CSE6242 Spring 2017 - OMS
HW1: R Programming
GT account name: xtao41

1. Get familiar with R.
   Observation about programming in R, with sample code snippet and output
   Types of subscripts in R.
   1) Positive integer index
      Unlike other programming language like C/C++, vector subscripts starts from 1. If subscript is out of range, it outputs 'NA' for vectors, but gives error for arrays.

```
#vector
> x=c(1,2,3,4,5)
> x[1]
[1] 1
> x[6]
[1] NA

# array
 > z=array(data=x,dim=c(2,3))
 > z[1,]
 [1] 1 3 5
 > z[3,]
 Error in z[3, ] : subscript out of bounds
```

   2) Negative integer index
      For vectors, x[-1] refers to all elements but the 1st one.

```
#vector
> x=c(1,2,3,4,5)
> x[-1]
[1] 2 3 4 5
> x[-(2:4)]
[1] 1 5

# array
> z=array(data=x,dim=c(2,3))
> z[,-2]
     [,1] [,2]
[1,]    1    5
[2,]    2    1
```

   3) Zero
      It produces nothing, not even an error (get ignored). This can potentially generate design flaws (hard to debug without error).

```
 > x=c(1,2,3,4,5)
 > x[0]
 numeric(0)
 > x[c(1,0,3,0)]
 [1] 1 3
```

```
> x[7]=7
> x
[1]  1  2  3  4  5 NA  7
```

4) Boolean index

It is very handy to select element, compare elements and make assignments.

```
> x=c(1,2,3,4,5)
> x[x>2]
[1] 3 4 5
> x>2
[1] FALSE FALSE  TRUE  TRUE  TRUE
> x[x>2]=3.5
> x
[1] 1.0 2.0 3.5 3.5 3.5
```

5) Nothing

For vector, missing subscript returns the vector itself. For array, missing subscript in one coordinate means returning all element of that coordinate.

```
#vector
> x=c(1,2,3,4,5,6)
> x[]
[1] 1 2 3 4 5 6

# array
z=array(data=x,dim=c(2,3))
> z[,2]
[1] 3 4
```

6) Mixed indexes

Mixing positive and negative is not allowed. Mixing zero and integer results in zero being ignored.

```
> x=c(1,2,3,4,5,6)
> x[c(-1,2)]
Error in x[c(-1, 2)] : only 0's may be mixed with negative subs
cripts
> x[c(0,-1)]
[1] 2 3 4 5 6
> x[c(0,2)]
[1] 2
```

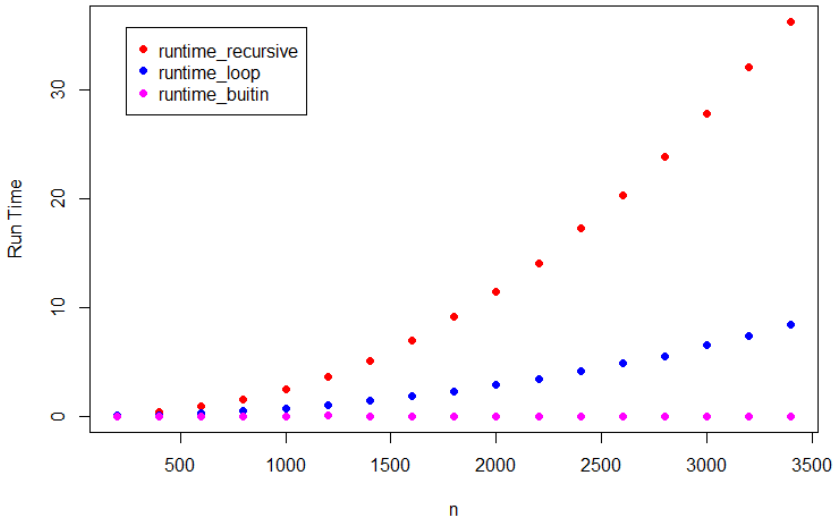
5. Compare Results to Built-In R Function-Report.

1) Comparison of the execution times:

The table below shows the run time (in second) of three different methods

- The recursive method is the slowest implementation. The running time of the loop method is between the recursive and the one implemented based on the built-in R function. The implementation based on the built-in lgamma(n) is the fastest method. For example, when n=3400, the recursive method is 4.3 times slower than the loop method, and the one with builtin method still has running time close to 0 sec.
- Timing of recursive implementation: sum_log_gamma_recursive(n) starts to see overflow when n equals to some number between 3500-3600.

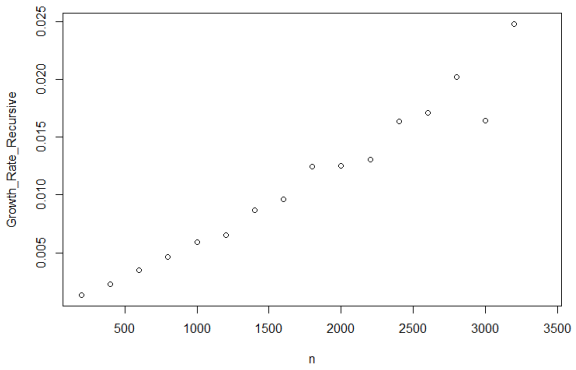
n	runtime_loop	runtime_recursive	runtime_buitin
200	0.03	0.1	0
400	0.12	0.39	0
600	0.25	0.87	0
800	0.44	1.55	0
1000	0.72	2.5	0
1200	1.01	3.59	0
1400	1.45	5.11	0
1600	1.84	6.99	0
1800	2.29	9.11	0
2000	2.88	11.41	0
2200	3.43	13.98	0
2400	4.13	17.23	0
2600	4.85	20.28	0
2800	5.47	23.82	0
3000	6.5	27.8	0
3200	7.4	32.05	0
3400	8.42	36.19	0



2) Growth Rate: Use the simple formula to calculate Growth Rate.
Growth Rate=ΔRun_Time/Δn

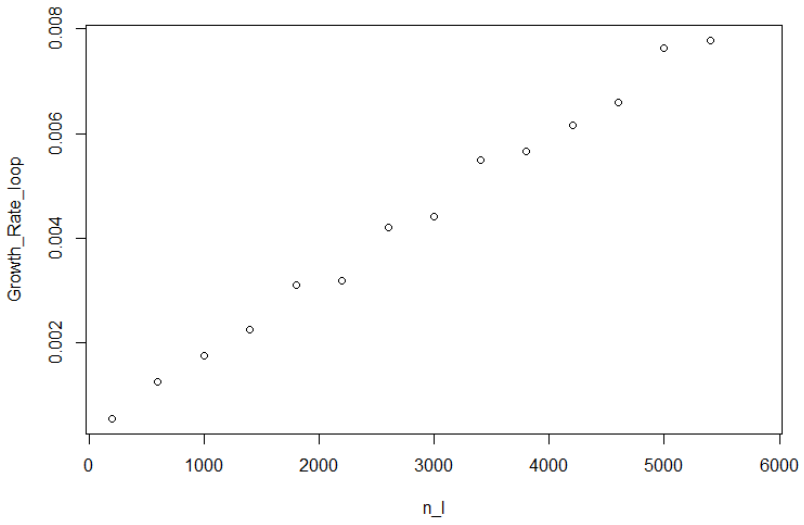
- Growth Rate of recursive method: I calculated the growth rate before it went overflow (n<3600). The growth rate increases almost linearly as n increases.

n	runtime_recursive	Growth_Rate_Recursive
200	0.1	0.00135
400	0.37	0.00230
600	0.83	0.00350
800	1.53	0.00465
1000	2.46	0.00595
1200	3.65	0.00655
1400	4.96	0.00870
1600	6.7	0.00960
1800	8.62	0.01245
2000	11.11	0.01255
2200	13.62	0.01305
2400	16.23	0.01635
2600	19.5	0.01710
2800	22.92	0.02025
3000	26.97	0.01645
3200	30.26	0.02480
3400	35.22	*NA*



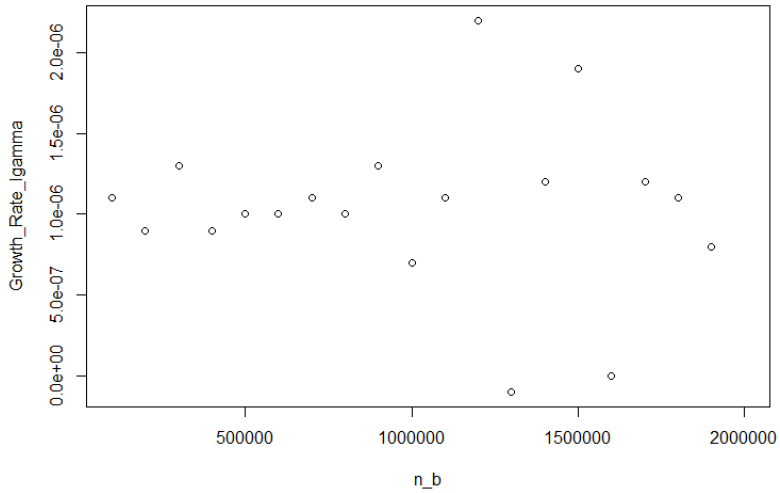
- Growth Rate of loop method: I calculated the growth rate with n<6000, when it slows down significantly. The growth rate increases almost linearly as n increases, but increases slower than recursive model.

n_l	runtime_loop2	Growth_Rate_loop
200	0.01	0.000550
600	0.23	0.001250
1000	0.73	0.001750
1400	1.43	0.002250
1800	2.33	0.003100
2200	3.57	0.003175
2600	4.84	0.004200
3000	6.52	0.004400
3400	8.28	0.005500
3800	10.48	0.005650
4200	12.74	0.006150
4600	15.2	0.006600
5000	17.84	0.007625
5400	20.89	0.007775
5800	24	*NA*



- Growth Rate of implementation based on built-in function: the execution time is very small for n<100000 (less than 0.1s). The growth rate is almost constant as n increases(n>100000, calculation not very reliable when n<100000). In fact, it is almost zero when n<10000.

n_b	runtime_lgamma	Growth_Rate_lgamma
30000	0.03	1.00E-06
60000	0.06	2.00E-06
80000	0.07	2.00E-06
90000	0.09	2.00E-06
100000	0.11	1.10E-06
200000	0.22	9.00E-07
300000	0.31	1.30E-06
400000	0.44	9.00E-07
500000	0.53	1.00E-06
600000	0.63	1.00E-06
700000	0.73	1.10E-06
800000	0.84	1.00E-06
900000	0.94	1.30E-06
1000000	1.07	7.00E-07
1100000	1.14	1.10E-06
1200000	1.25	2.20E-06
1300000	1.47	-1.00E-07
1400000	1.46	1.20E-06
1500000	1.58	1.90E-06
1600000	1.77	2.84E-19
1700000	1.77	1.20E-06
1800000	1.89	1.10E-06
1900000	2	8.00E-07
2000000	2.08	*NA*



- Comparison of growth rates of run time: For both Loop and recursive method, the growth rate increases as n increases. But the recursive method increases faster than loop method. For example, when n=1000, recursive model grows 3.4 time faster than loop method and when n=3000, recursive model grows 3.7 time faster than loop method. In contrast, the growth rate of the implementation based on the built-in function is almost constant and very small as n increases. In fact, it is almost zero when n<10000, even for n on the order of 10^6, the growth rate is neglectable (~1.0E-06).

3) Why we see specific growth rates for the various functions?
    - For the examples above, when using loops in R, because it is an interpreted language, every operation carries a lot of extra baggage/user-defined functions, as well creates environments for execution, assigns arguments to the environment, etc. Also loops can't easily be vectorised because each iteration depends on the previous. Thus, it shows a strong slowing down as dataset becomes bigger.
    - Recursion is even more slower than Loops in this case. One reason is that the recursion function always calls itself. A function call involves saving current stack frame and starting with a new stack frame (additional push and pop instructions), and thus takes more time. As the data set becomes large, the recursion can become too deep and easy to get stack overflow.
    - Unlike the functions written in R code, which is interpreted when they are run, the built-in function "lgamma" and "gamma" is based on C translations of Fortran subroutines. and is pre-compiled. Therefore, it runs much faster than the two user-defined R functions based on R.