Assignment 1

CS 514- Algorithms and Data Structures

Submitted by: Aman Pandita

Screenshot:

```python
def factors(n):
    num = n
    x, res = 2, []
    while x <= n:
        if n % x == 0:
            res.append(x)
            n = n // x
        else:
            x += 1
    if res[-1] == num:
        res.pop()
    return res

if __name__ == '__main__':
    print(factors(23))
```

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

The worst-case scenario for any given number "n", would be when the number is prime. This means that the incremental value would have to run till the value "n".

Secondly, there is also a division operator used that is, how many times "n" needs to be divided by "x". In such a worst case, the time complexity would be **O(log n)**

<u>For constant time:</u>

For the while loop: O(n).

<u>Assuming that multiplication and division of n-bit numbers take $O(n^2)$ time and additions and subtractions take O(n) time:</u>

For outer Loop: O(n)
For nested inside the loop: O(log n^2)
This results in O(n $(\log n)^2$)

Also, N is a prime number in the worst case scenario,that is: nested division will mostly not add much complexity, making it only a **O(n $(\log n)^2$)** for bit operations.

<u>Final Conclusion:</u>

1. In worst-case scenario (when n is prime), the algorithm runs in **O(n)** time.
2. In worst-case scenario (when n is prime and considering bit operations), the algorithm runs in **O(n $(\log n)^2$)** time

3. The size of the input n is usually measured by the number of bits needed to represent the input. But here we can use decimal digits since it is directly proportional to the bits. Give a table $T(n)$ vs. n from your 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.

3: T(n) is in ms.

```
TERMINAL
amanpandita@Amans-MacBook-Air ALGO % python3 p1.py
[INFO] Testing on numbers: [67, 137, 277, 557, 1117, 2237, 4481, 8963, 17929, 35863, 71741, 143483, 286973, 573953]

    n     |  T(n) (seconds)
----------------------------
67        |        0.0057  |
137       |        0.0052  |
277       |        0.01    |
557       |        0.021   |
1117      |        0.0522  |
2237      |        0.0982  |
4481      |        0.2129  |
8963      |        0.2968  |
17929     |        0.5751  |
35863     |        1.1411  |
71741     |        2.3079  |
143483    |        4.6091  |
286973    |        8.8072  |
573953    |        17.2741 |
amanpandita@Amans-MacBook-Air ALGO %
```

n	T(n) (in ms)
67	0.0057
137	0.0052
277	0.01
557	0.021
1117	0.0522
2237	0.0982
4481	0.2129
8963	0.2968
17929	0.5751
35863	1.1411
71741	2.3079
143483	4.6091
286973	8.8072
573953	17.2741

Does your table closely match one of the running time functions derived in 2?

- Yes, it matches very closely with what I mentioned in Question 2. We can see that the time doubles every time the input is added. Thus, proving linear time.

How large can n be so that T(n) is approximately 5 minutes.

- The number "n" should be a 10-digit prime number which would take aprox~ 5 mins
  This was calculated using the fact that n = 573953 took 17.2741 ms, and the algorithm grows linearly (O(n)).

What if T(n) is 5 hours? *5* days?

- Similarly, for 5 hours:

  Approx. **10-digit prime number** takes 5 mins, so based on this calculation it should take around **12-digit prime number** which would take 5 hours.

- Therefore 5 days would be needed by an approximately **13-digit prime number.**

- For practically impossible approx. **15-digit**

Proof of Correctness:

Loop Invariant:

Before the start of each iteration of the loop, the original value of n (before any divisions have occurred) divided by the current value of n is a product of primes in the list res.

Initialization:

Before the first iteration of the loop, "res" is an empty list. Original value of n when divided by itself is 1, which is the product of 0 primes, and hence the loop invariant holds true.

Maintenance:

Before each iteration, we assume that the original value of n divided by its current value is a product of primes in the list res. During an iteration, we have two main cases:

1. n % x == 0: In this case, x is a prime factor of n. We append x to the list res and divide n by x. Our invariant still holds since the original value of n divided by the new value of n is still a product of the primes in the list res.

2. n % x != 0: In this scenario, we simply increment x. This action doesn't affect the validity of our invariant since the list res remains unchanged and the value of n is the same as before.

Termination:

At the termination of the loop, the original value of n divided by its current value gives a product that consists solely of primes found in the list res. If the number itself was prime and got appended to res, it gets removed by the if res[-1] == num: res.pop() line.