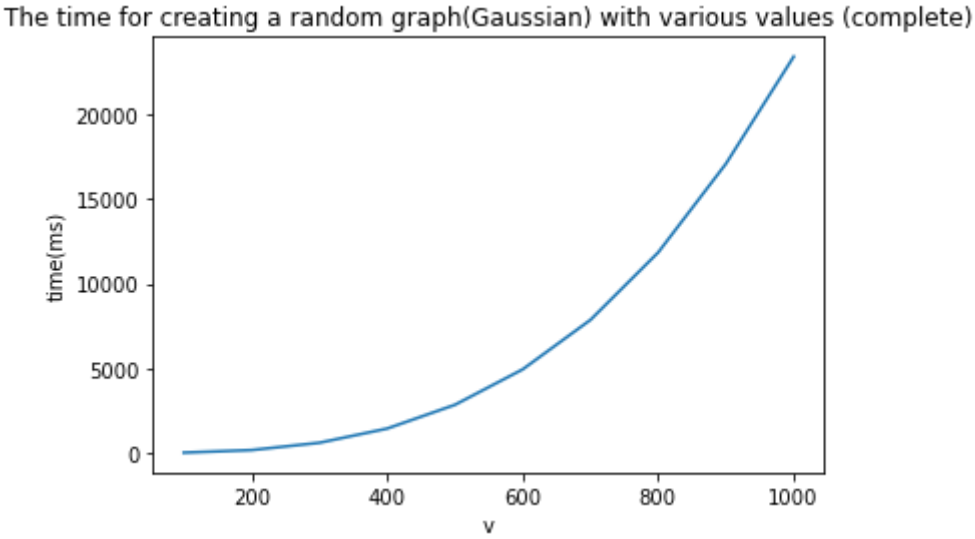
v	Time(ms)
2000	2507.99
3000	5240.02
4000	9060.00
5000	15326.99
6000	19803.04
7000	30913.94
8000	41285.99
9000	49886.57
10000	67787.64

6.3: Run the code multiple times (when it is a complete graph)

In [39]:
```
time8 = []
for i in range(100,1100,100):
    print(i)
    start = time.time()
    nedge = i*(i-1)/2
    RandomGraphG(i,nedge)
    end = time.time()
    duration = (end-start) * 100
    time8.append(duration)
    ClearList()
print(time8)
```

```
100
200
300
400
500
600
700
800
900
1000
[22.635364532470703, 174.42429065704346, 605.5257558822632, 1444.6000337600708, 2847.0
9951877594, 4942.592263221741, 7866.1561489105225, 11853.447723388672, 17096.655392646
79, 23423.483967781067]
```

In [40]:
```
xlaxis = [100,200,300,400,500,600,700,800,900,1000]
plt.plot(xlaxis, time8)
plt.title("The time for creating a random graph(Gaussian) with various values (complet
plt.xlabel("v")
plt.ylabel("time(ms)")
plt.show()
```



v	Time(ms)
100	22.63
200	174.424
300	605.52
400	1444.60
500	2847.10
600	4942.59
700	7866.16
800	11853.45
900	17096.66
1000	23423.48

6.4: Analysis

Since the only things we've changed are the totalpossibility and the weight for choosing each sum, and they will not affect the total time complexity. We can also find that on the result we collected above.

Therefore, the time complexity for this algorithm is still Θ(v^2+v*e) (or O(v^3))as a non-tight bound)

7: Conflict/Number of degree Analysis for random graph

7.1 Random Graph with Uniform Distribution

Here I use a function to find the number of vertices on the graph, and the average number of degrees for each vertex. Moreover, for one edge, since I used the sum of source+destination as the element for skewed and Gaussian distribution, the function will also print out for each sum, its average number of edges. I'll run that 100 times and take the average to get a better result.I will use 15 vertices and 100 edge just for nice printout, but for an accurate analysis I should use more vertices and edges.

In [5]:
```
def FindConflicts(dis):
```

```
    vnconflit = [0]*15
    estable = {}
    for i in range(100):
        if dis == 1:
            RandomGraph(15,100)
        elif dis == 2:
            RandomGraphS(15,100)
        else:
            RandomGraphG(15,100)
        for j in range(len(adj_list.keys())):
            vnconflit[j] = vnconflit[j] + len(adj_list[j])

            for a in adj_list[j]:
                if a[0]+j in estable.keys():
                    estable[a[0]+j] = estable[a[0]+j] + 1
                else:
                    estable[a[0]+j] = 1


    for i in range(len(adj_list.keys())):
        vnconflit[i] = vnconflit[i] / 100

    for i in estable:
        estable[i] = estable[i] / 100

    print("Number of vertices", len(adj_list.keys()))
    print("The average number of degrees for each vertex")
    print(vnconflit)
    for key in sorted(estable):
        print  (key, " and ", estable[key])
```

In [7]:

```
FindConflicts(1)
```

```
Number of vertices 15
The average number of degrees for each vertex
[13.41, 13.3, 13.22, 13.48, 13.31, 13.32, 13.31, 13.32, 13.21, 13.44, 13.27, 13.36, 1
3.35, 13.38, 13.32]
1  and  1.96
2  and  1.84
3  and  3.84
4  and  3.88
5  and  5.7
6  and  5.56
7  and  7.6
8  and  7.6
9  and  9.48
10  and  9.68
11  and  11.38
12  and  11.42
13  and  13.6
14  and  13.36
15  and  13.34
16  and  11.42
17  and  11.22
18  and  9.34
19  and  9.46
20  and  7.72
21  and  7.6
22  and  5.8
23  and  5.78
24  and  3.9
25  and  3.64
26  and  1.96
27  and  1.92
```

From the result above we can find out in a random graph with uniform distribution, the average number of degrees for each vertex is close, which is around 13. And for pairs of edges, from 0(0,1) to 27(13,14), it kinds form a Gaussian distribution. That's because if the sum is too big or too small, the number of ways of selection will be small.

7.2 Random Graph with Skewed Distribution

In [11]:
```
FindConflicts(2)
```

```
Number of vertices 15
The average number of degrees for each vertex
[13.98, 13.89, 13.82, 13.83, 13.68, 13.69, 13.66, 13.53, 13.46, 13.2, 13.17, 12.94, 1
2.71, 12.33, 12.11]
1  and  2.0
2  and  2.0
3  and  4.0
4  and  4.0
5  and  6.0
6  and  6.0
7  and  8.0
8  and  8.0
9  and  9.96
10  and  10.0
11  and  11.94
12  and  11.96
13  and  13.78
14  and  13.8
15  and  13.18
16  and  11.5
17  and  11.42
18  and  9.34
19  and  9.28
20  and  7.42
21  and  7.02
22  and  5.44
23  and  4.96
24  and  3.52
25  and  3.14
26  and  1.4
27  and  0.94
```

From this result you will find out that as the vertex becomes bigger, the average number degress for it becomes smallers. It forms a Skewed distribution.

7.3 Random Graph with Gaussian Distribution

In [13]:
```
FindConflicts(3)
```

```
Number of vertices 15
The average number of degrees for each vertex
[12.86, 12.91, 13.27, 13.49, 13.51, 13.59, 13.64, 13.54, 13.61, 13.63, 13.53, 13.37, 1
3.29, 13.07, 12.69]
1  and  1.18
2  and  1.6
3  and  3.3
4  and  3.64
5  and  5.4
6  and  5.72
7  and  7.72
8  and  7.78
9  and  9.66
10  and  9.84
11  and  11.76
12  and  11.82
```

```
13  and  13.76
14  and  13.86
15  and  13.64
16  and  11.88
17  and  11.84
18  and  9.8
19  and  9.66
20  and  7.76
21  and  7.46
22  and  5.62
23  and  5.52
24  and  3.68
25  and  3.38
26  and  1.52
27  and  1.2
```

For this type of graph, since in the case of uniform distribution we already know the distribution of sum of source and destination is a Gaussian Distribution, making the selection to a Gaussian Distribution then may not have a huge change. However, if we check the average number of degrees for each vertex, we can still see a result of Gaussian distribution.

8: Make File I/O for later part B

8.1 Source Code

In [45]:

```
Nvertices = 0
Nedge = 0
Gtype = ""
Dtype = ""


def ProcessInput():
    if Gtype == "Complete":
        CompleteGraph(Nvertices)
    elif Gtype == "CYCLE":
        CycleGraph(Nvertices)
    elif Gtype == "RANDOM":
        if Dtype == "UNIFORM":
            RandomGraph(Nvertices,Nedge)
        elif Dtype == "SKEWED":
            RandomGraphS(Nvertices,Nedge)
        elif Dtype == "GAUSSIAN":
            RandomGraphG(Nvertices,Nedge)
        else:
            print("Invalid Gtype")
    else:
        print("Invalid Gtype")


def LoadInput(filename):
    InputList = []
    with open(filename,'r') as f:
        InputList = f.read().splitlines()

    global Nvertices,Nedge,Gtype,Dtype

    Nvertices = int(InputList[0])
    Nedge = int(InputList[1])
    Gtype = InputList[2]
    if Gtype == "RANDOM":
        Dtype = InputList[3]
    f.close()
```

In [46]:
```
LoadInput("InputA.txt")
ProcessInput()
PrintOutput(False,"output.txt")
```

```
5
6
7
7
9
9
2
4
0
2
```

InputA.txt is

- 5
- 2
- RANDOM
- GAUSSIAN