实验报告：算法计量与验证

宁依华 1700015403

一、 验证list的按索引取值确实是O(1)

1.代码

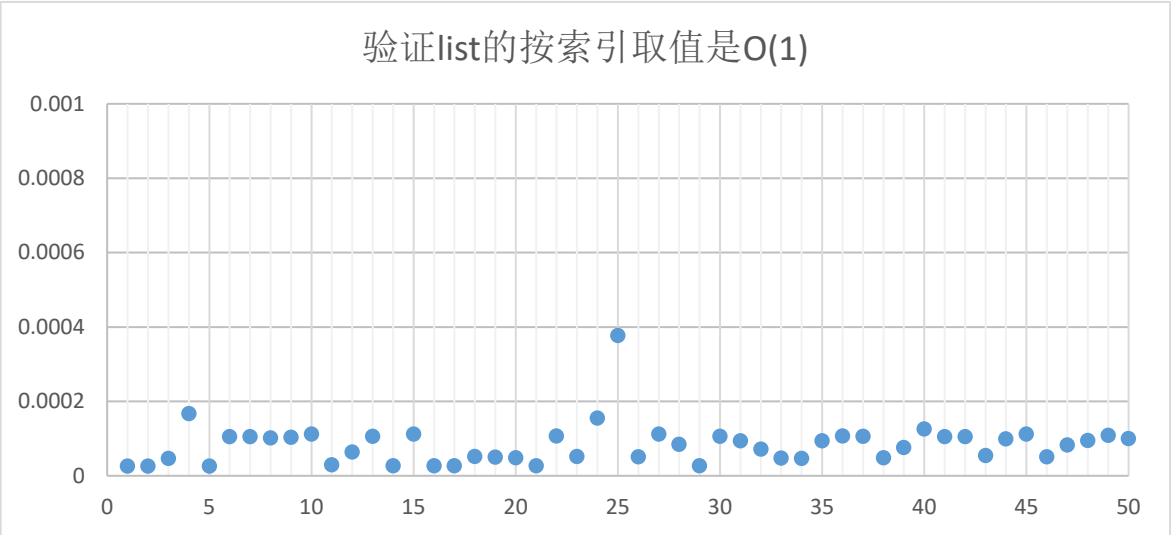
```
#1.验证list的按索引取值确实是O(1)
import random
from timeit import Timer

for i in range(20000,1000001,20000):
    t=Timer("a=thelist[x]", "from __main__ import thelist,x")
    thelist=list(range(i))
    x=random.randint(0,i-1)
    thetime=t.timeit(number=1000)
    print("%d,%f"%(i,thetime))
```

2.数据说明

问题规模N从20000至1000000共50个。对于一个N，list的长度为N，值为下标。代码运行时，把list中的N个元素任取一个赋值给a，a被这个值赋值1000次。

3.运行结果说明

运行得出的数据已经画成散点图，由简单的散点图已经可以看出，数据的波动很小，因此已经可以看出复杂度是O(1)，故不再作回归。



4.结果分析

根据散点图，发现list按索引取值的花费的时间不随N增大而显著增大。去除离群数据后，整体呈现较平稳的趋势，验证了list的按索引取值是O(1)。

二、 验证dict的get item和set item都是O(1)的

1.代码

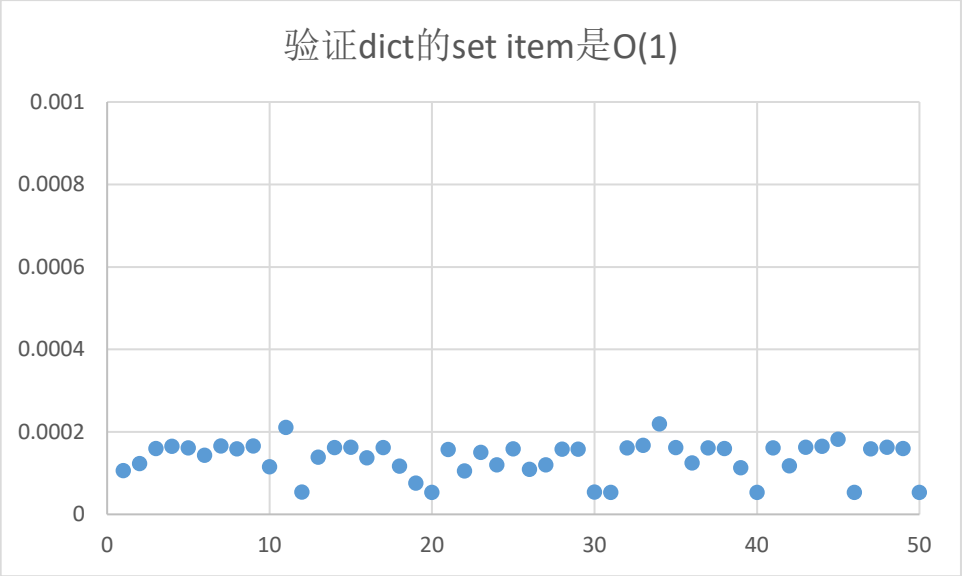
```
#2.验证dict的set item和get item都是O(1)的
import random
from timeit import Timer
for i in range(20000,1000001,20000):
    t1=Timer("thedict[x]=0", "from __main__ import thedict,x")
    t2=Timer("b= thedict[y]", "from __main__ import thedict,y")
    thedict={j:None for j in range(i)}
    x = random.randint(0, i - 1)
    y = random.randint(0, i - 1)
    thetimeset=t1.timeit(number=1000)
    thetimeget=t2.timeit(number=1000)
    print("%d,%f,%f"%(i,thetimeset,thetimeget))
```

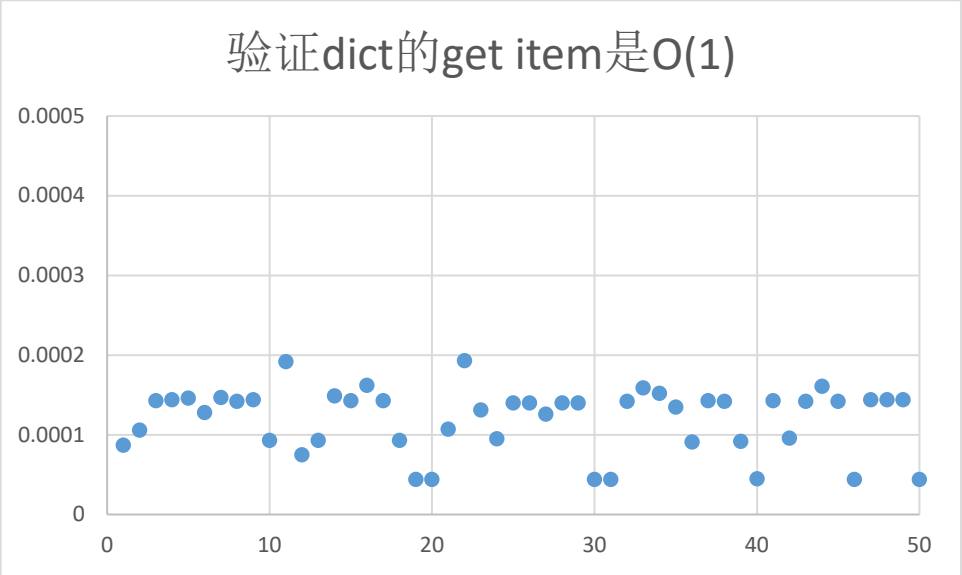
2.数据说明

生成有N个元素的字典，key为从0到N-1的自然数，值为None。将dict的set item和get item都重复1000次。

3.运行结果说明

运行结果已经画成散点图，由简单的散点图可以看出，数据变动基本波动不大。





4.结果分析

根据散点图，已经可以发现dict的set item和get item花费的时间不随N增大而显著增大，不必再做回归。去除离群数据后，整体呈现较平稳的趋势，验证了时间复杂度是O(1)。

三、比较list和dict的del操作符性能

1.代码

```
#3.做计时实验，比较list和dict的del操作符性能
import random
from timeit import Timer
for i in range(20000,1000001,20000):
    t1=Timer("del thelist[x]", "from __main__ import thelist,x")
    t2=Timer("del thedict[random.randint(0, int(i-10001))]", "from __main__ import thedict,random,i")
    thelist=list(range(i))
    thedict = {j: None for j in range(i)}
    x = random.randint(0, int(i- 10001))
    thetimeldel=t1.timeit(number=10)
    thetimeddel=t2.timeit(number=10)
    print("%d,%f,%f" % (i, thetimeldel, thetimeddel))
```

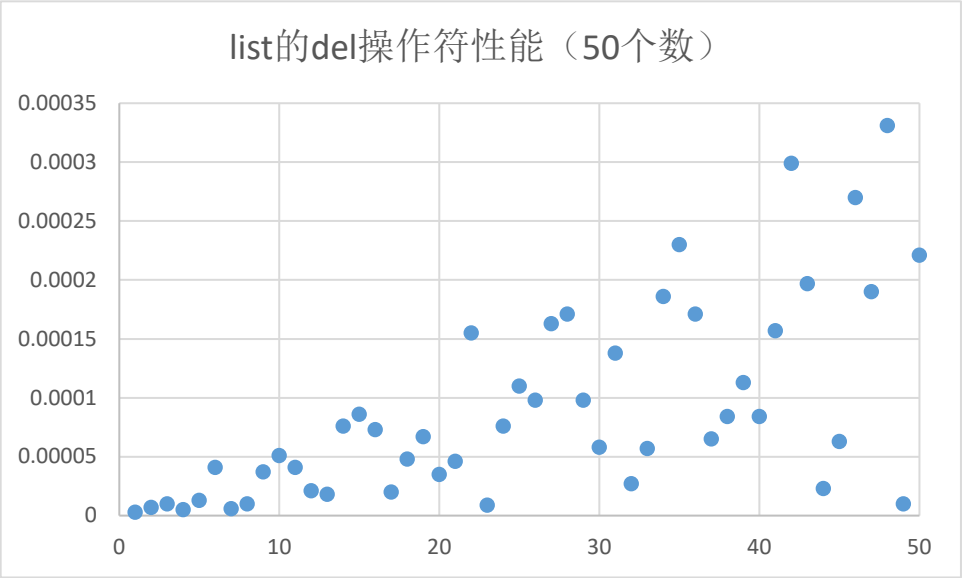
```
#3.做计时实验，比较list和dict的del操作符性能
import random
from timeit import Timer
for i in range(2000,1000001,2000):
    t1=Timer("del thelist[x]", "from __main__ import thelist,x")
    t2=Timer("del thedict[random.randint(0, int(i-1001))]", "from __main__ import thedict,random,i")
    thelist=list(range(i))
    thedict = {j: None for j in range(i)}
    x = random.randint(0, int(i- 1001))
    thetimeldel=t1.timeit(number=10)
    thetimeddel=t2.timeit(number=10)
    print("%d,%f,%f" % (i, thetimeldel, thetimeddel))
```

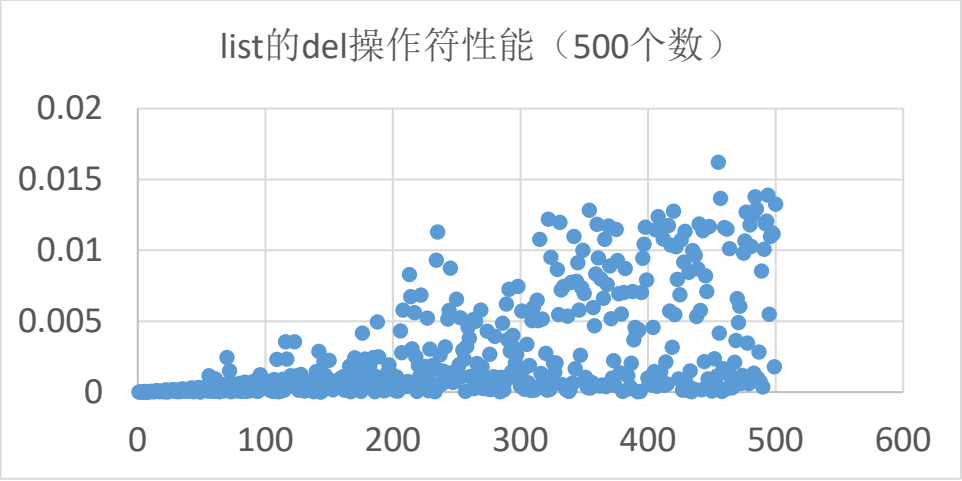
2.数据说明

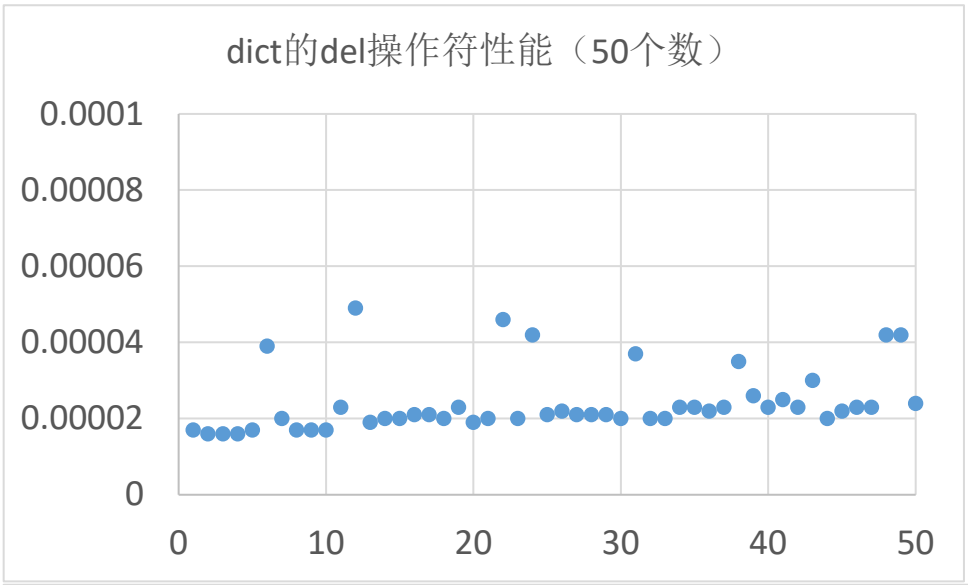
有两组代码，分别令N从20000到1000000取50个值，和N从2000到1000000取500个值。生成长度为N的list和dict，list的值等于下标，dict的key为从0到N-1，值均为零。生成随机的小于N-10000的下标，用del删除该下标的list值和该key的dict值，重复10次。

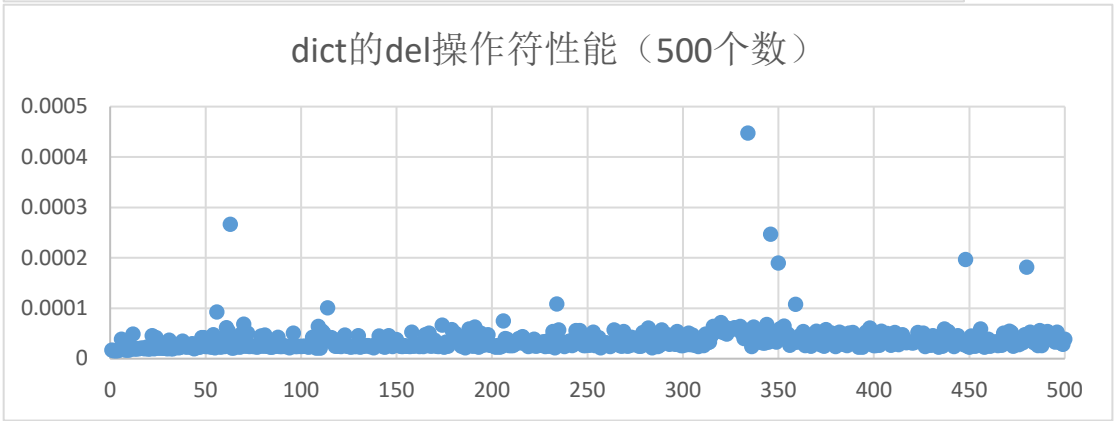
3.运行结果说明

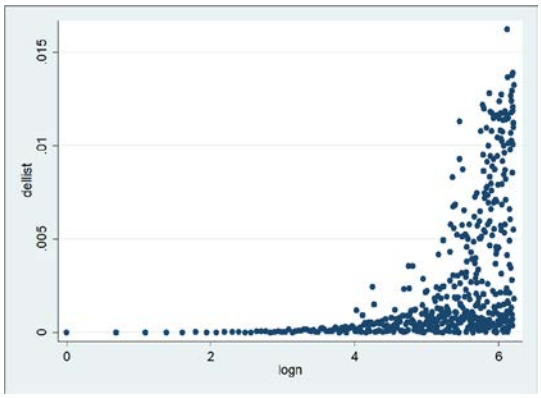
首先说明为什么有2段代码：第一段代码只能得出50组数据，而list.del()的散点图随着N的增大越来越发散，为了得到更多数据，对代码进行了修改，扩大了可以获得的数据量，便于更可靠的分析和后续对各项函数作散点图时的观察。而两段代码中得到的关于dict的del操作符的耗时均比较稳定，已经可以推测是O(1)，故再不作回归。

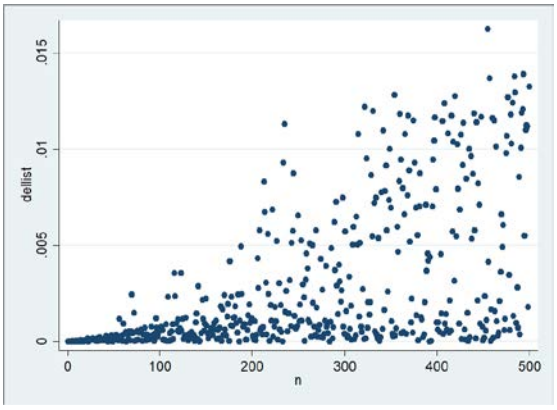


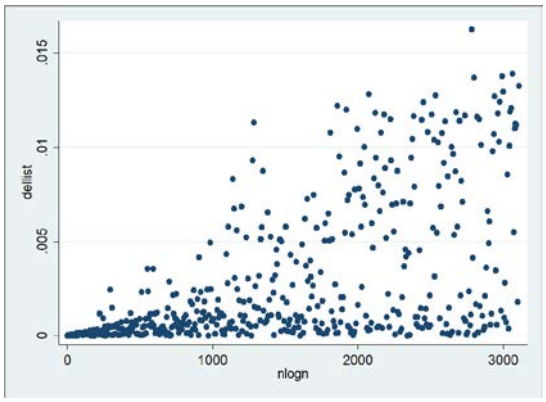


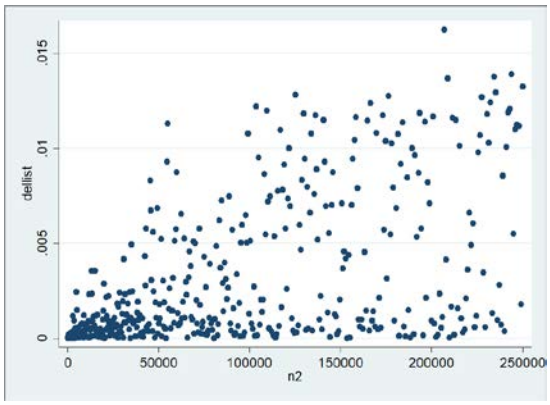

dict的del操作符性能（50个数）
0.0001
0.00008
0.00006
0.00004
0.00002
0
0
10
20
30
40
50


dict的del操作符性能（500个数）
0.0005
0.0004
0.0003
0.0002
0.0001
0
0
50
100
150
200
250
300
350
400
450
500


dellist
0
.005
.01
.015
0
2
4
6
logn


dellist
0
.005
.01
.015
0
100
200
300
400
500
n


dellist
0
.005
.01
.015
0
1000
2000
3000
nlogn


dellist
0
.005
.01
.015
0
50000
100000
150000
200000
250000
n2

4.结果分析

根据结果，发现dict的del()花费的时间不随N增大而显著增大。整体呈现较平稳的趋势，符合其时间复杂度是O(1)的特点。而list的del()花费时间随N变大而增多，且有一定的发散趋势，复杂度至少是O(n)，而不像dict的del()是O(1)。

四、做计时实验，通过对一些随机数列表排序，验证Python自带的list.sort的时间复杂度为O(n logn)

1.代码

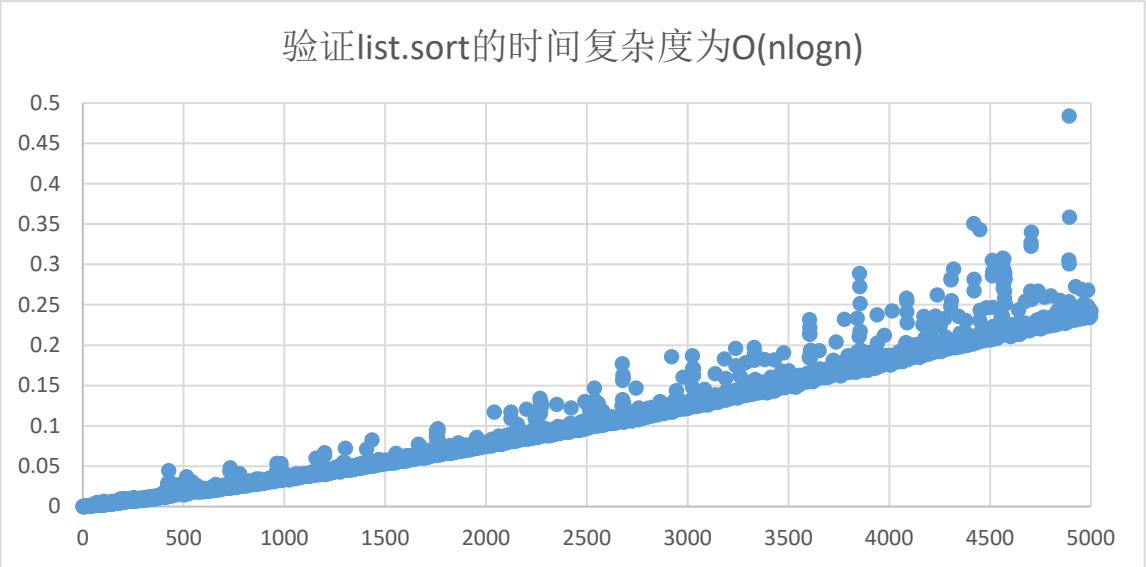
```
#4.对一些随机数列表排序，验证Python自带的list.sort的时间复杂度为O(n logn)
import random
from timeit import Timer
for i in range(20,100001,20):
    t=Timer("thelist[:].sort()", "from __main__ import thelist")
    thelist= [random.randrange(10**6) for n in range(i)]
    thetimesort=t.timeit(number=10)
    print("%d,%f"%(i,thetimesort))
```

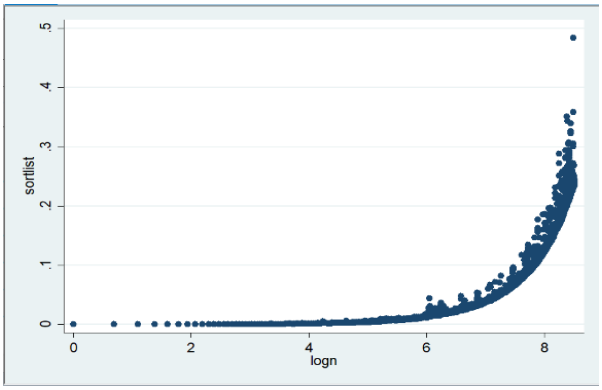
2.数据说明

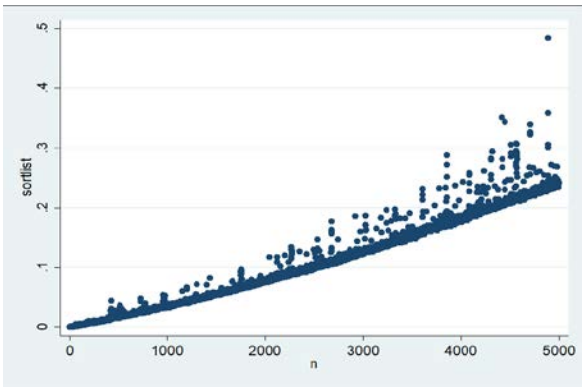
N从20到100000共5000个。生成长度为N的list，取1000000以内的随机自然数赋值，作list.sort()的排序，循环10次排序并计时。

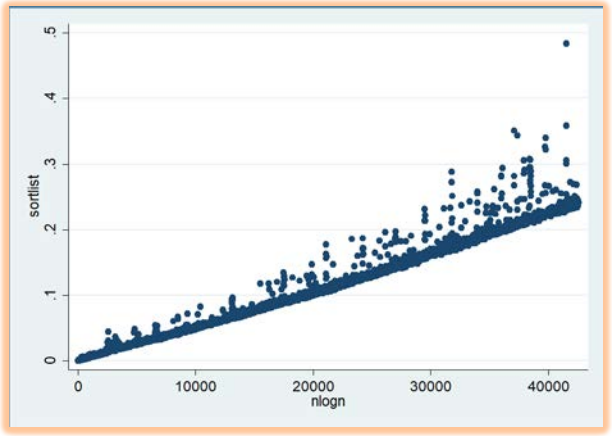
3.运行结果

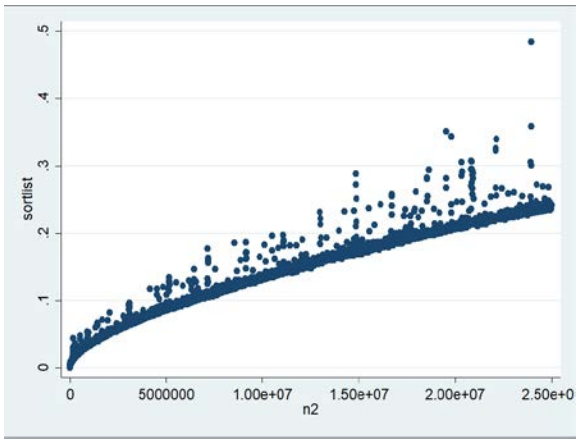
发现由于耗时显然不是线性的，于是分别和一些N的函数作散点图，可以看出凸性的变化。其中，和nlogn做的散点图是最接近直线的。

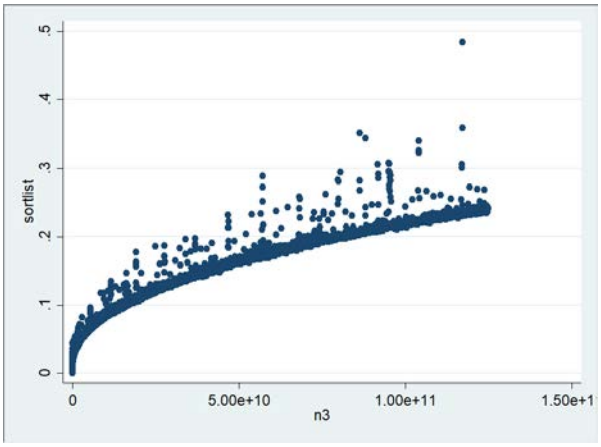













```
. regress sortlist nlogn n logn
```

Source	SS	df	MS	Number of obs	=	5,000
				F(3, 4996)	=	83273.29
Model	25.6960752	3	8.56535839	Prob > F	=	0.0000
Residual	.513880623	4,996	.000102858	R-squared	=	0.9804
				Adj R-squared	=	0.9804
Total	26.2099558	4,999	.00524304	Root MSE	=	.01014

sortlist	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
nlogn	.0000183	6.09e-07	30.10	0.000	.0000171	.0000195
n	-.0001131	5.61e-06	-20.19	0.000	-.0001241	-.0001022
logn	.005544	.0005935	9.34	0.000	.0043804	.0067076
_cons	-.0174473	.0027479	-6.35	0.000	-.0228343	-.0120603

```
. regress sortlist nlogn
```

Source	SS	df	MS	Number of obs	=	5,000
				F(1, 4998)	>	99999.00
Model	25.6189369	1	25.6189369	Prob > F	=	0.0000
Residual	.591018936	4,998	.000118251	R-squared	=	0.9775
				Adj R-squared	=	0.9774
Total	26.2099558	4,999	.00524304	Root MSE	=	.01087

sortlist	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
nlogn	5.71e-06	1.23e-08	465.46	0.000	5.68e-06	5.73e-06
_cons	-.0053855	.0002899	-18.58	0.000	-.0059539	-.0048171

```
. regress sortlist n2
```

Source	SS	df	MS	Number of obs	=	5,000
				F(1, 4998)	>	99999.00
Model	25.0440479	1	25.0440479	Prob > F	=	0.0000
Residual	1.16590793	4,998	.000233275	R-squared	=	0.9555
				Adj R-squared	=	0.9555
Total	26.2099558	4,999	.00524304	Root MSE	=	.01527

sortlist	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
n2	9.49e-09	2.90e-11	327.66	0.000	9.44e-09	9.55e-09
_cons	.0298784	.000324	92.21	0.000	.0292432	.0305136

```
. regress sortlist n3
```

Source	SS	df	MS	Number of obs	=	5,000
				F(1, 4998)	=	37496.81
Model	23.1272882	1	23.1272882	Prob > F	=	0.0000
Residual	3.08266756	4,998	.00061678	R-squared	=	0.8824
				Adj R-squared	=	0.8824
Total	26.2099558	4,999	.00524304	Root MSE	=	.02484

sortlist	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
n3	1.92e-12	9.91e-15	193.64	0.000	1.90e-12	1.94e-12
_cons	.0490287	.0004683	104.69	0.000	.0481106	.0499468

```
. regress sortlist n4
```

Source	SS	df	MS	Number of obs	=	5,000
				F(1, 4998)	=	20528.64
Model	21.0781662	1	21.0781662	Prob > F	=	0.0000
Residual	5.13178958	4,998	.001026769	R-squared	=	0.8042
				Adj R-squared	=	0.8042
Total	26.2099558	4,999	.00524304	Root MSE	=	.03204

sortlist	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
n4	3.89e-16	2.72e-18	143.28	0.000	3.84e-16	3.95e-16
_cons	.0603141	.0005665	106.47	0.000	.0592036	.0614247

4.结果分析

由时间和x的一些简单函数的散点图可以看出，耗时与nlogn的关系最接近线性，与list.sort()的复杂度为O(nlogn)相符。且将耗时与nlogn，n，logn做回归得到的系数大大小相较于其他统计模型的函数形式相比更合适，其他模型得到的系数都太小了，虽然p值也=0，但不如这个合适。因此，验证了list.sort的时间复杂度为O(nlogn)。