Homework Assignment Week 1

Python Programming and Complexity Analysis

Name : Woonki Kim

Email : Kimwoon@oregonstate.edu

1. Code of Python function named *factors* that returns all prime factors of an integer.

```
#Assignment Week1.
#Written by Woonki Kim.



# Write an efficient Python function named factors that returns a
ll prime factors of an integer.
#For example, factors(12) returns [2,2,3]. If the input is a prim
e or 1 it returns an empty list.
# The factors should be listed in increasing order.

def factors(n):
    arr = []
    if n ==1 :
        return []
    else:
        divisor = 2
        while divisor*divisor <= n :
            if n%divisor == 0:
                arr.append(divisor)
                n = n//divisor
            else:
                divisor += 1
```

```
        if n > 1:
            arr.append(n)

        if len(arr)== 1:
                arr.clear()

    return arr
```

💡

- while loop's condition is `divisor*divisor <= n`, since we only need to check whether n is divisible by `sqrt(n)`.
- for example) n= 36 the factors are (1,36),(2,18),(4,9),(6,6),(36,1),(18,2),(9,4)
  as you can see in this example after the number of sqrt(n) which is 6, it is just repeated pairs of number that is inverted.

💡

Why don't we have to check whether `n` is prime or not in while loop in `factors` function?

- We don't need to check whether divisor is prime or not since divisor starts with 2(smallest prime) and keep eliminating smaller primes.
- Which means by the time the loop encounters a composite number, the prime factor of that composite number will already have been divided out.

2. In the report, include the code and a derivation of the running time of your algorithm (a) assuming that multiplications (and additions) take constant time and (b) assuming that multiplication and division of -bit numbers take $O(n^2)$ time and additions and subtractions take $O(n)$ time.

**** Code included in number 1 ****

(a) Assuming Multiplications and Additions Take Constant Time

- When we assume Multiplications and additions take constant time, it means that we just need to check how many iterations occur.
- The number of iteration is $\sqrt{n}$
  - The outer while loop's condition is divisor * divisor ≤ n which is same as divisor≤ sqrt(n).
- Since the multiplication and additions take constant time we can write:
  - $T(n) = \sum_{i=1}^{\sqrt{n}} C$ (where C is constant time that represents the time of any operations done)

    = $C \times (\sqrt{n} - 1 + 1) = C \times \sqrt{n}$
- Thus, time complexity for fractions function:
  - $O(\sqrt{n})$ (Since we learned in exploration that we can ignore the constant because the constant is implementation- and platform-dependent)

(b) Assuming that multiplication and division of n-bit numbers take $O(n^2)$ time and additions and subtractions take $O(n)$ time.

- When we assume multiplication and division of n-bit numbers take $O(n^2)$ time and additions and subtractions take $O(n)$ time.
  - The number n requires $\log_2(n)$ bits to represent.
  - Multiplications and divisions: $O((\log_2(n))^2)$ (since decimal n requires $log_2(n)$ bits to be represented)
  - Additions and subtractions: $O(log_2(n))$
- Number of Iterations is $\sqrt{n}$ (as explained in (a))
- Time taken per iteration:
  - Modulus:
    - Doing modulus calculation to determine whether n is divisible.

```
if n%divisor == 0:
```

    - Time taken: $O((\log_2(n))^2)$ Since we can consider this calculation as division
  - Division:
    - Doing division if n is divisible

```
n = n//divisor
```

    - Time taken : Time taken: $O((\log_2(n))^2)$
  - There's a 2 division calculation in while loop making total time spent per iteration : $O((2\log_2(n))^2)$

- Calculating the total time:
  - $T(n) = \sum_{i=1}^{\sqrt{n}} O((2\log_2 n)^2)$
    = $O(\sqrt{n}) \times O((\log_2 n)^2)$ (since constant can be ignored)
    = $O(\sqrt{n}(\log_2(n))^2)$

3. Experimental Results

1. n vs $T(n)$
   - Since the $O(n)$ is representing worst-case, I decided to make input numbers that are smallest prime number that is bigger than $2^{2\alpha}$ *(so that it is easy to calculate $\sqrt{n}(\log_2 n)^2$ derived from 2-(b))*, where α starts from 10 to 15 *(since the number is too small to observe when n is 1 to 10).*
   - Table:

Input number: n	Closest $2's$ square number to n	Approximately estimated Running Time: $\sqrt{n}(\log_2(n))^2$	Approximately estimated Running Time: $\sqrt{n}$	Actual Running Time(Seconds)

1,048,583	2^{20}	$1024 \times 400 = 409600$	1024	0.00005
4,194,319	2^{22}	$2048 \times 484 = 991232$	2048	0.00010
16,777,259	2^{24}	$4096 \times 576 = 2359296$	4096	0.00021
67,108,879	2^{26}	$8192 \times 676 = 5537792$	8192	0.00043
268,435,459	2^{28}	$16384 \times 784 = 12845056$	16384	0.00087
1,073,741,827	2^{30}	$32768 \times 900 = 29491200$	32768	0.00176

2. Does your table closely match one of the running time functions derived in 2?
    - When $T(n) = \sqrt{n}$)
        - As the input number increases estimated time increases by times 2.
    - When $T(n) = \sqrt{n}(\log_2(n))^2$
        - As the input number increases, the estimated time grows with multipliers of approximately 2.42, 2.38, 2.35, 2.32, and 2.30.
        - The growth rate shows gradual decreases.
    - Actual Running time
        - As you can see from actual running time, it increases almost by times 2 and it's rate is stable.
        - **Thus I can say that, the table does closely matches the running time of $O(\sqrt{n})$ derived in 2.**
3. How large can be so that is approximately *5* minutes? What if is *5* hours? *5* days? Factoring is a fundamental crypto-primitive that underlies modern cryptography. What size of makes it practically impossible for your algorithm to factorize, e.g., > *10* years?

- Before getting into calculating time of *5* minutes, *5* hours and *5* days, lets first find out the relationship between ***approximately estimated Running time*** when $T(n) = \sqrt{n}$ and ***actual running time***, since it is shown those two are closely related.
- By seeing a column where the input is 4,194,319
  - $\sqrt{n} = t \times 2048 \times 10^4$
  - $n = t^2 \times 2^{22} \times 10^8 \approx t^2 \times 2^2 \times 10^6 \times 10^8 = t^2 \times 2^2 \times 10^{14}$

- *5* minutes(300seconds)
  - **n** $\approx 90000 \times 4 \times 10^{14} = 3.6 \times 10^{19}$
- 5hours(18,000 seconds)
  - **n** $\approx 3.24 \times 10^8 \times 4 \times 10^{14} = 1.296 \times 10^{23}$
- 5days (432,000 seconds)
  - **n** $\approx 1.86624 \times 10^{11} \times 4 \times 10^{14} = 7.46496 \times 10^{25}$
- $>$10 years(315,360,000 seconds)
  - **n** $> 9.94519296 \times 10^{16} \times 4 \times 10^{14} \approx 3.97 \times 10^{31}$

4. State a useful invariant of the loop towards proving the correction of the algorithm

Loop invariant:

- Some property that always holds true before the loop, in each iteration and after the loop terminates.
- Thus in my code the loop invariant is:
  - Before the start of each loop, list arr always is either empty or holds prime factor of n.

5. Prove that the algorithm is correct using your previously defined invariant.

Initialization:

Before the first iteration of the while loop:

- The list arr is initialized to be empty: arr = [].
- This satisfies the loop invariant since the list is empty

Maintenance:

During while loop,

1. When n %divisor == 0 :
    - In this case, we append divisor to arr.
    - Since divisor is the smallest prime number we are continuously checking (start by 2), and we are only appending it when it divides n, we can say that divisor is a prime factor of n.
    - After this operation, n is updated to n // divisor, and the code continues the loop.
    - So even after this iteration, arr still holds prime factors,
    - Thus, the invariant remains true.
2. When n % divisor ≠ 0:

- In this case, the code simply increases divisor by 1: divisor += 1.
- The arr is not modified and it still contains either no elements (if it was empty) or only prime factors (if it had previously appended them).
- Thus, the invariant remains true.

Termination:

After the while loop is terminated

1. When n > 1:
    - It means that the remaining n is a prime factor of input number, since the code has divided n with its prime factors while divisor * divisor ≤ n.
    - So after the loop it appends prime factor n to list arr.

- Thus, the invariant remains true.

2. When n≤1:
    - The code does not append anything to arr.
    - Thus the invariant remains true.
3. If there’s only one factor in list arr.
    - It means that n is only one prime factor for its own, which means that n is prime number.
    - As described in the question we need to return empty list if n is prime.
    - So the code empties the arr list.
    - Thus, the invariant remains true.