Hyuntaek Oh

Samina Ehsan

CS 514_400

Oct. 7, 2024

Homework Assignment Week1: Python Programming and Complexity Analysis

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

```python
def factors(n):
    ans = []

    # Dividing by 2
    while n % 2 == 0:
        ans.append(2)
        n //= 2

    # Dividing by odd numbers
    d = 3
    while d <= int(n**(1/2))+1:
        if n % d == 0:
            ans.append(d)
            n //= d
        else:
            d += 2

    # if the number is left
    # that would be prime number
    if n > 1:
        ans.append(n)

    # Return an empty array
    # if there is only prime number
    if len(ans) == 1:
        return []

    return ans
```

2. Derivation of the running time of the algorithm

(a) Assuming that multiplications (and additions) take constant time

In the first loop, the algorithm divides n by 2 until n cannot be divided by 2. The time complexity of the loop is $O(log_2n)$ since the number n is exponentially reduced during every dividing by 2. In the second loop, the number n, which is not dividable by 2, is divided by odd numbers from 3 to $\sqrt{n}$, meaning that the time complexity of the second loop is $O(\sqrt{n})$.

Thus, the total time complexity of the algorithm under the assumption that operations take constant time is:

$$T(n) = O(log_2n) + O(\sqrt{n}) = O(\sqrt{n})$$

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

The n-bit number can be represented in binary. For example, 4-bit number 13 can be represented in 1101, meaning that dividing the number by 2 can be $O(log_2n)$. So, multiplication and division of n-bit numbers take $O((log_2n)^2)$ time and additions and subtractions take $O(log_2n)$ time.

In the loop of dividing by 2, the time complexity of it is $O(log_2n)$ since the number is divided by 2 repeatedly, which is as same as (a). Therefore, the total time for the first loop is:

$$T(n)_{divided\ by\ 2} = O(log_2n) \times O((log_2n)^2) = O(log_2n)^3$$

The loop dividing by odd numbers starts from 3 to $\sqrt{n}$. The number of iterations keep running while divisor d is less than or equal to square root of n. Its time complexity is $O(\sqrt{n})$. In the loop, the number n is divided by odd numbers, taking $O((log_2 n)^2)$ time. Thus, the total time for the second loop is:

$$T(n)_{divided\ by\ odd\ numbers} = O(\sqrt{n}) \times O((log_2 n)^2) = O\left(\sqrt{n} * (log_2 n)^2\right)$$

Finally, the total time complexity is under assumption:

$$T(n) = O(log_2 n)^3 + O\left(\sqrt{n} * (log_2 n)^2\right) = O\left(\sqrt{n} * (log_2 n)^2\right)$$

3. Give a table $T(n)$ vs. n from the experimental results. Does your table closely match one of the running time functions derived in 2? How large can n be so that $T(n)$ is approximately 5 minutes? What if $T(n)$ is 5 hours? 5 days? Factoring is a fundamental crypto-primitive that underlies modern cryptography. What size of n makes it practically impossible for your algorithm to factorize, e.g., $T(n) > 10$ years?

A table $T(n)$ vs. n below is from the experimental results.

n	10^m	$T(n)$ (seconds)
1000000007	$\approx 10^9$	0.004
10000000019	$\approx 10^{10}$	0.012
100000000003	$\approx 10^{11}$	0.041
1000000000039	$\approx 10^{12}$	0.139
10000000000037	$\approx 10^{13}$	0.412
100000000000031	$\approx 10^{14}$	1.288
1000000000000037	$\approx 10^{15}$	3.996
10000000000000061	$\approx 10^{16}$	12.758

< **Table 1.** Measurement real-time $T(n)$ vs. n >

The time complexity of the algorithm is $O(\sqrt{n})$, and the empirical data from the table reflects it. As n increases 10 times, the runtime increases roughly between 3 to

4 times. This growth is consistent with $O(\sqrt{n})$ since the square root of 10 would be 3.xxx.

To estimate the size of n for 5 minutes, 5 hours, and 5 days, we need to use the empirical data from the table. When n is equal to 10,000,000,000,000,061 ($\approx 10^{16}$), which is the last data in the table, the running time is 12.76 seconds. Let's start from 5 minutes (300 seconds). The estimation is below:

$$n \approx \left(\frac{300}{12.76}\right)^2 \times 10^{16} = (23.51)^2 \times 10^{16} \approx 552.7 \times 10^{16} = 5.527 \times 10^{18}$$

For 5 hours (18,000 seconds),

$$n \approx \left(\frac{18000}{12.76}\right)^2 \times 10^{16} = (1410)^2 \times 10^{16} = 1.99 \times 10^{22}$$

For 5 days (432,000 seconds),

$$n \approx \left(\frac{432000}{12.76}\right)^2 \times 10^{16} = (33863.3)^2 \times 10^{16} = 1.15 \times 10^{26}$$

For 10 years (315,360,000 seconds),

$$n \approx \left(\frac{315360000}{12.76}\right)^2 \times 10^{16} = (247205589.5)^2 \times 10^{16} = 6.11 \times 10^{30}$$

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

Loop invariants in this algorithm are the current value of n, which is the product of the prime factors that have not yet been added to the list ans at the start of each iteration of the loop, and the list ans contains all prime factors of the original input n that has been discovered through the algorithm. So, we can understand it like:

$n = \{2^1 \times .. \times 2^m \times d_1 \times d_2 \ldots\}$, where the elements are in the list $ans = [2^1, .., 2^m, d_1, d_2, \ldots]$

Initialization: Before any loops, n is the product of the prime factors, and the list ans does not have no prime factor at the start of the algorithm.

Maintenance: After the first iteration, if $n\%2 == 0$, the factor 2 is discovered and added to the list ans, meaning that n is divided by 2. After the second iteration, if $n\%d == 0$, the factor d works in the same way as dividing by 2 did. The remaining n still holds the remaining prime factors, so the invariant is maintained. On the other hand, if $n\%d \neq 0$, the factor d increases, and the invariant still holds because the factor d is not part of n, and nothing is added to the list ans.

Termination: When all the loops are finished, the remaining n is a prime factor, which is greater than $\sqrt{n_{original}}$, and added to the list ans. Then, if there is only one prime factor, the algorithm returns an empty list.

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

Based on the previously defined invariant, we need to prove three stages: initialization, maintenance, and termination. In the initialization, the number n is initial input that has not been divided, and a list ans is an empty list that has not discovered nothing. Since original n is the product of all its prime factors, and no prime factors have been discovered yet, the invariant holds at this stage.

In the maintenance, there are two iteration loops: the first is division by 2, and the second is division by odd numbers. Before the first loop, if n is dividable by 2 as a factor of n, 2 will be added to the list ans. After the iteration, n is updated and consists of remaining prime factors, and then the list ans contains 2 as a factor of n. Before the second division by odd numbers, if n is divisible by the factor d, which is one of the odd numbers and can be a factor of n. After dividing by d, n is still the product of the remaining prime factors, and ans contains all previously discovered prime factors including the new one, d. If d cannot divide n, the next odd number will be $d + 2$, and loop invariant still holds since the previous factor d is not a part of n, and there is no change in the list ans.

Finally, in the termination, after all the loops are finished, the remaining n has only the prime number that is greater than $\sqrt{n_{original}}$ since the loops iterates under the condition $\sqrt{n_{original}}$. The last remaining prime number is inserted into the list ans if there is not only one prime factor, keeping the loop invariant holding.

Thus, the algorithm yields a list of all prime factors of initial value of n.