INTRODUCTION TO TIME COMPLEXITY

Objective :

1. To see how efficiency of an algorithm can be modelled in such a way that it can be compared against those of others.
2. To familiarize with computational problems' specifications.
3. To familiarize with the experiment platform, based on Python programming, that is used throughout the course.

Statement of problem : Given a list of n numbers and an integer k, find a pair of *different* numbers in the list that multiply to exactly k. If no such pair exists, print "no such pair exists" as output.

NOTE: Each number, including k, can be positive or negative.

From the Class Materials for week 1, download Week1.zip.

Week1.zip contains 8 test cases for this problem.

- The only number in the first line is the targeted product, K.
- The second line contains list of numbers.
- Max, Min possible values of any number are 12999709, -12999709
- 100.in and 1000.in are test cases to verify the correctness of the program
- Answer to the case that begins with "t" is "no pair exists". It is used for producing worst-case running time.

1) Examine m2kNaive.py
   The naïve solution consists of nested for loop, one for each number to be considered, and if the multiplication matches the value K, the program breaks out of the loop and print result.

   The running time for the processing part of the code can be calculated as follows.

```
import time


start_time = time.process_time()
# part of the code whose running time is to be measured
end_time = time.process_time()
running_time = end_time – start_time
```

   Verify the correctness of m2kNaive.py by running with 100.in and 1000.in

2) Run the naïve solution with the provided “t” test cases. Record the running time in the provided RunningTime.xlsx spreadsheet.

3) Examine m2kLoop.py
   This version calculates the target co-multiplier of each value such that both multiply together into K. Then, scan the number list to check whether the co-multiplier also exists in the list. If found, the loop are broken and the program prints result.

   Verify the correctness of m2kNaive.py by running with 100.in and 1000.in

4) Run the loop-search solution with the provided “t” test cases. Record the running time in the provided RunningTime.xlsx spreadsheet.

5) Examine m2kList.py
   This version is virtually the same as m2kLoop.py, except that it utilizes the “in” operation to scan the list instead of using for loop. The implementation of list operation is in faster programming language, so the running time can be expected to improve.

   Verify the correctness of m2kNaive.py by running with 100.in and 1000.in

6) Run the list-search solution with the provided “t” test cases. Record the running time in the provided RunningTime.xlsx spreadsheet.

7) Examine m2kBinS.py
   As the name indicates, this version first sort the number list, then seach for the co-multiplier using Binary Search approach. The mechanism and expected speed of Binary Search can be studied at Binary search algorithm - Wikipedia

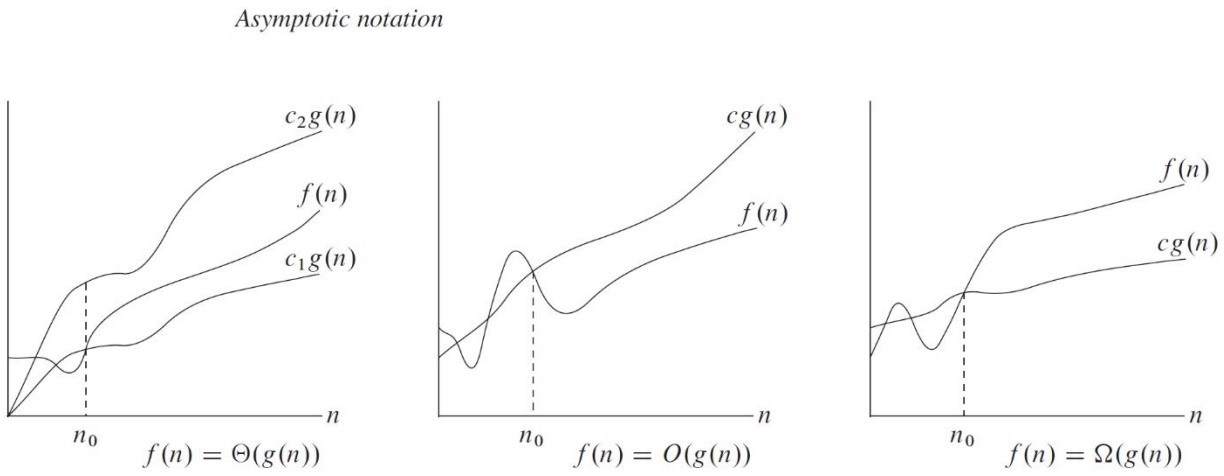
   Verify the correctness of m2kNaive.py by running with 100.in and 1000.in

8) Run the binary-search solution with the provided “t” test cases. Record the running time in the provided RunningTime.xlsx spreadsheet.

9) The running time curves will show that there are three programs that actually produce the same curve’s shape. These curves are different by only a constant factor, which is equivalent to changing CPU, changing OS, or changing programming language. Their growth over the size of input is virtually the same. These curves reflect the fact that the three programs realize the same algorithm.

10) One of the curves is clearly of different shape, indicating lower growth of time over the input size. This is because it utilizes different algorithm.

11) Asymptotic notations are informally described as the following.



$f(n)$ is the function of interest (running time of the algorithm, for a certain case of input)
$g(n)$ is the representative function of the set of functions that are categorized into the same asymptotic bound

Three most common asymptotic bounds are Θ (tight-bound), O (upperbound), and Ω (lowerbound)

O (called big-Oh) is the most commonly used because

- It is easy to estimate than Θ
- Ω does not necessarily indicates the expected running time
- An upperbound indicates worst-case scenario, which guarantees that the algorithm will never take longer.

12) Propose the upperbound on the running time for each of the four programs experimented in this class.

EXTRA (if you are fast to reach this point)

- Try to come up with your own algorithm to solve this problem. Then, code it and test run to verify its correctness and if correct, compare its running time with the above four programs.
- Deduce an upperbound on the running time of your solution.