CS 599: Algorithms **(Due: October 1st, 2020)**

Submission Homework 1

Instructor: Prof. Prasad Tadepalli *Name:* Rashmi Jadhav, *Student ID:* 934-069-574

1. Implementation for prime factorization of a number

```
4   def factors(number):
5       start_time = time()
6       output = []                 # output array to store factors
7
8       if number == 1:
9           return output           # 1 is not a prime
10
11      # Reduce the number until it can be divided by 2
12      while number % 2 == 0:      # worst case runs log_2(N) times,  O (log_2 N)
13          output.append(2)
14          number = number // 2
15
16      # For each number from 3 to sqrt n, keep dividing number whenever possible and add to factors
17      # since even numbers are already taken care of by previous loop, increment by 2
18      for i in range(3, int(number ** (1 / 2)) + 1):          # worst case runs sqrt N times, O (sqrt N)
19          while number % i == 0:                              # worst case runs log_3 N times, otherwise log_i N
20              output.append(i)
21              number = number // i
22          i += 2
23
24      if number > 2:                                          # number hasn't been divided by anything, it's prime
25          output.append(number)
26
27      time_taken = time() - start_time
28      times.append(time_taken)
29      return output
30
31
32  def verify(number, factors_list):
33      product = 1
34      for factor in factors_list:
35          product *= factor
36      return number == product
```

2: Time Complexity Analysis

(a) Assuming multiplications (and additions) take constant time (typically for smaller integers with lesser bits:

```
    # Reduce the number until it can be divided by 2
    while number % 2 == 0:      # worst case runs log_2(N) times,  O (log_2 N)
        output.append(2)
        number = number // 2
```

The first while loop runs for at most $\log_2 N$ where N is the input integer (e.g., if number is 256, it will run for $\log_2 256 = 8$ times; 256 →128 →64 →32 →16 →8 →4 →2

```
    # For each number from 3 to sqrt n, keep dividing number whenever possible and add to factors
    # since even numbers are already taken care of by previous loop, increment by 2
    for i in range(3, int(number ** (1 / 2)) + 1):          # worst case runs sqrt N times, O (sqrt N)
        while number % i == 0:                              # worst case runs log_3 N times, otherwise log_i N
            output.append(i)
            number = number // i
        i += 2
```

The outer for loop runs for at most $\sqrt{N}/2$ times as we increment by 2 per iteration. The while loop inside can run for $\log_3 N, \log_5 N, \log_7 N, ..., \log_i N$. This makes it $O(\sqrt{N}/2 * \log N)$. But often times, inner while won't be executed, moreover $\sqrt{N}$ increases faster than $\log N$ and hence we can consider overall upper bound to be $O(\sqrt{N})$

Thus the time complexity for smaller integers with lesser bits and constant time additions/multiplications/divisions is $O(\sqrt{N})$.

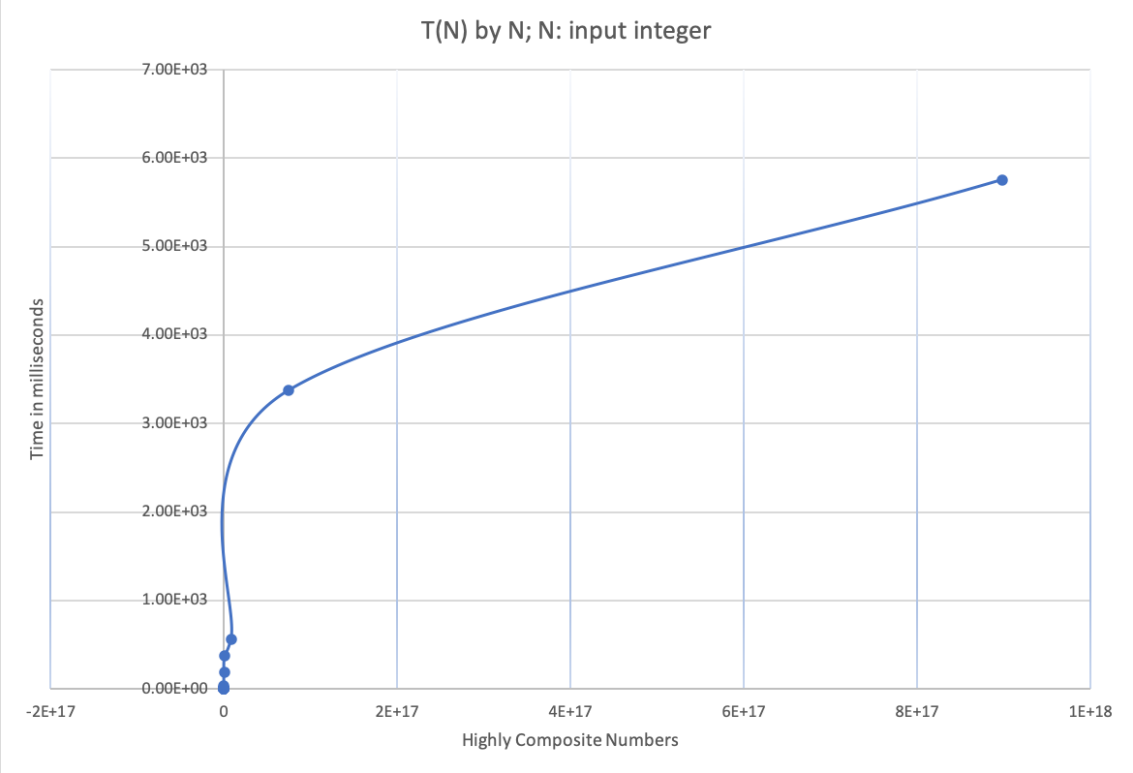
(b) Assuming multiplications (and divisions) of n-bit numbers take $O(n^2)$ and additions and subtractions take $O(n)$ time:
We can say that number of bits n in a decimal number N are:
$n = log_2 N$ (e.g. $log_2 16 = 4$ bits in integer 16).
Therefore, $N = 2^n$.
As per part (a), first while loop now takes $O(\log_2(n^2 * 2^n)) = O(\log_2(n^2) + n)$
The next for loop alone takes $O(\sqrt{2^n} * n) = O(n * 2^{(n/2)})$ and the while loop inside takes
$O(\log_2(n^2 * 2^n)) = O(\log_2(n^2) + n)$. The time complexity now becomes: $O((n * 2^{(n/2)}) * (\log_2(n^2) + n))$
where exponential term increases faster than polynomial and so we asymptotically compute time complexity to be $O(n * 2^{(n/2)})$.

3: Algorithm experimentations and interpretations

1. T(N) vs N where N is input numbers and not bit size

Input integers N	Time taken in milliseconds
6	0.025987625122070312
60	0.00476837158203125
840	0.0050067901611328125
7560	0.0069141387939453125
83160	0.018835067749023438
720720	0.03314018249511719
8648640	0.0629425048828125
73513440	0.2808570861816406
735134400	0.6406307220458984
6983776800	2.3488998413085938
97772875200	7.289886474609375
963761198400	18.944978713989258
9316358251200	57.41286277770996
97821761637600	191.1947727203369
866421317361600	344.6781635284424
8086598962041600	533.689022064209
74801040398884800	3184.960126876831
897612484786617600	5647.605895996094

The above table of numbers exhibits the following curve when plotted:

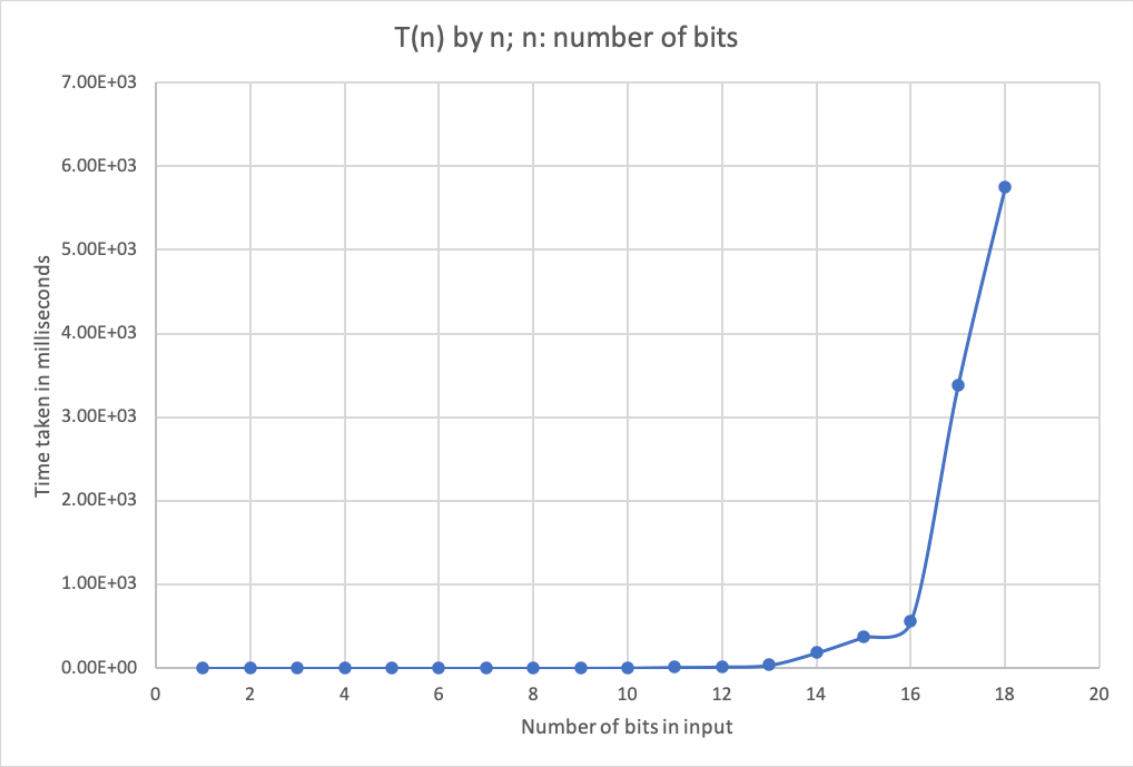


The graph mimics the behavior of function $f(x) = \sqrt{x}$ which pretty much resembles the time complexity when multiplication and division take constant time $O(\sqrt{n})$.

2. T(n) vs n where n is input bit size

Input bit length n	Time taken in milliseconds
1	0.025987625122070312
2	0.00476837158203125
3	0.0050067901611328125
4	0.0069141387939453125
5	0.018835067749023438
6	0.03314018249511719
7	0.0629425048828125
8	0.2808570861816406
9	0.6406307220458984
10	2.3488998413085938
11	7.289886474609375
12	18.944978713989258
13	57.41286277770996
14	191.1947727203369
15	344.6781635284424
16	533.689022064209
17	3184.960126876831
18	5647.605895996094

The above table of numbers exhibits the following curve when plotted:



The graph pretty much resembles the time complexity of multiplication and division of n bit large input integers $O(2^n)$. The graph shoots up exponentially as n increases.

3. The time taken doesn't however depend just on bit length for large numbers. It is quite possible that a highly composite number will be able to factorize quite quicker due to having smaller factors however a number that is the highest prime number under 50 million may take much longer for same number of bits. As per the derived time complexity, the average bit length of the input number should be around 30 bits for it to run for 5 minutes. Surprisingly, n can be around 36 so that T(n) is approximately 5 days. And n can be 42 for T(n) to be 1 year. We will have to wait 10 years to compute factors for a 44 bits integer. This can be considered

practically impossible. Thus we know that as n increases at later stages around 30-40, T(n) exponent shoots up very high and this can probably not be computable for normal computers. Quantum computing however can tackle exponential complexities.