1. Write a 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.

Please do not search or copy from the internet. You can orally discuss solutions with other students, but any collaboration that involves writing things down will be considered as cheating and earn a zero credit. If you are stuck, please talk to the instructor. Python does not limit the size of the integers so you can run it on arbitrarily large integers (bignums).

```
def factors(num):

    is_prime = True
    # this flag checks if the number is prime
    factor_list = []
    i = 2

    # this loop finds the factors
    while i * i <= num:
        if num % i == 0:
            factor_list.append(i)
            num = num // i
            is_prime = False
        else:
            i = i + 1

    if num != 1 and is_prime == False:
        factor_list.append(num)
    return factor_list
```

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.

a. **assuming that multiplications and division (and additions) take constant time**

Consider the "while" loop which is the main loop used to find the factors to derive the running time of the algorithm–

```
i = 2
# this loop finds the factors
while i * i <= num:
    if num % i == 0:
        factor_list.append(i)
        num = num // i
        is_prime = False
    else:
        i = i + 1
```

The while loop starts with i=2 and num which is initially the input number of n-bits and changes if the loop enters the “if” condition of the while loop.
This loop runs as long as i * i is less than or equal to num. Inside the loop, we perform two operations: If num % i == 0, i.e. – num is divisible by i at any iteration, then i is a factor of num so we append i to the factor_list, perform integer division num // i, and set is_prime to False. These operations take constant time, O (1) in this case.

If num is not divisible by i, i is incremented by 1. Even this operation is assumed to take constant time,

In each iteration of the loop, either i is incremented by 1 or num is divided by i, depending on the condition. In the worst case, as while loop checks for I against num, num does not get divided (or reduced) in any iteration, causing the loop to only execute “else” statement till it comes out of the loop. So, the worst-case number of iterations will be sqrt(num).

We know, time complexity = number of iterations * time taken in each iteration.
= sqrt(num) * O(1) #as operations are constant time
= O(sqrt(num))

Now, as we know that for any number N, its input size(number of bits) n = log2(N). Conversely N = 2^n.

Here, num = 2^n where n is the number of input bits.

Hence, the worst case time complexity is O(sqrt(2^n)) where n is the number of input bits.

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

Consider the while loop to derive the running time of the algorithm–

```
i = 2
# this loop finds the factors
while i * i <= num:
    if num % i == 0:
        factor_list.append(i)
        num = num // i
        is_prime = False
    else:
        i = i + 1
```

The while loop starts with i=2 and num which is the initially the input number of n-bits.
This loop runs as long as i * i is less than or equal to num.

Inside the loop, we perform two operations:
If num % i == 0, i.e. – num is divisible by i at any iteration, then i is a factor of num so we append i to the factor_list, perform integer division num // i, and set is_prime to False.

In this “if” statement, appending i to the factor_list and setting is_prime flag to False take constant time whereas the division of num by i will take O(n^2) time.

If num is not divisible by i, i is incremented by 1. And this addition will take O(n) time.

As the “if” statement takes O(n^2) time and else statement takes O(n) time, for worst case analysis we consider the branch with the highest complexity, here, it being the “if” statement. So here, only the “if” statement is considered for complexity at each iteration. And the number will be in the if condition as long as it is repeatedly divided by i. So the worst case complexity occurs when we have a large number which is power of a small prime number. In this case the if statement is executed logi(num) (log num base i) times. Hence for the worst time complexity scenario, the number of iterations will be log (num) times.

Hence, worst case time complexity = the number of iterations * time complexity of operation at each iteration.

= log(num) * O(n^2) -------(1)

Now, as we know that for any number N, its input size(number of bits) n = log2(N). Conversely N = 2^n.
Substituting this value off 2^n in equation (1), we get
Worst case time complexity = log(2^n) * O(n^2)
As we know, log (2^n) = n
So, the worst-case time complexity is O(n^3).

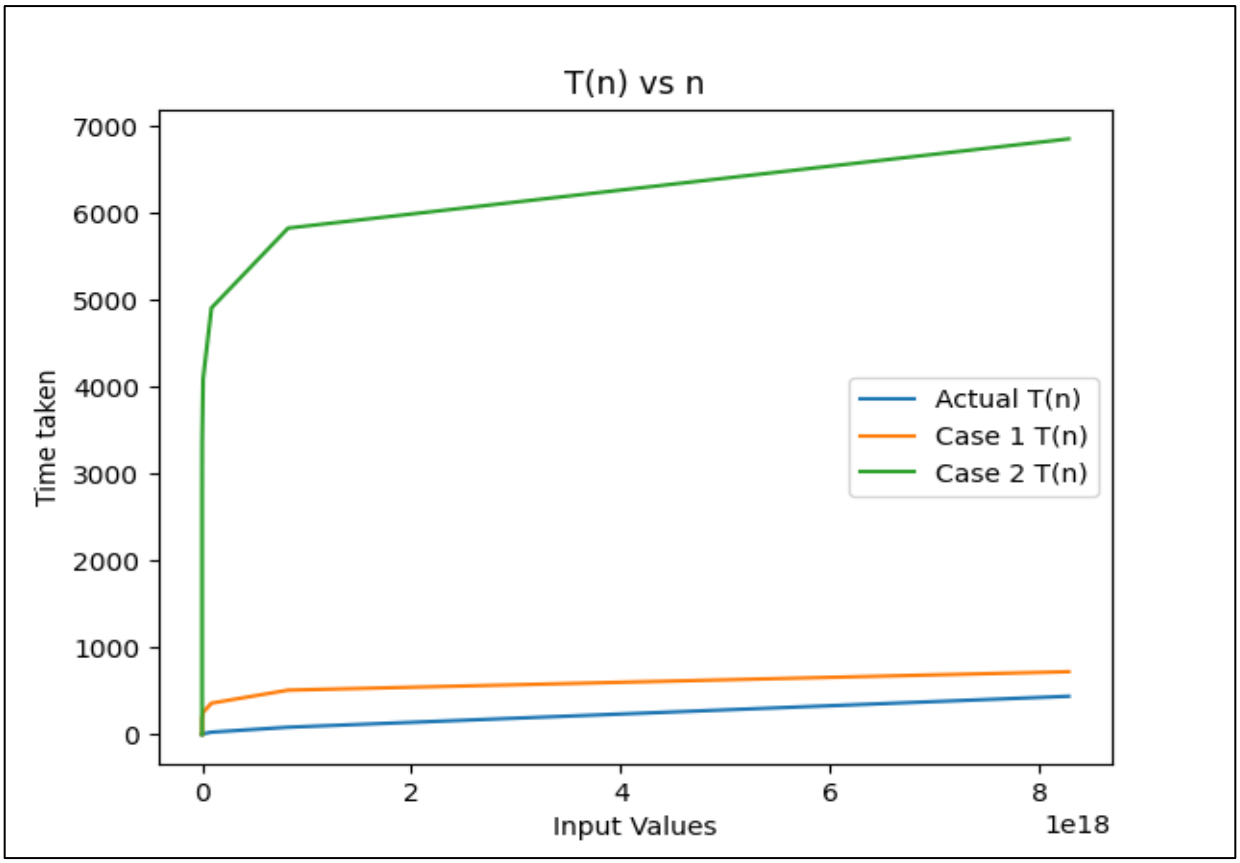
Hence, the worst case time complexity is O(n^3) where n is the number of input bits.

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.

To test the worst time complexity of the algorithm we take prime numbers (as they have to iterate through all values of num) and we consider them in a way that they should have digits in increasing order. Given below is the table which gives the values of T(n) which is the actual running time and the values obtained from formula derived from question 2 –

N(initial input number)	No. of bits(n)	T(N)- Actual running time(in secs)	Case 1 formula = O(sqrt(2^n))	Case 2 formula = O(n^3)
7	1	2.1457672119140625e-06	1.414213562	1
31	2	1.9073486328125e-06	2	8
347	3	4.291534423828125e-06	2.828427125	27
8831	4	1.1205673217773438e-05	4	64
44041	5	2.3603439331054688e-05	5.656854249	125
821479	6	0.00013327598571777344	8	216
2353979	7	0.000202178955078125	11.3137085	343
12625979	8	0.0003905058557128906	16	512
368693491	9	0.002110004425048828	22.627417	729
7386751439	10	0.011230230331420898	32	1000
76948052543	11	0.034651756286621094	45.254834	1331
993663647239	12	0.1201639175415039	64	1728
6286217659789	13	0.24076080322265625	90.50966799	2197
14416872708493	14	0.3456728458404541	128	2744
843466681396517	15	2.683371067047119	181.019336	3375
9893009419892713	16	9.560468912124634	256	4096
90016477798417061	17	28.158084869384766	362.038672	4913
822607206536814947	18	84.93849802017212	512	5832
8283727473827984467	19	441.3893241882324	724.0773439	6859
98553204089564581033	20	1886.4785542488098	1024	8000
181193660667862454621	21	2561.6403789520264	1448.154688	9261

On plotting the graph of input number vs T(n), Case 1 time and case 2 time, we get –



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

To see which function matches the actual running time, we can check the rate of increase of the values from the above table, we can look at how the values change as the number of bits increase. This can be done by examining the data and calculating the ratios of consecutive values T(n+1)/T(n). Let us analyze the 3 cases-

1. We consider T(n) and calculate the ratio for some middle values:

0.00013327598571777344 / 0.000202178955078125 ≈ 0.659

0.000202178955078125 / 0.00039505958557128906 ≈ 0.511

0.00039505958557128906 / 0.002110004425048828 ≈ 0.188

Here, we can see that as n increases, the rate of increase in the value decreases. In other words, the values are not increasing linearly with n, but rather the rate of increase is slowing down.

2. Consider Case 1 formula O(sqrt(2^n))

Calculating the ratio of corresponding values-

8 / 5.656854249 ≈ 1.414213562

11.3137085 / 8 ≈ 1.414213562

16 / 11.3137085 ≈ 1.414213562

22.627417 / 16 ≈ 1.414213562

Here, the calculated ratios are approximately constant, and each ratio is roughly equal to the square root of 2, which is approximately 1.414213562. This suggests that the values in the data are increasing by a constant factor with each increment of n.

3. Consider Case 2 formula O(n^3)

Calculating the ratio of corresponding values-

216 / 125 = 1.728

343 / 216 = 1.587962962962963

512 / 343 = 1.4930555555555556

729 / 512 = 1.423828125

Here, we see that the calculated ratios are not constant but decrease as n increases. This suggests that the rate of increase is not exponential but rather sublinear. The values are increasing, but the rate at which they increase decreases with each increment of n.

Based on all the three-case analysis done above, we see that the rate of increase of T(n) and O(n^3) decreases as n increases but the rate of increase of O(sqrt(2^n)) remains constant with every increase

in n. By this analysis, we can state that T(n) is closely matches with O(n^3) as their rate of increase decrease as n is incremented.

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

From the table, if we consider T(n) column, which is the actual running time, we can see that we have got 441.39 seconds T(n) for 19 digits. This value is the closest to 300 seconds (5*60 seconds), so n has to be either 18 or 19 digits to get the running time to be approx. 5 mins.

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

For T(n) = 5 hours = 5*3600 = 18000 seconds, as the last value is approx. 2500, and 18000 is approx. 7 times 2500, so the number of digits for T(n) to be 5 hours will be approx. 21*7 = 142 digits.

For T(n) = 5 day = 5*24 = 120 hours seconds, and as we know for 5 hours, we have 142 digits, so for 120 hours the number of digits for T(n) will be approx. 142 * 120/5 = 3408 digits.

d. What size of n makes it practically impossible for your algorithm to factorize, e.g., *T(n) >* 10 years.

For T(n) = 10 years = 87600 hours, so as we know for 5 hours we need n = 142 digits, for 87600 hours, n will be approx. 2000000 digits. This size of n will be practically impossible to factorize.

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

A useful invariant of the loop is that the product of factors in the factor list (found till that given point) and the value of number(num) at any given point is equal to the initial input number.

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

Our loop Invariant states that - The product of factors in the factor list found until the current point and the value of `num` at any given point is equal to the initial input number.
i.e.- product(factors) * num == initial num

Initialization:
Before the loop starts, `num` is initialized with the input number, and `factor_list` is empty. Therefore, the product of factors in `factor_list` (which is 1 since it's empty) and `num` (which is the initial input number) is equal to the initial input number. So, the loop invariant holds initially.

Maintenance:
Within the loop, there are two possibilities:

1. If `num` is divisible by `i` (i.e., `num % i == 0`):
  - We append `i` to `factor_list`, which means we're adding a new factor to the product of factors in `factor_list`.
  - We update `num` to `num // i`, which means we're dividing `num` by `i`.
  - Since we've added `i` as a factor and divided `num` by `i`, the product of factors in `factor_list` and the new value of `num` will still be equal to the initial input number. This maintains the loop invariant.

2. If `num` is not divisible by `i`:
  - We increment `i` by 1 to check the next possible factor.

- This doesn't change the product of factors in `factor_list` or the value of `num`.
- So, the loop invariant is still maintained.

Termination:
The loop terminates when the condition `i * i <= num` is no longer true. At this point, `i` has exceeded the square root of the initial `num`, and the loop terminates.

When the loop terminates, the loop invariant still holds because the product of factors in `factor_list` and the final value of `num` will be equal to the initial input number.

Therefore, it is proved that the loop invariant is maintained throughout the loop, and it also holds at termination. This proves the correctness of the algorithm for finding factors of the input number while maintaining the specified loop invariant.

Thank You !