# Self-Test & Grading

You have experienced how your assignments have been graded for a couple of months. Now, I ask you to grade your assignment by yourself.

* **"timeit**" means that you measure the elapsed time with 1 Million nodes if the time complexity of the function is O(n), and 100,000 nodes for O(n^2) in your computer.
* **Mark X if you have not implement or if it does not work. You may give a partial credit, but describe what they were in the right most column.**
* You may expand this form as needed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Steps** | **pnts** | **Functions** | **Self-Grade**  **or Mark X** | Use this column to record timeit.  Record your failure case or why you minus point or comments about your test results.  **Don't list successful test cases themselves here.** |
| Step 1 | 1.0 | push\*() | 1.0 | 1st and last node, large list |
| Step 2 | 1.0 | pop\*() | 1.0 | pop all nodes, 1st and last node, large list |
| Step 3 | 1.0 | show() | 1.0 | small or large n, check 6 digit numbers |
| Step 4 | 1.0 | "F", "B",  "Y", "P" | 1.0 | timeit:  F: 0.188(sec)  P: 1.643(sec)  B: 39.917(sec)  Y: 78.824(sec) |
| Step 5 | 2.0 | push\_N(),  pop\_N() | 2.0 | Modify driver file from this step  timeit:  F: 0.181(sec)  P: 1.574(sec)  B: 41.566(sec)  Y: 80.598(sec) |
| Step 6 | 2.0 | reverse\_ in\_stack() | 2.0 | timeit: 0.141(sec) |
| Step 7 | 2.0 | reverse\_ in\_place | 2.0 | timeit: 0.027(sec)  this must be much faster than step 6 |
| Step 8 | 2.0 | reverse\_ odd2() | 2.0 | timeit: 33.856(sec)  all odds, all evens, beginning or ending with odd numbers or even numbers, mixed. |
| Step 9 | 3.0 | reverse\_ oddn() | 3.0 | Timeit: 0.073(sec)  all odds, all evens, beginning or ending with odd numbers or even numbers, mixed. |

Your comments overall if necessary:

Analysis:

1. Estimate the elapsed time of push\_backN and pop\_backN for running 1 million of nodes.  
   Use the elapsed time for 100,000 nodes you are getting from your machine, respectively.   
     
   First what we need to find is, ‘How many nodes can it push back’ or ‘How many nodes can it pop back’ in one second. by calculating this, we can easily get time performance of this program with certain amount of data N.

According to Step4. Timeit, we got approximately 40 seconds while pushing 100,000 nodes in the linked list. Since, Push\_backN is O(n^2) operation, we can assume that Push\_backN operation can push nodes per 1 second.

Just like Push\_backN, we can go through the same steps in order to achieve Pop\_backN’s performance is 1 second. Calculating , we can assume that is pops nodes per 1 second.

Now, we can predict time duration for running 1 million nodes respectively. Both operation has time complexity of O(n^2), so by dividing with ‘Nodes each operation can push / pop’, we can get time duration for 1 million nodes.

Therefore, for Push\_backN to push back 1million nodes, it takes .

In addition, for Pop\_backN to pop back 1million nodes, it takes

1. Estimate the elapsed time of reverse\_odd2() running for 1 million of nodes.  
   Use the elapsed time for 100,000 nodes you are getting from your machine, respectively.  
     
   Reverse\_odd2 is an algorithm with time complexity of O(n^2). Reverse\_odd2 function reverses only ‘odd numbers’, so time duration is affected by the number of odd integers. But for convenience let’s ignore that factor, since its difference is minor problem. So, in order to calculate ‘How many nodes can it reverse’, we can refer to calculated time in step 8. According to step 8, reversing 100,000 nodes took 33.856 seconds. So we can estimate that it can reverse nodes in 1 second.

Now, to estimate time duration for 1 million nodes, we have to divide with ‘Number of nodes Reverse\_odd2 can reverse in 1second’. Therefore reversing 1 million nodes using Reverse\_odd2 function, it will take approximately .

1. Analyze the elapsed time of running reverse\_using\_stack() and reverse\_in\_place() with a million nodes.

Both algorithms’ time complexity is O(n). However, we can see that Reverse\_in\_place() is faster than Reverse\_using\_stack. Why does this happen? By looking at codes respectively, we can analyze the difference. They both are functions that reverses linked list.

Reverse\_using\_stack function saves all nodes of linked list in a stack, pops them all and relinks it backwards. This is inefficient way of implementing reversing function because it actually takes twice of time. Frankly speaking, Reverse\_using\_stack’s time complexity is about T(2n).

On the other hand, Reverse\_in\_place function just goes through the list once, and simply relinks them in reverse order. By this way, its time complexity shrinks into T(n). Which is twice faster than previous function.

By observing Step 6 and 7, we can say that Reverse\_using\_stack is much slower than Reverse\_in\_place. And we can conclude that although having the same Big-O notation, 2 different algorithms can have performance difference according to how algorithm was implemented.